Programação Orientada a Objetos

Herança

Herança Múltipla, Herança Virtual

A herança múltipla permite que uma classe derive de mais de uma classe base. Esse recurso oferece flexibilidade, mas também pode introduzir complexidade e problemas como ambiguidade e colisão de nomes. A herança virtual é uma técnica utilizada para resolver alguns desses problemas, como o "diamond problem".

A herança múltipla, quando usada corretamente, é uma ferramenta poderosa que oferece flexibilidade para o design de software. No entanto, devido a problemas como ambiguidade e colisões de nomes, é importante entender as técnicas para resolver esses problemas, como a **herança virtual**.

1. Herança Múltipla

Na herança múltipla, uma classe pode herdar características de mais de uma classe base. O C++ suporta herança múltipla, mas é importante usá-la de forma cuidadosa devido à sua complexidade.

Exemplo de Herança Múltipla

```
#include <iostream>

class person {
public:
    void greet() const {
        std::cout << "Olá! Eu sou uma pessoa." << std::endl;
    }
};

class employee {
public:
    void work() const {
        std::cout << "Estou trabalhando." << std::endl;
    }
};

// A classe manager herda de ambas, person e employee.
class manager : public person, public employee {
public:
    void manage() const {
        std::cout << "Estou gerenciando a equipe." << std::endl;
    }
};

int main() {
    manager m;
    m.greet();  // Chama greet() de person
```

```
    m.work();   // Chama work() de employee
    m.manage(); // Chama manage() de manager
    return 0;
}
```

Explicação

- **Classe `manager`**: Herda de `person` e `employee`. A instância `m` de `manager` pode acessar métodos de ambas as classes base.

2. Problemas da Herança Múltipla e Soluções

A herança múltipla pode introduzir problemas como:

- **Ambiguidade de Métodos**: Se duas classes base tiverem métodos ou atributos com o mesmo nome, ocorre uma ambiguidade.
- **Colisão de Nomes de Membros**: Quando duas classes base possuem membros com o mesmo nome (atributos ou métodos), o compilador não sabe qual usar.

Exemplo de Ambiguidade de Métodos

```
#include <iostream>

class base_a {
public:
    void print() const {
        std::cout << "Base A" << std::endl;
    }
};

class base_b {
public:
    void print() const {
        std::cout << "Base B" << std::endl;
    }
};

class derived : public base_a, public base_b {};

int main() {
    derived d;
    // d.print();  // Erro de ambiguidade!
    d.base_a::print();  // Resolvendo ambiguidade chamando explicitamente.
    d.base_b::print();  // Resolvendo ambiguidade chamando explicitamente.
    return 0;
}
```

Explicação

- **Erro de Ambiguidade**: O compilador não sabe se deve chamar `print()` de `base_a` ou `base_b`.
- **Resolução**: A chamada explícita `d.base_a::print()` e `d.base_b::print()` resolve a ambiguidade.

Solução para Colisão de Nomes de Membros

Se houver colisão de nomes, você pode:

1. **Usar Qualificação de Nomes**: Referencie os membros das classes base explicitamente como mostrado no exemplo acima.
2. **Redefinir Membros**: Redefina o membro na classe derivada.

3. Herança Virtual

A **herança virtual** é usada para resolver o problema do diamante na herança múltipla, onde a mesma classe base é herdada mais de uma vez por meio de diferentes caminhos.

Exemplo do Problema do Diamante e Herança Virtual

```
#include <iostream>

class device {
public:
    void power_on() const {
        std::cout << "Dispositivo ligado." << std::endl;
    }
};

// Herança virtual usada para evitar múltiplas cópias de 'device'
class printer : public virtual device {};

class scanner : public virtual device {};

class multifunction_device : public printer, public scanner {};

int main() {
    multifunction_device mfd;
    mfd.power_on();  // Sem ambiguidade!
    return 0;
}
```

Explicação

- **Classe `device`**: Base para `printer` e `scanner`.
- **Herança Virtual**: `printer` e `scanner` herdam `device` virtualmente, evitando duplicação.
- **Classe `multifunction_device`**: Herda de `printer` e `scanner` sem ambiguidade.

Exemplo: Sistema de Notificações

Vamos criar um exemplo de um sistema de notificações que usa herança múltipla e herança virtual.

Código Exemplo

```cpp
#include <iostream>

// Classe base para notificações
class notification {
public:
    virtual void send() const = 0;  // Função virtual pura
    virtual ~notification() {}  // Destrutor virtual
};

// Herança virtual para evitar ambiguidade
class email_notification : public virtual notification {
public:
    void send() const override {
        std::cout << "Enviando notificação por e-mail." << std::endl;
    }
};

class sms_notification : public virtual notification {
public:
    void send() const override {
        std::cout << "Enviando notificação por SMS." << std::endl;
    }
};

// Herança múltipla, combinando notificações por e-mail e SMS
class email_sms_notification : public email_notification, public sms_notification
{
public:
    void send() const override {
        email_notification::send();
        sms_notification::send();
    }
};

int main() {
    email_sms_notification notifier;
    notifier.send();  // Envia notificações por e-mail e SMS

    return 0;
}
```

Explicação

- **Classe** `notification`: Interface para envio de notificações.
- **Herança Virtual**: `email_notification` e `sms_notification` herdam virtualmente de `notification`.
- **Herança Múltipla**: `email_sms_notification` combina notificações por e-mail e SMS.

4. Construtores e Herança Virtual

Na herança múltipla com herança virtual, o construtor da classe base (neste caso, `animal`) é chamado pela classe derivada mais distante (neste caso, `bat`). Se não houver um construtor padrão disponível (ou seja, um construtor sem argumentos) na classe base, e a classe derivada não fornecer uma chamada explícita para um construtor da classe base que aceite argumentos, isso resultará em um erro de compilação.

Quando usamos herança virtual, o construtor da classe base precisa ser chamado pela classe derivada mais distante. O compilador, por padrão, tenta chamar o construtor padrão da classe base, a menos que seja especificado de outra forma. Se o construtor padrão não estiver disponível e não houver uma chamada explícita para outro construtor, o compilador não saberá qual construtor usar e gerará um erro.

Exemplo de Código Demonstrando o Problema

```
#include <iostream>

class animal {
public:
    // Construtor padrão ausente; presença de um construtor com um argumento
    animal(int age) {
        std::cout << "animal: Construtor com idade = " << age << " chamado." <<
std::endl;
    }
};

class mammal : virtual public animal {
public:
    // Construtor de 'mammal' sem chamada explícita ao construtor de 'animal'
    mammal() {
        std::cout << "mammal: Construtor chamado." << std::endl;
    }
};

class bird : virtual public animal {
public:
    bird() {
        std::cout << "bird: Construtor chamado." << std::endl;
    }
};

class bat : public mammal, public bird {
public:
    // Construtor de 'bat' sem chamada explícita ao construtor de 'animal'
    bat() {
        std::cout << "bat: Construtor chamado." << std::endl;
    }
};

int main() {
    bat b;  // Isso resultará em um erro de compilação!
    return 0;
}
```

Explicação do Erro

- Quando tentamos criar uma instância de `bat`, o compilador precisa chamar o construtor da classe `animal` porque `animal` é herdada virtualmente.
- A classe `bat` precisa decidir qual construtor de `animal` chamar. No entanto, como não há um construtor padrão em `animal` (e nenhum outro construtor é especificado explicitamente para ser chamado), o compilador não sabe qual construtor usar.
- Isso leva a um **erro de compilação**.

Como Resolver o Problema

Existem duas formas principais de resolver o problema:

Solução 1: Adicionar um Construtor Padrão em `animal`

Uma solução simples é adicionar um construtor padrão (sem argumentos) na classe base `animal`:

```
class animal {
public:
    animal() {
        std::cout << "animal: Construtor padrão chamado." << std::endl;
    }

    animal(int age) {
        std::cout << "animal: Construtor com idade = " << age << " chamado." <<
std::endl;
    }
};
```

Solução 2: Chamar Explicitamente o Construtor da Classe Base

Outra solução é chamar explicitamente um construtor de `animal` que aceite argumentos a partir do construtor de `bat`:

```
#include <iostream>

class animal {
public:
    animal(int age) {
        std::cout << "animal: Construtor com idade = " << age << " chamado." <<
std::endl;
    }
};

class mammal : virtual public animal {
public:
    mammal(int age) : animal(age) {  // Chamada explícita para o construtor de
'animal'
```

```
        std::cout << "mammal: Construtor chamado." << std::endl;
    }
};

class bird : virtual public animal {
public:
    bird(int age) : animal(age) {  // Chamada explícita para o construtor de
'animal'
        std::cout << "bird: Construtor chamado." << std::endl;
    }
};

class bat : public mammal, public bird {
public:
    bat(int age) : animal(age), mammal(age), bird(age) {  // Chamada explícita
para o construtor de 'animal'
        std::cout << "bat: Construtor chamado." << std::endl;
    }
};

int main() {
    bat b(5);  // Funcionará corretamente
    return 0;
}
```

- **Construtor Padrão Necessário?**: Não é sempre necessário ter um construtor padrão, mas se ele não estiver presente, é necessário chamar explicitamente um construtor da classe base no construtor da classe derivada mais distante.
- **Por Que o Construtor Padrão É Comum?**: O construtor padrão é frequentemente adicionado para evitar esse tipo de erro de compilação, facilitando o uso de herança múltipla.
- **Uso Correto da Herança Virtual**: Ao usar herança virtual, tenha sempre em mente como os construtores devem ser chamados, especialmente quando há mais de um construtor na classe base.