Nome: Otávio Marques Cruz

Assim como em C, C++ permite o gerenciamento de memória, ou seja, a partir de estruturas como ponteiros, armazenando o endereço do dado, pode-se manipular a criação, o acesso, e a destruição deste dado. Este trabalho abordará o gerenciamento de memória, incluindo ponteiros brutos e inteligentes.

4. Os perigos de usar ponteiros brutos, incluindo o risco de vazamentos de memória e ponteiros pendentes.

Devido à utilização padrão dos ponteiros brutos, é necessário gerenciar a memória manualmente, utilizar o *new* para alocar dinamicamente, e utilizar o *delete* para desalocar a memória, liberando aquele espaço ocupado. As vantagens são de não precisar criar uma cópia local do objeto, economizando memória, ao acessar o original, e de alocar dinamicamente no Heap.

Os principais perigos incluem a omissão do `delete`, que pode causar vazamento de memória ao deixar um espaço alocado sem uso. Além disso, podem ocorrer ponteiros pendentes ao acessar um espaço já desalocado, problemas de acessar memória inválida, quando o ponteiro não foi inicializado devidamente, e tentar desalocar um local já desalocado.

1. Os fundamentos dos ponteiros inteligentes C++, incluindo unique_ptr, shared_ptr e weak_ptr.

Os ponteiros inteligentes foram introduzidos para suprir a ausência de um coletor de lixo em C++ e mitigar os problemas dos ponteiros brutos. Quais gerenciam a memória automaticamente, sendo divididos em três tipos principais de ponteiros inteligentes: unique_ptr, shared_ptr, weak_ptr. Cada um tem suas características de propriedades e compartilhamento de recursos.

- unique_ptr: Garante que somente um ponteiro tenha propriedade sobre um recurso.
- shared_ptr: Permite que múltiplos ponteiros compartilhem a propriedade de um recurso.
- weak_ptr: Um ponteiro fraco que não influencia na contagem de referência do shared_ptr, prevenindo ciclos de referência.

2. As diferenças entre ponteiros inteligentes e ponteiros brutos C++ tradicionais, incluindo gerenciamento e propriedade de memória.

A forma habitual seria utilizando ponteiros brutos, como em C, como visto em **Código 1**. Entretanto, essa utilização pode causar vulnerabilidade e vazamentos de memórias, fazendo com que área de memória fique inacessível. Enquanto no **Código 2.** ocorre o gerenciamento automático da memória com o ponteiro inteligente não precisando usar o

new e o delete, para alocar e desalocar do heap. Outras propriedades podem ser observadas em **Tabela 1.**

```C/C++
class Inteiro {
        public:
        int n;
        Inteiro(int n) {this-> n = n;}
};

void UseRawPointer()
{
        Inteiro* pNumero = new Inteiro(2);
        std::cout << *pNumero->n << std::endl; // Saída: 2
        delete pNumero;
}
```

Código 1. Exemplo de ponteiros bruto adaptado de Learn Microsoft

```C/C++
class Inteiro {
        public:
        int n;
        Inteiro(int n) {this-> n = n;}
};

void UseSmartPointer()
{
        // Declaração de um ponteiro inteligente
        std::unique_ptr<Inteiro> Numero = std::make_unique<Inteiro>(42);
        std::cout << *Numero->n << std::endl; // Saída: 42
        return 0;
}
```

Código 2. Exemplo de ponteiros inteligente adaptado de Learn Microsoft

Característica	**Ponteiro Bruto (int*)**	**Ponteiro Inteligente (std::unique_ptr, std::shared_ptr)**
Gerenciamento de memória	**Manual (new e delete)**	**Automático (libera memória quando não há mais referências)**
Segurança	**Propenso a vazamentos, ponteiros pendentes e acessos inválidos**	**Evita vazamentos e acessos inválidos**

Propriedade	**O programador decide quando liberar**	**O ponteiro controla automaticamente a vida útil do objeto**
Desempenho	**Menos sobrecarga, mas maior risco de bugs**	**Leve sobrecarga (contagem de referência), mas mais seguro**
Facilidade de uso	**Exige gerenciamento cuidadoso**	**Mais fácil de usar e manter**

Tabela 1. *Adaptada do chatGPT

Os **ponteiros inteligentes** reduzem o risco de erros no gerenciamento de memória, tornando o código mais seguro e fácil de manter. Enquanto os **ponteiros brutos** oferecem mais controle e melhor desempenho em algumas situações, seu uso exige muito mais cuidado.

5. Unique_ptr e sua função no gerenciamento de propriedade de objeto único.

O unique_ptr gerencia um recurso, fazendo com que ele não possa ser compartilhado fora do escopo, e desaloca-o assim que o sair do escopo, assim permitindo somente um dono.

Isso faz com que seja ideal, para gerenciar recursos exclusivos, como arquivos ou alocações dinâmicas de objetos que não devem ser compartilhados entre diferentes partes do programa, e precisa utilizar o move para transferir a propriedade do recurso para outro ponteiro. Dessa forma evita-se vazamento de memória, veja o exemplo do **Código 3.**

```C/C++
#include <iostream>
#include <memory>
using namespace std;

class Test {
public:
        Test() { cout << "Objeto criado\n"; }
        ~Test() { cout << "Objeto destruído\n"; }
};

int main() {
        unique_ptr<Test> ptr1 = make_unique<Test>(); // Criando unique_ptr
        unique_ptr<Test> ptr2 = std::move(ptr1); // Transferindo propriedade

        if (!ptr1) {cout << "ptr1 não possui mais o objeto\n";}
        return 0; // ptr2 sai de escopo e o objeto é destruído
}
```

Código 3. Exemplo de unique_ptr

6.Shared_ptr e sua função no gerenciamento de propriedade compartilhada.

O shared_ptr é um ponteiro que permite compartilhar o recurso, assim este aponta para um objeto dinamicamente alocado e permite que outros ponteiros passem a referenciá-lo. Existe um contador interno para saber quantos ponteiros referenciam aquele recurso, cada vez que um daqueles ponteiros saem do escopo o contador diminui, e quando todos saem o recurso é desalocado. Veja exemplo no **Código 4.**

```C/C++
#include <iostream>
#include <memory>
using namespace std;

class Test {
public:
	Test() { cout << "Objeto criado\n"; }
	~Test() { cout << "Objeto destruído\n"; }
};

int main() {
	shared_ptr<Test> ptr1 = std::make_shared<Test>(); // Cria o objeto
	shared_ptr<Test> ptr2 = ptr1; // Compartilha o recurso

	cout << "Contagem de referência: " << ptr1.use_count() << endl; // Saída:
Contagem de referência: 2

	return 0;
}
```

Código 4. Exemplo de shared_ptr

Esse tipo de ponteiro pode fazer com que haja o problema que possa ser gerado de referência circular, pois dois ou mais ponteiros apontam criando um círculo, impedindo de ser desalocado automaticamente. Devido isso, é necessário um weak_ptr para resolver, pois ele não incrementa na contagem de referências do shared_ptr, evitando o círculo e o desalocamento.

Bibliografia e Referências:
https://learn.microsoft.com/pt-br/cpp/cpp/raw-pointers?view=msvc-170
https://learn.microsoft.com/pt-br/cpp/cpp/smart-pointers-modern-cpp?view=msvc-170
https://coderslegacy.com/c/cpp-smart-pointers/
https://codelucky.com/cpp-smart-pointers/
https://medium.com/@victorgsampaio/entendendo-c-smart-pointers-d0ca51a87886
ChatGPT Tabela 1. https://chatgpt.com/share/67ddf454-2390-8006-a9e1-21f09aa6be84