Sobrecarga de Operadores em C++

Sumário

---

1. Introdução

1.1 O que é Sobrecarga de Operadores?

Sobrecarga de operadores permite definir o comportamento dos operadores padrão (`+`, `-`, `*`, etc.) para tipos definidos pelo usuário, tornando o código mais intuitivo e legível.

1.2 Benefícios e Usos

- **Legibilidade**: Escrever `a + b` em vez de `a.adicionar(b)`
- **Intuitividade**: Uso consistente com tipos primitivos
- **Integração**: Compatibilidade com algoritmos da biblioteca padrão
- **Expressividade**: Código mais conciso e claro

1.3 Regras Básicas e Restrições

A sobrecarga de operadores em C++ oferece flexibilidade, mas também possui regras e restrições importantes que devem ser seguidas:

1. **Não é possível criar novos operadores**:
    - Não se pode inventar operadores como `**` para potenciação ou `<>` para alguma operação personalizada
    - Apenas os operadores existentes na linguagem podem ser sobrecarregados
2. **Não se pode alterar a precedência, associatividade ou aridade dos operadores**:
    - O operador `+` continuará tendo a mesma precedência em relação ao `*`
    - Um operador unário permanece unário, e um operador binário permanece binário
    - A ordem de avaliação não muda (por exemplo, `a + b + c` ainda é avaliado como `(a + b) + c`)
3. **Requisitos sobre os operandos**:
    - Pelo menos um dos operandos deve ser de um tipo definido pelo usuário
    - Não é possível redefinir operadores para tipos fundamentais como `int` ou `double`

   - Exemplo: podemos definir `MyClass + int` ou `int + MyClass`, mas não `int + int`

4. **Operadores que não podem ser sobrecarregados**:
   - `::` (operador de resolução de escopo)
   - `.` (operador de acesso a membro direto)
   - `.*` (ponteiro para membro)
   - `?:` (operador ternário)
   - `sizeof` (operador para obter tamanho)
   - `typeid` (operador RTTI)
   - `alignof` (C++11, para obter alinhamento de tipo)

5. **Operadores que devem ser obrigatoriamente membros da classe**:
   - `=` (atribuição)
   - `()` (chamada de função)
   - `[]` (acesso a elementos)
   - `->` (acesso a membro via ponteiro)
   - `T&` (conversão de tipo para referência)

6. **Preservação da semântica básica**:
   - É recomendado preservar a semântica básica do operador
   - Por exemplo, `+` deve ter um comportamento "aditivo", `==` deve verificar equivalência
   - Alterar drasticamente o comportamento esperado pode levar a código confuso e propenso a erros

7. **Atenção à sobrecarga excessiva**:
   - Sobrecarregar operadores em excesso pode tornar o código difícil de entender
   - A sobrecarga deve ser intuitiva e melhorar a legibilidade

8. **Não é possível controlar a precedência de operadores personalizados**:
   - A precedência dos operadores sobrecarregados segue a precedência dos operadores originais
   - Por exemplo, `*` sempre terá precedência sobre `+` independentemente de como você os sobrecarrega

1.4 Operadores que Podem Ser Sobrecarregados

Em C++, os seguintes operadores podem ser sobrecarregados:

- **Aritméticos**: `+`, `-`, `*`, `/`, `%`
- **Relacionais**: `==`, `!=`, `<`, `>`, `<=`, `>=`
- **Lógicos**: `&&`, `||`, `!`
- **Bit a bit**: `&`, `|`, `^`, `~`, `<<`, `>>`
- **Atribuição**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- **Incremento/decremento**: `++`, `--`
- **Acesso a arrays**: `[]`
- **Desreferenciação**: `*` (ponteiros), `->`
- **Conversão de tipos**: operando como funções de conversão explícita

1.5 Formas de Sobrecarga: Membro vs. Função Externa

Em C++, existem duas maneiras principais de sobrecarregar operadores, cada uma com características e casos de uso específicos:

1.5.1 Sobrecarga como Método Membro

Quando sobrecarregamos um operador como método membro, ele se torna parte da definição da classe:

```
class String {
public:
    // Sobrecarga do operador + como método membro
    String operator+(const String& other) const {
        // Implementação
    }
};

// Uso
String s1, s2;
String s3 = s1 + s2;  // Chama s1.operator+(s2)
```

Características importantes:

- O operando à esquerda é sempre o objeto atual (this)
- Tem acesso direto aos membros privados e protegidos da classe
- É obrigatório para operadores como `=`, `()`, `[]`, `->` e conversões de tipo
- Recebe um parâmetro a menos que o número normal de operandos (porque o objeto atual é o primeiro operando)

1.5.2 Sobrecarga como Função Externa (Global ou Friend)

Quando sobrecarregamos como função externa, o operador é definido fora da classe:

```
class Complex {
private:
    double real, imag;
public:
    // Declarando como friend para acesso a membros privados
    friend Complex operator+(const Complex& lhs, const Complex& rhs);
};

// Implementação da função externa
Complex operator+(const Complex& lhs, const Complex& rhs) {
    return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}

// Uso
Complex c1, c2;
Complex c3 = c1 + c2;  // Chama operator+(c1, c2)
```

Características importantes:

- Recebe todos os operandos como parâmetros
- Não tem acesso direto aos membros privados/protegidos (a menos que seja declarada como friend)
- Permite que o operando à esquerda seja de um tipo diferente da classe
- É essencial para operadores onde o operando à esquerda não é do tipo da classe

1.5.3 Critérios para Escolher entre Membro e Função Externa

1. **Obrigatoriamente como membro**:
    - Operador de atribuição `=`
    - Operador de chamada `()`
    - Operador de acesso a elemento `[]`
    - Operador de acesso a membro `->`
    - Operadores de conversão de tipo
2. **Preferencialmente como função externa**:
    - Quando o operando à esquerda não é do tipo da classe
    - Operadores binários que devem suportar conversão implícita no primeiro operando
    - Operadores simétricos (como `+`, `-`, `*`, etc.) para permitir comutatividade
3. **Preferencialmente como membro**:
    - Operadores que modificam o estado do objeto (como `+=`, `-=`, etc.)
    - Operadores unários que operam no objeto (como `++`, `--`, `~`)
    - Quando o operador está intimamente ligado ao estado interno do objeto

1.5.4 Exemplo Comparativo

```
class Number {
private:
    int value;

public:
    Number(int v) : value(v) {}

    // Como método membro
    Number operator+(const Number& other) const {
        return Number(value + other.value);
    }

    // Operador multiplicação como friend para ilustração
    friend Number operator*(const Number& lhs, const Number& rhs);

    // Getter para o valor
    int getValue() const { return value; }
};
```

```cpp
// Implementação do operador * como função friend
Number operator*(const Number& lhs, const Number& rhs) {
    return Number(lhs.value * rhs.value);
}

// Operador para permitir: int + Number (precisa ser função externa)
Number operator+(int lhs, const Number& rhs) {
    return Number(lhs + rhs.getValue());
}
```

Este exemplo ilustra como a escolha entre método membro e função externa afeta a flexibilidade e uso dos operadores sobrecarregados.

1.6 Exemplo Básico

```cpp
class Complex {
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Método membro para adição
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    // Função friend para multiplicação
    friend Complex operator*(const Complex& lhs, const Complex& rhs);

    // Exibição formatada
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
};

// Implementação da função friend
Complex operator*(const Complex& lhs, const Complex& rhs) {
    return Complex(
        lhs.real * rhs.real - lhs.imag * rhs.imag,
        lhs.real * rhs.imag + lhs.imag * rhs.real
    );
}

std::ostream& operator<<(std::ostream& os, const Complex& c) {
    return os << c.real << (c.imag >= 0 ? " + " : " - ")
              << std::abs(c.imag) << "i";
}
```

2. Operadores Unários

Os operadores unários operam sobre um único operando, modificando ou transformando seu valor. Em C++, a sobrecarga desses operadores permite personalizar seu comportamento para tipos definidos pelo usuário, tornando a manipulação de objetos mais intuitiva e natural.

2.1 Operadores de Incremento (++ e --)

Utilidade e funcionamento: Estes operadores permitem incrementar ou decrementar um valor de forma compacta. A sobrecarga deles é particularmente útil para classes que representam contadores, iteradores, ou qualquer tipo com um conceito de "próximo" ou "anterior". Um detalhe importante é a distinção entre as versões prefixadas (++x) e pós-fixadas (x++).

```
class Counter {
private:
    int value;

public:
    Counter(int v = 0) : value(v) {}

    // Prefixo (++x) - retorna referência após incremento
    Counter& operator++() {
        ++value;
        return *this;
    }

    // Pós-fixo (x++) - retorna cópia antes do incremento
    // O parâmetro int é apenas para diferenciar da versão prefixada
    Counter operator++(int) {
        Counter temp = *this;
        ++value;
        return temp;
    }

    int getValue() const { return value; }
};
```

2.2 Operadores de Sinal (+, -)

Utilidade e funcionamento: Estes operadores permitem expressar um valor positivo (+x) ou negativo (-x) de um objeto. São frequentemente utilizados em classes matemáticas como vetores, números complexos ou qualquer tipo que tenha um conceito de "inversão de sinal". O operador unário + geralmente retorna uma cópia não modificada, enquanto o - retorna uma cópia com sinal invertido.

```
class Vector {
private:
    double x, y;

public:
    Vector(double x_val = 0, double y_val = 0) : x(x_val), y(y_val) {}
```

```
    // Operador unário +
    Vector operator+() const {
        return *this;  // Retorna uma cópia sem alteração
    }

    // Operador unário -
    Vector operator-() const {
        return Vector(-x, -y);  // Inverte o vetor
    }
};
```

2.3 Operador de Negação Lógica (!)

Utilidade e funcionamento: Este operador inverte um valor booleano, transformando verdadeiro em falso e vice-versa. É útil para classes que representam estados binários, condições ou expressões lógicas. Quando sobrecarregado, permite que objetos sejam usados diretamente em expressões condicionais.

```
class Optional {
private:
    bool has_value;
    int value;

public:
    Optional() : has_value(false), value(0) {}
    Optional(int v) : has_value(true), value(v) {}

    // Operador ! (verdadeiro se não tiver valor)
    bool operator!() const {
        return !has_value;
    }

    // Conversão para bool (verdadeiro se tiver valor)
    explicit operator bool() const {
        return has_value;
    }
};
```

2.4 Operador de Complemento de Bits (~)

Utilidade e funcionamento: Este operador inverte todos os bits de um valor, transformando 0s em 1s e vice-versa. É particularmente útil para classes que representam conjuntos de flags, máscaras de bits ou estruturas de dados que trabalham em nível de bit. Permite manipulações de bits de forma intuitiva, similar aos tipos integrais.

```
class BitwiseData {
private:
    int value;
```

```
public:
    BitwiseData(int val = 0) : value(val) {}

    // Operador ~ (complemento de bits)
    BitwiseData operator~() const {
        return BitwiseData(~value);
    }

    int getValue() const { return value; }
};
```

3. Operadores Binários

Operadores binários trabalham com dois operandos, combinando-os de alguma forma para produzir um novo valor. A sobrecarga desses operadores permite que objetos de classes personalizadas interajam entre si de maneira intuitiva e consistente com a sintaxe padrão da linguagem.

3.1 Operadores Aritméticos (+, -, *, /, %)

Utilidade e funcionamento: Estes operadores realizam operações matemáticas básicas entre dois valores. A sobrecarga permite que classes que representam valores numéricos, como números complexos, frações, vetores ou matrizes, possam ser manipuladas com a mesma naturalidade dos tipos primitivos. Cada operador deve implementar a operação matemática correspondente para o domínio específico da classe.

```
class Fraction {
private:
    int num, den;

    // Simplifica a fração
    void simplify() {
        // Implementação omitida para brevidade
    }

public:
    Fraction(int n = 0, int d = 1) : num(n), den(d) {
        if (den == 0) throw std::invalid_argument("Denominator cannot be zero");
        simplify();
    }

    // Operador de adição
    Fraction operator+(const Fraction& other) const {
        return Fraction(
            num * other.den + other.num * den,
            den * other.den
        );
    }

    // Operador de multiplicação
    Fraction operator*(const Fraction& other) const {
        return Fraction(
```

```
            num * other.num,
            den * other.den
        );
    }

    // Outros operadores seguiriam lógica similar
};
```

3.2 Operadores de Comparação (==, !=, <, >, <=, >=)

Utilidade e funcionamento: Estes operadores definem relações de igualdade e ordem entre objetos. São essenciais para permitir que objetos de uma classe personalizada sejam comparados entre si, usados em contêineres ordenados (como `std::set` ou `std::map`), ou em algoritmos de ordenação da STL. Implementá-los de forma consistente é crucial para comportamento previsível em estruturas de dados e algoritmos.

```
class Temperature {
private:
    double celsius;

public:
    Temperature(double c = 0) : celsius(c) {}

    // Operador ==
    bool operator==(const Temperature& other) const {
        return std::abs(celsius - other.celsius) < 0.001;
    }

    // Operador <
    bool operator<(const Temperature& other) const {
        return celsius < other.celsius;
    }

    // Outros operadores podem ser definidos em termos dos acima
    bool operator!=(const Temperature& other) const {
        return !(*this == other);
    }

    bool operator>(const Temperature& other) const {
        return other < *this;
    }

    bool operator<=(const Temperature& other) const {
        return !(other < *this);
    }

    bool operator>=(const Temperature& other) const {
        return !(*this < other);
    }
};
```

3.3 Operadores de Atribuição Composta (+=, -=, *=, /=, etc.)

Utilidade e funcionamento: Estes operadores combinam uma operação aritmética ou de bits com atribuição, modificando o operando à esquerda. São úteis para realizar modificações in-place (sem criar cópias desnecessárias), tornando o código mais eficiente e conciso. Quando sobrecarregados, devem retornar uma referência para o objeto modificado (`*this`), permitindo encadeamento de operações.

```
class String {
private:
    char* data;
    size_t length;

    // Métodos de utilitário omitidos

public:
    // Construtor e destrutor omitidos

    // Operador +=
    String& operator+=(const String& other) {
        // Aloca novo buffer para a concatenação
        char* new_data = new char[length + other.length + 1];

        // Copia os dados existentes
        std::strcpy(new_data, data);
        // Concatena os novos dados
        std::strcat(new_data, other.data);

        // Libera o buffer antigo
        delete[] data;

        // Atualiza os ponteiros e o tamanho
        data = new_data;
        length += other.length;

        return *this;
    }

    // Definindo + em termos de +=
    String operator+(const String& other) const {
        String result = *this;  // Cria cópia
        result += other;        // Utiliza +=
        return result;
    }
};
```

3.4 Operadores Lógicos (&&, ||)

Utilidade e funcionamento: Estes operadores implementam as operações lógicas AND e OR, tipicamente resultando em um valor booleano. São úteis para classes que representam expressões condicionais,

predicados ou qualquer tipo que possua um conceito de "valor verdade". A sobrecarga permite que objetos sejam usados diretamente em expressões condicionais, melhorando a legibilidade do código.

```
class BooleanValue {
private:
    bool value;

public:
    BooleanValue(bool val = false) : value(val) {}

    // Operador && (AND lógico)
    bool operator&&(const BooleanValue& other) const {
        return value && other.value;
    }

    // Operador || (OR lógico)
    bool operator||(const BooleanValue& other) const {
        return value || other.value;
    }

    // Operador ! (NOT lógico)
    bool operator!() const {
        return !value;
    }

    bool getValue() const { return value; }
};
```

3.5 Operadores de Bits (&, |, ^, <<, >>)

Utilidade e funcionamento: Estes operadores realizam manipulações a nível de bits. São extremamente úteis para classes que representam conjuntos de flags, máscaras de bits, ou qualquer tipo que precise de manipulação em nível de bit. Também são frequentemente usados para implementar operações vetoriais ou otimizações de baixo nível.

```
class BitFlags {
private:
    unsigned int flags;

public:
    BitFlags(unsigned int f = 0) : flags(f) {}

    // Operador & (AND bit a bit)
    BitFlags operator&(const BitFlags& other) const {
        return BitFlags(flags & other.flags);
    }

    // Operador | (OR bit a bit)
    BitFlags operator|(const BitFlags& other) const {
        return BitFlags(flags | other.flags);
```

```
    }

    // Operador ^ (XOR bit a bit)
    BitFlags operator^(const BitFlags& other) const {
        return BitFlags(flags ^ other.flags);
    }

    // Operador << (shift à esquerda)
    BitFlags operator<<(int shift) const {
        return BitFlags(flags << shift);
    }

    // Operador >> (shift à direita)
    BitFlags operator>>(int shift) const {
        return BitFlags(flags >> shift);
    }

    unsigned int getValue() const { return flags; }
};
```

3.6 Operadores de Stream (<<, >>)

Utilidade e funcionamento: Estes operadores permitem a entrada e saída formatada de objetos. A sobrecarga do operador `<<` para `std::ostream` permite que objetos sejam impressos diretamente com `std::cout`, enquanto a sobrecarga do operador `>>` para `std::istream` permite leitura de objetos com `std::cin`. São essenciais para debugging, logging, serialização e deserialização de objetos personalizados.

```
class Point {
private:
    double x, y;

public:
    Point(double x_val = 0, double y_val = 0) : x(x_val), y(y_val) {}

    // Operador de stream de saída
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }

    // Operador de stream de entrada
    friend std::istream& operator>>(std::istream& is, Point& p) {
        // Permitindo formato como "(x,y)"
        char c1, c2, c3;
        is >> c1 >> p.x >> c2 >> p.y >> c3;

        if (c1 != '(' || c2 != ',' || c3 != ')') {
            is.setstate(std::ios::failbit);
        }

        return is;
    }
```

```
};
```if (c1 != '(' || c2 != ',' || c3 != ')') {
            is.setstate(std::ios::failbit);
        }

        return is;
    }
};
```

4. Operadores Especiais

Além dos operadores unários e binários comuns, o C++ permite a sobrecarga de operadores especiais que possuem comportamentos e semânticas específicas. Estes operadores dão às classes personalizadas capacidades avançadas como comportamento de função, acesso indexado e conversões de tipo.

4.1 Operador de Chamada de Função ()

Utilidade e funcionamento: O operador de chamada de função `()` permite que objetos sejam usados como funções. Isto cria o que chamamos de "functors" ou objetos função, que podem armazenar estado entre chamadas e ser parametrizados na construção. São extremamente úteis em programação genérica, algoritmos da STL e implementação de callbacks. Um functor pode ser passado para algoritmos como `std::sort` ou `std::for_each`, combinando dados e comportamento em uma única entidade.

```
class Multiplier {
private:
    double factor;

public:
    Multiplier(double f = 1.0) : factor(f) {}

    // Operador de chamada de função
    double operator()(double value) const {
        return value * factor;
    }
};

// Uso:
Multiplier times2(2.0);
double result = times2(4.0);  // result = 8.0
```

4.2 Operador de Indexação []

Utilidade e funcionamento: O operador de indexação `[]` permite acessar elementos de um objeto através de um índice ou chave, como em arrays ou contêineres. É essencial para implementar classes como vetores dinâmicos, matrizes, mapas ou qualquer estrutura de dados com acesso indexado. Quando sobrecarregado, pode incluir verificação de limites e outras validações, tornando o acesso mais seguro que arrays nativos.
```

```cpp
class SafeArray {
private:
    int* data;
    size_t size;

public:
    SafeArray(size_t s) : size(s) {
        data = new int[size]();  // Inicializa com zeros
    }

    ~SafeArray() { delete[] data; }

    // Operador de indexação (versão não-const)
    int& operator[](size_t index) {
        if (index >= size) {
            throw std::out_of_range("Index out of bounds");
        }
        return data[index];
    }

    // Operador de indexação (versão const)
    const int& operator[](size_t index) const {
        if (index >= size) {
            throw std::out_of_range("Index out of bounds");
        }
        return data[index];
    }
};
```

4.3 Operadores de Conversão

Utilidade e funcionamento: Os operadores de conversão permitem que objetos de uma classe sejam convertidos implícita ou explicitamente para outros tipos. São úteis para criar classes que possam ser usadas naturalmente em contextos que esperam outros tipos, como expressões numéricas ou booleanas. A palavra-chave `explicit` pode ser usada para evitar conversões implícitas indesejadas, exigindo um cast explícito do programador.

```cpp
class Angle {
private:
    double radians;

public:
    Angle(double rad = 0.0) : radians(rad) {}

    // Construtor para conversão de graus
    static Angle fromDegrees(double degrees) {
        return Angle(degrees * M_PI / 180.0);
    }

    // Operador de conversão implícita para double (radianos)
```

```
    operator double() const {
        return radians;
    }

    // Conversão explícita para int (graus arredondados)
    explicit operator int() const {
        return static_cast<int>(radians * 180.0 / M_PI + 0.5);
    }
};

// Uso:
Angle a = Angle::fromDegrees(45);
double rad = a;                    // Conversão implícita para double
int deg = static_cast<int>(a);     // Conversão explícita para int
```

4.4 Operador de Atribuição =

Utilidade e funcionamento: O operador de atribuição = define como um objeto recebe os valores de outro objeto do mesmo tipo. É crucial para o gerenciamento adequado de recursos, especialmente para classes que possuem membros que apontam para memória dinamicamente alocada (ponteiros). A implementação correta deve lidar com auto-atribuição, liberação de recursos antigos e cópia profunda, garantindo que o objeto mantenha um estado consistente.

```
class DynamicArray {
private:
    int* data;
    size_t size;

public:
    // Construtor
    DynamicArray(size_t s = 0) : size(s) {
        data = (size > 0) ? new int[size]() : nullptr;
    }

    // Destrutor
    ~DynamicArray() {
        delete[] data;
    }

    // Construtor de cópia
    DynamicArray(const DynamicArray& other) : size(other.size) {
        data = (size > 0) ? new int[size] : nullptr;
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }
    }

    // Operador de atribuição (cópia profunda)
    DynamicArray& operator=(const DynamicArray& other) {
        // Verificação de auto-atribuição
        if (this == &other) return *this;
```

```
        // Libera memória antiga
        delete[] data;

        // Aloca nova memória e copia dados
        size = other.size;
        data = (size > 0) ? new int[size] : nullptr;
        for (size_t i = 0; i < size; ++i) {
            data[i] = other.data[i];
        }

        return *this;
    }

    // Operador de atribuição de movimento (C++11)
    DynamicArray& operator=(DynamicArray&& other) noexcept {
        // Verificação de auto-atribuição
        if (this == &other) return *this;

        // Libera memória antiga
        delete[] data;

        // Move os recursos
        data = other.data;
        size = other.size;

        // Deixa o outro objeto em estado válido mas vazio
        other.data = nullptr;
        other.size = 0;

        return *this;
    }
};
```

4.5 Operadores de Desreferenciação (*, ->)

Utilidade e funcionamento: Estes operadores são essenciais para a implementação de ponteiros inteligentes (smart pointers) e iteradores. O operador `*` (desreferenciação) acessa o objeto apontado, enquanto o operador `->` permite acessar membros do objeto apontado. A sobrecarga destes operadores permite criar abstrações que se comportam como ponteiros, mas com funcionalidades adicionais como contagem de referências, liberação automática de memória ou validação de acesso.

```
class SmartPointer {
private:
    int* ptr;

public:
    // Construtor
    SmartPointer(int* p = nullptr) : ptr(p) {}

    // Destrutor
```

```
    ~SmartPointer() {
        delete ptr;
    }

    // Operador de desreferenciação *
    int& operator*() const {
        if (!ptr) {
            throw std::runtime_error("Null pointer dereference");
        }
        return *ptr;
    }

    // Operador de acesso a membro ->
    int* operator->() const {
        if (!ptr) {
            throw std::runtime_error("Null pointer dereference");
        }
        return ptr;
    }

    // Conversão para bool (verificação de nulidade)
    explicit operator bool() const {
        return ptr != nullptr;
    }
};

// Uso:
SmartPointer sp(new int(42));
if (sp) {                       // Verifica se não é nulo
    std::cout << *sp << std::endl;  // Acessa o valor
    *sp = 100;                  // Modifica o valor
}
```

A sobrecarga destes operadores especiais permite criar tipos personalizados que se integram perfeitamente com o resto da linguagem C++, oferecendo interfaces intuitivas e eficientes para manipulação de dados e comportamentos complexos.

5. Conceitos Relacionados

5.1 A Keyword `friend`

Permite que funções ou classes acessem membros privados.

```
class Point {
private:
    double x, y;

public:
    Point(double x_val = 0, double y_val = 0) : x(x_val), y(y_val) {}

    // Declarando função friend para cálculo de distância
```

```
    friend double distance(const Point& p1, const Point& p2);

    // Declarando operador de stream como friend
    friend std::ostream& operator<<(std::ostream& os, const Point& p);
};

// Função friend com acesso direto a membros privados
double distance(const Point& p1, const Point& p2) {
    double dx = p1.x - p2.x;  // Acesso direto a membros privados
    double dy = p1.y - p2.y;
    return std::sqrt(dx*dx + dy*dy);
}

std::ostream& operator<<(std::ostream& os, const Point& p) {
    return os << "(" << p.x << ", " << p.y << ")";
}
```

5.2 Const-correctness

Uso apropriado do qualificador `const` para garantir semântica correta.

```
class Array {
private:
    int* data;
    size_t size;

public:
    Array(size_t s) : size(s) {
        data = new int[size]();
    }

    ~Array() { delete[] data; }

    // Operador de indexação (não-const)
    int& operator[](size_t index) {
        return data[index];
    }

    // Operador de indexação (const)
    const int& operator[](size_t index) const {
        return data[index];
    }
};
```

5.3 Move Semantics (C++11)

Transfere recursos entre objetos em vez de copiá-los.

```cpp
class String {
private:
    char* data;
    size_t length;

public:
    // Construtor de movimento
    String(String&& other) noexcept : data(other.data), length(other.length) {
        other.data = nullptr;
        other.length = 0;
    }

    // Operador de atribuição de movimento
    String& operator=(String&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            length = other.length;
            other.data = nullptr;
            other.length = 0;
        }
        return *this;
    }
};
```

6. Novidades em C++17/20/23

6.1 O Operador Spaceship (<=>)

O operador spaceship (<=>) foi introduzido no C++20 e oferece uma maneira de implementar todos os operadores de comparação (==, !=, <, >, <=, >=) com uma única definição. Ele retorna um dos seguintes tipos:

- `std::strong_ordering`: Ordenação total (menor, igual, maior)
- `std::weak_ordering`: Ordenação total, mas pode haver equivalência sem igualdade
- `std::partial_ordering`: Ordenação parcial (alguns valores podem ser incomparáveis)

Exemplo Básico com `= default`

```cpp
#include <compare>

class Point {
public:
    int x, y;

    // Implementação automática para todos operadores de comparação
    auto operator<=>(const Point& other) const = default;
};
```

```
// Uso:
Point p1{1, 2};
Point p2{3, 4};
bool result = p1 < p2;  // true (compara lexicograficamente)
```

Implementação Personalizada

```
#include <compare>

class Rectangle {
private:
    int width, height;

public:
    Rectangle(int w, int h) : width(w), height(h) {}

    // Comparação baseada na área
    std::strong_ordering operator<=>(const Rectangle& other) const {
        int area1 = width * height;
        int area2 = other.width * other.height;
        return area1 <=> area2;
    }

    // Ainda precisamos definir == para igualdade baseada em dimensões
    bool operator==(const Rectangle& other) const {
        return width == other.width && height == other.height;
    }
};
```

Exemplo com `std::partial_ordering` (Valores Incomparáveis)

```
#include <compare>
#include <cmath>

class FloatingPoint {
private:
    double value;

public:
    FloatingPoint(double v) : value(v) {}

    // Comparação que trata NaN como incomparável
    std::partial_ordering operator<=>(const FloatingPoint& other) const {
        if (std::isnan(value) || std::isnan(other.value)) {
            return std::partial_ordering::unordered;  // Incomparável
        }
        return value <=> other.value;  // Comparação normal
    }
```

```
    // Igualdade específica que trata NaN como não igual a qualquer coisa
    bool operator==(const FloatingPoint& other) const {
        return !std::isnan(value) && !std::isnan(other.value) && value ==
other.value;
    }
};
```

6.2 Conceitos e Restrições (C++20)

Permitem especificar requisitos em templates.

```
#include <concepts>

// Definindo um conceito para tipos que suportam adição
template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

// Função template que usa o conceito
template<Addable T>
T sum(const T& a, const T& b) {
    return a + b;
}

// Sobrecarga de operador restrita por conceito
template<Addable T>
T operator+(const std::vector<T>& lhs, const std::vector<T>& rhs) {
    if (lhs.size() != rhs.size()) {
        throw std::invalid_argument("Vector sizes don't match");
    }

    std::vector<T> result(lhs.size());
    for (size_t i = 0; i < lhs.size(); ++i) {
        result[i] = lhs[i] + rhs[i];
    }
    return result;
}
```

6.3 Designated Initializers (C++20)

Inicializa membros por nome, útil com operadores sobrecarregados.

```
struct Point {
    double x, y;

    // Operadores sobrecarregados
    Point operator+(const Point& other) const {
```

```
        return {.x = x + other.x, .y = y + other.y};  // Designated initializer
    }
};
```

6.4 Melhorias em constraints (C++23)

Refinamento de restrições de conceitos para operadores sobrecarregados.

```
// C++23 oferece syntax mais simples para constraints
template<typename T>
    requires requires(T a, T b) { a + b; }  // Requer que a + b seja válido
T add(T a, T b) {
    return a + b;
}
```

7. Boas Práticas

7.1 Respeitar Semântica Intuitiva

- O operador `+` deve ser comutativo quando faz sentido
- O operador `==` deve ser reflexivo, simétrico e transitivo
- Operadores como `+=` devem retornar referência para permitir encadeamento

7.2 Implementações Consistentes

Para operadores relacionados, garanta consistência:

- Implemente `!=` em termos de `==`: `a != b` deve ser o mesmo que `!(a == b)`
- Implemente `>` em termos de `<`: `a > b` deve ser o mesmo que `b < a`
- Implemente `+` em termos de `+=` para evitar duplicação de código

7.3 Conversões Explícitas vs. Implícitas

- Use `explicit` para conversões que podem causar perda de informação ou serem contraintuitivas
- Prefira conversões explícitas para evitar comportamentos inesperados

```
class Percentage {
private:
    double value;  // 0-100

public:
    Percentage(double v) : value(v) {}

    // Conversão segura para double
    operator double() const {
        return value;
    }
```

```
    // Conversão potencialmente perigosa para int
    explicit operator int() const {
        return static_cast<int>(value);
    }
};
```

7.4 Exemplo de Código Bem Estruturado

Vamos criar uma classe `Money` completa que utiliza boas práticas:

```
#include <iostream>
#include <string>
#include <cmath>
#include <compare>  // Para <=> (C++20)

class Money {
private:
    long cents;        // Armazenado em centavos para evitar erros de ponto
flutuante
    std::string currency;

    static long roundToCents(double amount) {
        return static_cast<long>(std::round(amount * 100));
    }

public:
    // Construtores
    Money(long c = 0, std::string curr = "USD") : cents(c),
currency(std::move(curr)) {}
    Money(double amount, std::string curr = "USD") : cents(roundToCents(amount)),
currency(std::move(curr)) {}

    // Operadores aritméticos
    Money operator+(const Money& other) const {
        if (currency != other.currency) {
            throw std::invalid_argument("Cannot add different currencies");
        }
        return Money(cents + other.cents, currency);
    }

    Money& operator+=(const Money& other) {
        if (currency != other.currency) {
            throw std::invalid_argument("Cannot add different currencies");
        }
        cents += other.cents;
        return *this;
    }

    Money operator-() const {
        return Money(-cents, currency);
    }
```

```
    Money operator-(const Money& other) const {
        if (currency != other.currency) {
            throw std::invalid_argument("Cannot subtract different currencies");
        }
        return Money(cents - other.cents, currency);
    }

    Money& operator-=(const Money& other) {
        if (currency != other.currency) {
            throw std::invalid_argument("Cannot subtract different currencies");
        }
        cents -= other.cents;
        return *this;
    }

    Money operator*(double factor) const {
        return Money(roundToCents(cents * factor / 100.0), currency);
    }

    Money& operator*=(double factor) {
        cents = roundToCents(cents * factor / 100.0);
        return *this;
    }

    // Operador spaceship (C++20)
    auto operator<=>(const Money& other) const {
        if (currency != other.currency) {
            throw std::invalid_argument("Cannot compare different currencies");
        }
        return cents <=> other.cents;
    }

    bool operator==(const Money& other) const {
        return currency == other.currency && cents == other.cents;
    }

    // Conversões
    explicit operator double() const {
        return cents / 100.0;
    }

    // Acessores
    double getAmount() const { return cents / 100.0; }
    const std::string& getCurrency() const { return currency; }

    // Exibição formatada
    friend std::ostream& operator<<(std::ostream& os, const Money& m) {
        os << m.currency << " " << std::abs(m.cents) / 100;
        if (std::abs(m.cents) % 100 != 0) {
            os << "." << (std::abs(m.cents) % 100) / 10 << (std::abs(m.cents) %
10);
        } else {
            os << ".00";
```

```
        }
        return os;
    }

    friend Money operator*(double factor, const Money& money) {
        return money * factor;
    }
};

// Exemplo de uso
void demonstrateMoney() {
    Money salary(3000.50, "USD");
    Money bonus(500.75, "USD");

    Money total = salary + bonus;
    std::cout << "Total: " << total << std::endl;

    Money tax = total * 0.15;
    std::cout << "Tax (15%): " << tax << std::endl;

    Money netIncome = total - tax;
    std::cout << "Net income: " << netIncome << std::endl;

    if (netIncome > Money(3000, "USD")) {
        std::cout << "Net income exceeds $3000" << std::endl;
    }
}
```

Com estas diretrizes e exemplos, você tem um guia prático para implementar sobrecarga de operadores de forma eficiente e correta em C++ moderno.

Apêndice: Tabela de Referência de Operadores

Operador	Descrição	Pode ser sobrecarregado como Membro	Pode ser sobrecarregado como Friend
+ - * / %	Aritméticos	Sim	Sim
== != < > <= >=	Comparação	Sim	Sim
&& || !	Lógicos	Sim	Sim
& | ^ ~ << >>	Bit a bit	Sim	Sim
= += -= *= /= %= &= |= ^= <<= >>=	Atribuição	Sim, obrigatório para =	Não
++ --	Incremento/decremento	Sim	Sim
()	Chamada de função	Sim, obrigatório	Não
[]	Acesso a elementos	Sim, obrigatório	Não

Operador	Descrição	Pode ser sobrecarregado como Membro	Pode ser sobrecarregado como Friend
->	Acesso a membro via ponteiro	Sim, obrigatório	Não
*	Desreferenciação	Sim	Sim
new delete	Alocação/desalocação	Sim	Sim
,	Vírgula	Sim	Sim
<=>	Spaceship (C++20)	Sim	Sim

Considerações Finais sobre o Operador Spaceship (<=>)

O operador spaceship (<=>), introduzido no C++20, simplifica significativamente a implementação de comparações. Com ele, podemos reduzir a quantidade de código necessário para implementar todos os seis operadores de comparação (==, !=, <, >, <=, >=).

Tipos de Ordenação Retornados por <=>

1. **std::strong_ordering**:
    - Indica ordenação total e determinística
    - Valores possíveis: less, equal, greater
    - Usado para tipos como int, long
2. **std::weak_ordering**:
    - Indica ordenação total, mas permite equivalência sem igualdade
    - Valores possíveis: less, equivalent, greater
    - Útil para tipos como strings case-insensitive
3. **std::partial_ordering**:
    - Permite valores incomparáveis (como NaN em ponto flutuante)
    - Valores possíveis: less, equivalent, greater, unordered
    - Usado para tipos como double

Exemplo com os Três Tipos de Ordenação

```
#include <iostream>
#include <compare>
#include <string>
#include <cmath>

// Utilizando strong_ordering (ordenação total)
class Version {
private:
    int major, minor, patch;
```

```
public:
    Version(int maj = 0, int min = 0, int pat = 0)
        : major(maj), minor(min), patch(pat) {}

    // Comparação lexicográfica (versão a versão)
    std::strong_ordering operator<=>(const Version& other) const {
        if (auto cmp = major <=> other.major; cmp != 0) {
            return cmp;
        }
        if (auto cmp = minor <=> other.minor; cmp != 0) {
            return cmp;
        }
        return patch <=> other.patch;
    }

    // Ainda precisamos definir ==
    bool operator==(const Version& other) const = default;
};

// Utilizando weak_ordering (equivalência sem igualdade)
class CaseInsensitiveString {
private:
    std::string text;

    // Converte para lowercase
    static std::string toLower(const std::string& s) {
        std::string result = s;
        for (char& c : result) {
            c = std::tolower(c);
        }
        return result;
    }

public:
    CaseInsensitiveString(const std::string& s) : text(s) {}

    // Comparação case-insensitive
    std::weak_ordering operator<=>(const CaseInsensitiveString& other) const {
        return toLower(text) <=> toLower(other.text);
    }

    // Igualdade case-sensitive
    bool operator==(const CaseInsensitiveString& other) const {
        return text == other.text;
    }
};

// Utilizando partial_ordering (valores incomparáveis)
class FloatingPoint {
private:
    double value;

public:
```

```
    FloatingPoint(double v) : value(v) {}

    // Comparação com tratamento de NaN
    std::partial_ordering operator<=>(const FloatingPoint& other) const {
        if (std::isnan(value) || std::isnan(other.value)) {
            return std::partial_ordering::unordered;
        }
        return value <=> other.value;
    }

    bool operator==(const FloatingPoint& other) const {
        return !std::isnan(value) && !std::isnan(other.value) && value ==
other.value;
    }
};

int main() {
    // Teste com strong_ordering
    Version v1(2, 1, 0);
    Version v2(2, 0, 5);

    if (v1 > v2) {
        std::cout << "v1 é maior que v2" << std::endl;
    }

    // Teste com weak_ordering
    CaseInsensitiveString s1("Hello");
    CaseInsensitiveString s2("hello");

    std::cout << "s1 == s2: " << (s1 == s2) << std::endl;                  //
false (case-sensitive)
    std::cout << "s1 é equivalente a s2: " << (s1 <=> s2 == 0) << std::endl; //
true (case-insensitive)

    // Teste com partial_ordering
    FloatingPoint f1(1.0);
    FloatingPoint f2(std::numeric_limits<double>::quiet_NaN());

    if ((f1 <=> f2) == std::partial_ordering::unordered) {
        std::cout << "f1 e f2 são incomparáveis (um deles é NaN)" << std::endl;
    }

    return 0;
}
```

Estas técnicas modernas tornam a sobrecarga de operadores em C++ mais expressiva, segura e com menos código redundante.