Programação Orientada a Objetos

Variáveis e Métodos de Classe (`static`)

1. Introdução

Em C++, a palavra-chave `static` pode ser aplicada a membros de dados (variáveis) e métodos (funções) de uma classe. Quando um membro de classe é declarado como `static`, ele é associado à classe em si, e não a nenhuma instância específica. Isso significa que:

- **Variáveis `static` de classe** são compartilhadas por todas as instâncias da classe.
- **Métodos `static` de classe** podem ser chamados sem a necessidade de uma instância da classe e só podem acessar outros membros `static` da classe.

Esses membros de classe `static` são utilizados para manter informações compartilhadas entre todas as instâncias ou para fornecer funções utilitárias que não precisam de uma instância específica.

Variáveis e métodos `static` em C++ são ferramentas poderosas para gerenciar o estado compartilhado e fornecer funcionalidades utilitárias que não dependem de instâncias de classe. No entanto, seu uso deve ser cuidadosamente planejado para evitar problemas de concorrência e design não modular.

2. Variáveis de Classe `static`

2.1 Conceito

Uma variável de classe `static` é uma variável que é compartilhada por todas as instâncias de uma classe. Em vez de cada objeto ter sua própria cópia da variável, há apenas uma cópia para a classe inteira, acessível a todas as instâncias. Variáveis `static` são úteis para contar o número de instâncias de uma classe, manter dados comuns ou fornecer informações globais que são relevantes para todas as instâncias.

2.2 Exemplo Básico

```
#include <iostream>

class counter {
public:
    static int count;  // Variável de classe (static)

    counter() {
        ++count;  // Incrementa a contagem quando uma nova instância é criada
    }

    ~counter() {
        --count;  // Decrementa a contagem quando uma instância é destruída
    }

    static void show_count() {  // Método de classe (static)
        std::cout << "Contagem: " << count << std::endl;
```

```
    }
};

// Inicialização da variável estática fora da classe
int counter::count = 0;

int main() {
    counter::show_count();  // Saída: Contagem: 0

    counter c1;  // Criação de uma instância
    counter::show_count();  // Saída: Contagem: 1

    {
        counter c2;  // Criação de outra instância
        counter::show_count();  // Saída: Contagem: 2
    }  // c2 é destruído aqui

    counter::show_count();  // Saída: Contagem: 1

    return 0;
}
```

2.3 Explicação

- **Variável `static count`**: Ela é incrementada no construtor e decrementada no destrutor. Como é `static`, ela reflete o número total de instâncias da classe `counter` que estão em existência a qualquer momento.
- **Método `static show_count()`**: Exibe a contagem atual e pode ser chamado diretamente pela classe sem criar uma instância (`counter::show_count()`).

2.4 Pontos Importantes

- **Visibilidade**: Variáveis `static` de classe são geralmente declaradas como `public` para permitir o acesso externo, mas podem ser `private` para controle interno.
- **Inicialização**: Deve ser feita fora da definição da classe, pois é compartilhada por todas as instâncias.

3. Métodos de Classe `static`

3.1 Conceito

Um método de classe `static` pertence à classe em si e não a nenhuma instância específica. Ele pode ser chamado usando o nome da classe e o operador de resolução de escopo (`::`). Métodos `static` não podem acessar variáveis ou métodos não `static` diretamente, pois eles não têm acesso ao `this` (que aponta para a instância atual).

3.2 Exemplo de Métodos `static`

```
#include <iostream>
#include <cmath>
```

```cpp
class math_utils {
public:
    // Método static para calcular o quadrado de um número
    static int square(int x) {
        return x * x;
    }

    // Método static para calcular a raiz quadrada de um número
    static double square_root(double x) {
        return std::sqrt(x);
    }
};

int main() {
    std::cout << "Quadrado de 5: " << math_utils::square(5) << std::endl;  //
Saída: 25
    std::cout << "Raiz quadrada de 16: " << math_utils::square_root(16) <<
std::endl;  // Saída: 4

    return 0;
}
```

3.3 Explicação

- **Método `static square()`**: Calcula o quadrado de um número e pode ser chamado diretamente pela classe.
- **Método `static square_root()`**: Calcula a raiz quadrada de um número e pode ser usado sem criar uma instância da classe `math_utils`.

3.4 Pontos Importantes

- Métodos `static` são úteis para implementar funções utilitárias que operam em dados fornecidos como parâmetros e não requerem acesso ao estado de uma instância específica.

4. Problemas e Considerações de Design

4.1 Limitações dos Métodos e Variáveis `static`

- **Sem Acesso ao Estado de Instância**: Métodos `static` não têm acesso ao `this`, o que significa que não podem acessar membros de instância diretamente.
- **Compartilhamento de Estado**: Variáveis `static` são compartilhadas por todas as instâncias, o que pode levar a problemas de concorrência em contextos multithreaded se não forem usadas corretamente.

5. Comparação do Uso de `static` em C e C++

Embora `static` seja um conceito presente tanto em C quanto em C++, seu uso e comportamento podem diferir entre as duas linguagens devido às suas características distintas. Aqui, vamos explorar as principais comparações entre o uso de `static` em C e C++.

5.1 `static` em C

5.1.1 Variáveis Locais

Em C, uma variável local `static` retém seu valor entre as chamadas da função. Diferente das variáveis locais normais, que são reatribuídas em cada chamada, a variável `static` é inicializada apenas uma vez e preserva seu valor entre as execuções da função.

Exemplo:

```c
#include <stdio.h>

void counter() {
    static int count = 0;  // Variável local estática
    count++;
    printf("Contagem: %d\n", count);
}

int main() {
    counter();  // Saída: Contagem: 1
    counter();  // Saída: Contagem: 2
    counter();  // Saída: Contagem: 3
    return 0;
}
```

5.1.2 Variáveis e Funções Globais

Para variáveis e funções globais, a palavra-chave `static` limita o escopo da variável ou função ao arquivo em que é definida, impedindo que outros arquivos acessem ou se vinculem a ela. Isso é útil para encapsular detalhes de implementação.

Exemplo:

```c
// arquivo1.c
static int hidden = 42;  // Variável global estática

static void secret() {   // Função global estática
    printf("Segredo: %d\n", hidden);
}

void public_function() {
    secret();
}

// arquivo2.c
extern void public_function();  // Declaração da função pública

int main() {
    public_function();  // Pode acessar a função pública, mas não a `hidden` ou
```

```
`secret`
    return 0;
}
```

5.2 `static` em C++

5.2.1 Variáveis e Métodos de Classe

Em C++, o `static` é amplamente utilizado em classes para definir variáveis e métodos que são compartilhados entre todas as instâncias da classe. Este uso é análogo ao conceito de variáveis globais estáticas, mas encapsulado dentro de uma classe.

Exemplo:

```cpp
#include <iostream>

class example {
public:
    static int count;  // Variável de classe estática

    example() {
        ++count;
    }

    static void show_count() {  // Método de classe estático
        std::cout << "Contagem: " << count << std::endl;
    }
};

int example::count = 0;

int main() {
    example e1;
    example e2;
    example::show_count();  // Saída: Contagem: 2
    return 0;
}
```

5.2.2 Escopo e Linkagem

Em C++, o `static` aplicado a variáveis e funções em um arquivo de implementação (`.cpp`) também limita o escopo ao arquivo, de forma semelhante ao que ocorre em C. No entanto, em C++, a palavra-chave `static` também é usada em contexto de classe e funções de classe, adicionando uma nova dimensão à sua funcionalidade.

Exemplo:

```
// file1.cpp
static int hidden = 100;  // Variável global estática

static void private_function() {  // Função global estática
    std::cout << "Hidden: " << hidden << std::endl;
}

void public_function() {
    private_function();
}

// file2.cpp
extern void public_function();  // Declaração da função pública

int main() {
    public_function();  // Pode acessar a função pública, mas não `hidden` ou
`private_function`
    return 0;
}
```

5.3 Comparação e Considerações

- **Escopo**: Em C, `static` afeta variáveis e funções locais, e também pode limitar o escopo global a um arquivo. Em C++, o `static` também se aplica a membros de classe, oferecendo uma maneira de compartilhar dados e métodos entre todas as instâncias da classe.
- **Variáveis Locais**: Em ambas as linguagens, variáveis locais `static` retêm seu valor entre chamadas de função.
- **Encapsulamento e Organização**: C++ usa `static` para oferecer encapsulamento dentro de classes, permitindo melhor organização e modularidade em comparação com C, onde `static` é usado para controlar o escopo e visibilidade de variáveis e funções globais.

Essas diferenças refletem as características de cada linguagem e sua evolução, com C++ oferecendo um conjunto mais rico de funcionalidades orientadas a objetos.

6. Padrão de Projeto Singleton

O padrão de projeto Singleton é um dos padrões de design mais utilizados na programação orientada a objetos. Ele garante que uma classe tenha uma única instância e fornece um ponto de acesso global a essa instância. Esse padrão é útil quando exatamente um objeto deve coordenar ações em todo o sistema.

6.1 Conceito

O padrão Singleton é implementado garantindo que:

- Apenas uma instância da classe seja criada.
- A classe fornece um método global de acesso à sua única instância.

Em C++, isso é comumente feito usando variáveis e métodos `static`. A instância é criada na primeira vez que é necessária e é acessível globalmente por meio de um método `static`.

6.2 Exemplo de Código

Aqui está um exemplo completo e comentado de uma implementação de Singleton em C++:

```cpp
#include <iostream>

// Classe Singleton
class singleton {
private:
    // Ponteiro para a única instância da classe
    static singleton* instance;

    // Construtor privado para evitar criação de instâncias externas
    singleton() {
        std::cout << "Instância criada." << std::endl;
    }

    // Destruidor privado para evitar destruição externa
    ~singleton() {
        std::cout << "Instância destruída." << std::endl;
    }

public:
    // Método de acesso global à única instância
    static singleton* get_instance() {
        if (!instance) {
            instance = new singleton();
        }
        return instance;
    }

    // Método de demonstração
    void show_message() {
        std::cout << "Mensagem do Singleton." << std::endl;
    }

    // Método para destruir a instância (não é sempre necessário)
    static void destroy_instance() {
        delete instance;
        instance = nullptr;
    }
};

// Inicialização do ponteiro estático
singleton* singleton::instance = nullptr;

int main() {
    // Tentativa de criação da instância do Singleton
    singleton* s1 = singleton::get_instance();
    singleton* s2 = singleton::get_instance();

    // Ambos os ponteiros devem apontar para a mesma instância
```

```
    if (s1 == s2) {
        std::cout << "s1 e s2 são a mesma instância." << std::endl;
    }

    s1->show_message();  // Saída: Mensagem do Singleton.

    // Destruição da instância
    singleton::destroy_instance();

    return 0;
}
```

6.3 Explicação

- **Variável Estática** `instance`: Armazena o ponteiro para a única instância da classe `singleton`. É inicializada como `nullptr`.
- **Construtor Privado**: Evita que outras partes do código criem instâncias da classe diretamente.
- **Método** `get_instance()`: Verifica se a instância já foi criada. Se não, cria uma nova instância e a retorna. Se já existir, retorna a instância existente.
- **Método** `show_message()`: Um exemplo de método que pode ser chamado na instância Singleton.
- **Método** `destroy_instance()`: Limpa a instância, permitindo a destruição da mesma quando não for mais necessária.

6.4 Considerações Adicionais

- **Thread Safety**: Em um ambiente multithreaded, o método `get_instance()` deve ser protegido para evitar a criação de múltiplas instâncias. Isso pode ser feito usando mutexes ou outras técnicas de sincronização.
- **Destruição**: A destruição da instância pode não ser necessária em todos os casos, especialmente se a instância for gerenciada pela aplicação inteira. No entanto, é uma boa prática liberar recursos quando a instância não é mais necessária.

O padrão Singleton é amplamente utilizado em situações onde é essencial garantir uma única instância de uma classe, como em gerenciadores de configuração, controladores de acesso a recursos compartilhados e outras áreas onde a coordenação centralizada é necessária.