Aula 05:

Sobrecarga de Operadores
Parte I

ECOP03 - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Rev. Abril 23

Funções e classes *friend*

Uma função ***friend*** (amiga) de uma classe é definida fora do escopo dessa classe, mas tem o direito de acessar todos os seus membros: *public* e *não-public* (private e protected).

> Funções independentes, classes inteiras ou funções membro de classes podem ser declaradas como amigas de outra classe.

O motivo pelo qual isto é permitido está diretamente relacionado com a eficiência, pois a “amizade” permite o acesso direto aos campos privados de um objeto e por isso não depende de métodos de acesso.

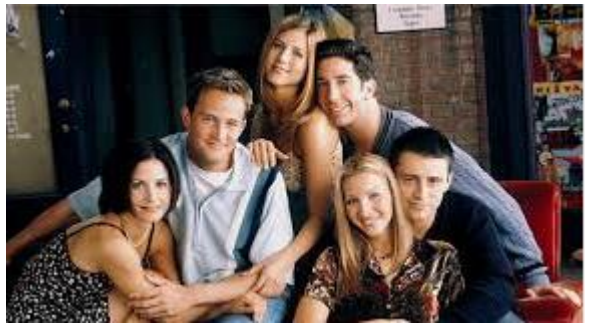
```cpp
#include <iostream>
using namespace std;

class Box
{
  private:
    double width;
  public:
    friend void print_width(Box box);
    void set_width(double wid);
};

// Definição de uma função membro
void Box::set_width(double wid)
{
    width = wid;
}

// Veja: print_width() não é uma função membro da classe
void print_width(Box box)
{
  // Uma vez que print_wodth é uma amiga (friend) de box, ela
  // pode acessar qualquer membro da classe diretamente
   cout << "Width of box : " << box.width <<endl;
}

// Função principal
int main( )
{
   Box box;
   box.set_width(10.0); // utiliza função membro=
   print_width( box ); // imprime com função friend
   return 0;
}
```



Esta violação limitada da ocultação de dados pode ser muito útil se usada de maneira disciplinada.

É claro que a função ou classe que desfruta do relacionamento de amizade precisa estar intimamente ligada à classe ao qual se concedeu esta amizade.

Funções e classes *friend*

Além da criação de funções independentes, também é possível criar funções membro de classes que são *friends* de outras. Veja um exemplo:

```
// declaração prévia de classe List é necessária
class List;

class List_iterator {
  // ...
  public:
    int* next(List&); // precisa ser public, para que List tenha acesso
};

class List {
  // concede acesso aos membros de List para next()
  friend int* List_iterator::next(List&);
  //...
};
```

E ainda, é possível fazer uma classe inteira amiga de outra:

```
class List {
  // concede acesso dos membros de List para
  // toda a classe List_iterator
  friend class List_iterator;
};
```

Funções e classes *friend*

Importante: Lembre-se sempre de que é necessário expressar a relação de amizade apenas no lado que concederá acesso. A relação **não** implica em acesso mútuo liberado.

Dicas:

- Utilize funções ***friend*** quando for necessário que uma **função não-membro tenha acesso à representação da classe** (por exemplo, para melhorar a notação ou acessar a representação de duas classes).
- **Prefira sempre funções membro** a funções ***friend*** para garantir acesso à implementação da classe.
- Um dos principais usos de funções ***friend*** em C++ é na **Sobrecarga de Operadores**.

Sobrecarga de Operadores

Cada campo da ciência desenvolveu uma notação padrão própria para tornar mais conveniente a apresentação e discussão de problemas que envolvessem conceitos utilizados com frequência.

Por exemplo, temos que, devido à sua ampla aceitação,

x + y * z

é uma representação muito mais clara do que

“Multiplique y e z e depois adicione ao resultado o valor de x”.

Não podemos nunca desprezar a importância de uma **notação concisa** para acrescentar **clareza** e **agilidade** às operações comuns.

Sobrecarga de Operadores

C++ suporta um conjunto de operadores (+, -, *, &, =, etc.) para tipos nativos. No entanto, a maioria dos conceitos para os quais os operadores são convencionalmente utilizados são tipos não-nativos do C++, e precisam ser representados como classes.

> Para lidar com aritmética de números complexos, álgebra matricial, sinais lógicos e strings de caracteres, quase sempre utilizamos classes que representem essas noções.

Definir operadores para estas classes permite ao programador prover uma notação mais convencional e conveniente para a manipulação de objetos do que poderia ser realizado apenas com a notação funcional básica (chamada de funções membro).

Veja o que isso significa:

```
class Complexo
{
    private:
        double re, im;
    public:
        Complexo(double=0, double=0);
        ~Complexo() {}
        double get_real() {return re;}
        double get_imag() {return im;}

        // definindo operadores
        Complexo& adicionar(Complexo&); // +
        Complexo& subtrair(Complexo&); // -
        void print(); // <<
};
```

```
int main()
{
    Complexo a{10,0};
    Complexo b{10, 15};
    a.adicionar(b); // a = a+b;
    b.subtrair(a); // b = b-a;
    a.print(); cout << "\n";
    b.print();
}
```

As funções de adição, subtração e impressão cumprem seu papel, mas sua utilização segue uma notação pouco convencional para o tratamento de números complexos (chamadas de função). Seria muito melhor que aproximássemos a maneira de realizar operações do que estamos acostumados a realizar na prática.

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

class Complexo
{
    private:
        double re, im;
    public:
        Complexo(double=0, double=0);
        ~Complexo() {}
        double get_real() {return re;}
        double get_imag() {return im;}

        // definindo operadores
        Complexo operator+(Complexo&); // +
        Complexo operator-(Complexo&); // -
        friend ostream& operator<<(ostream&, const Complexo&); // <<
};
```

Se declararmos funções membro “sobrecarregando” operadores, podemos utilizar uma notação muito mais simples quando formos usar nossos objetos:

```
int main()
{
    Complexo a{10,0};
    Complexo b{10, 15};
    a = a + b; // notação direta e simples
    b = b - a; // como em representação aritmética comum
    cout << a << "\n" << b; // e impressão natural
}
```

Veja as diferenças entre as duas implementações:

```
// Operadores com sintaxe funcional básica
// soma
Complexo& Complexo::adicionar(Complexo& c)
{
    re += c.get_real();
    im += c.get_imag();
    return *this;
}

// subtração
Complexo& Complexo::subtrair(Complexo& c)
{
    re -= c.get_real();
    im -= c.get_imag();
    return *this;
}

void Complexo::print()
{
    cout << re << " + (" << im << ")i";
}
```

Implementação com
Notação Funcional Básica

```
// Sobrecarga de operadores
// soma
Complexo Complexo::operator+(Complexo& c)
{
    double r, i;
    r = re + c.get_real();
    i = im + c.get_imag();
    return Complexo{r, i};
}

// subtração
Complexo Complexo::operator-(Complexo& c)
{
    double r, i;
    r = re - c.get_real();
    i = im - c.get_imag();
    return Complexo{r, i};
}

// impressão
ostream& operator<<(ostream& output, const Complexo& c)
{
    output << c.re << " + (" << c.im << ")i";
    return output;
}
```

Implementação com
Sobrecarga de Operadores

Repare como nossas classes estão ficando cada vez mais parecidas com os tipos nativos. C++ nos provê com uma série de ferramentas para a criação de tipos definidos pelo usuário, de maneira que, ao final de sua criação, quase não seja possível distinguir um do outro olhando apenas para sua utilização.

> *"Those types are not 'abstract'; they are as real as int and float."* (Doug McIlroy)
>
> *"A diferença entre a construção e a criação é exatamente esta: uma coisa construída só pode ser amada depois de construída; mas uma coisa criada é amada antes mesmo de existir."* (G. K. Chesterton)

Muitos dos usos mais óbvios da sobrecarga de operadores são para tipos numéricos, no entanto, não há restrição neste sentido.

Sobrecarga de Operadores

Funções para os seguintes operadores podem ser declaradas:

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

O nome de uma função de operador é sempre formada pela palavra "**operator**" seguida pelo próprio operador, como por exemplo em *operator <<.*

Não é possível criar novos operadores;

Para este tipo de ação, utilize a notação funcional básica.

Sobrecarga de Operadores

Uma função operator é declarada normalmente e pode ser chamada como qualquer outra função.

A utilização do próprio operador funciona como uma abreviação para uma chamada explícita da função operador. Veja:

```
// as duas notações funcionam
a = a + b; // notação direta e simples
a = a.operator+(b); // notação explícita funcional
```

Apenas alguns operadores não podem ser sobrecarregados:

.	.*	::	?:

Sobrecarga de Operadores

Restrições:

- A ***precedência*** de um operador não pode ser alterada pela sobrecarga. Entretanto, os parênteses podem ser utilizados para forçar a ordem de avaliação dos operadores sobrecarregados de uma função caso isso seja necessário.
- A ***associatividade*** de um operador não pode ser modificada pela sobrecarga (se o operador é aplicado da esquerda pra direita ou inverso).

Sobrecarga de Operadores

Restrições:

- Não é possível mudar a "aridade" do operador: operadores unários permanecem unários e binários continuam sendo operadores binários.
- Não é possível sobrecarregar operadores para tipos nativos, mudando a forma, por exemplo, como é feita a soma de int's.
- **Cuidado**: A sobrecarga de **+** não sobrecarrega operadores relacionados, como **+=**. O mesmo com **==** e **!=**. É preciso sobrecarregar de cada um deles separadamente.

Sobrecarga de Operadores

Funções operator podem ser funções membro não-*static* ou funções independentes (ou globais), que normalmente são tornadas *friend* para motivos de notação simplificada e desempenho.

- **Funções membro** utilizam o ponteiro ***this*** implicitamente para obter um de seus argumentos de objeto da classe (o operando da esquerda para operadores binários).
- Em uma **função independente**, os argumentos para os dois operandos de um operador binário devem ser listados explicitamente.

Cuidado: Ao sobrecarregar `()`, `[]`, `->` ou qualquer outro dos operadores de **atribuição**, a função de sobrecarga deve ser declarada como um **membro da classe**. Nos outros casos, a sobrecarga pode ser feita como membro ou função independente.

Sobrecarga de Operadores

Independente da maneira como será realizada a implementação da sobrecarga, o operador será utilizado da mesma maneira nas expressões. Mas qual a melhor?

> Quando uma função é implementada como membro da classe, o operando mais a esquerda (ou único, se unário) deve ser, obrigatoriamente, um objeto (ou referência de um objeto) da classe do operador.

Se o objeto mais a esquerda da operação puder ser de outra classe ou mesmo de um tipo fundamental, a função deverá ser implementada como global (*friend* da classe).

Veja um exemplo:

```
class Point {

    private:
        int x, y;
    public:
        Point(int=0, int=0);
        ~Point() {}
        int get_x() {return x;}
        int get_y() {return y;}

        // primeiro operando é do tipo Point
        Point operator+(Point&);
        Point operator+(int);

        // Primeiro operando é do tipo int
        friend Point operator+(int, Point&);
};
```

Repare que é necessário pensar em todas as possibilidades, especialmente caso seja possível utilizar valores de outros tipos nas operações.

A ordem dos operadores **importa**, e muito.

```
Point::Point(int xx, int yy) : x{xx}, y{yy} {}

// operações do tipo Point + Point
Point Point::operator+(Point& p)
{
    int xx = x + p.x;
    int yy = y + p.y;
    return Point{xx, yy};
}

// operações do tipo Point + int
Point Point::operator+(int value)
{
    int xx = x + value;
    int yy = y + value;
    return Point{xx, yy};
}

// Função global: operações do tipo int + Point
Point operator+(int value, Point& p)
{
    int xx = p.x + value;
    int yy = p.y + value;
    return Point{xx, yy};
}
```

Veja: é possível acessar os membros privado x e y diretamente (p.x e p.y) mesmo quando são de outro objeto, mas somente dentro da classe Point.

Retornamos um novo objeto ainda sem nome, Point{xx, yy}, que será atribuído ao final da chamada da operação.

```
int main()
{
    Point p1{20, 20};
    Point p2 = p1 + 10; // chama Point::operator+(int value)

    cout << "p1: (" << p1.get_x() << ", " << p1.get_y() << ")\n";
    cout << "p2: (" << p2.get_x() << ", " << p2.get_y() << ")\n";

    Point p3 = p1 + p2; // chama Point::operator+(Point& p)
    cout << "p3: (" << p3.get_x() << ", " << p3.get_y() << ")\n";

    Point p4 = 10 + p1; // chama operator+(int value, Point& p)
    cout << "p4: (" << p4.get_x() << ", " << p4.get_y() << ")\n";
}
```

Na função main() realizamos as operações de adição normalmente, como se houvesse uma só implementação de operator+.

> Note que o significado de qualquer dos operadores sobrecarregados pode ser mudado completamente, de acordo com a vontade do programador.

Para maior consistência do uso, é recomendado seguir o modelo dos tipos nativos quando definirmos os operadores sobrecarregados.

> Se a semântica de um operador sobrecarregado difere substancialmente de seu significado em outros contextos, pode ser mais confuso que útil utilizá-los.

Mantenha sua implementação consistente com o que a intuição diz sobre um operador. **Não invente**!

Operadores de Inserção e Extração de Stream

A biblioteca I/O *stream* provê operações de entrada e saída para texto e tipos numéricos. Suas definições se encontram principalmente em <istream> e <ostream>.

> Um objeto do tipo **ostream** converte objetos de um determinado tipo em um fluxo de caracteres (ou bytes), enquanto um objeto do tipo **istream** converte um fluxo de caracteres (bytes) em objetos de um determinado tipo.

Os fluxos (ou *streams*) padrão que utilizaremos são:

- cout (<<): saída padrão (geralmente para a tela)
- cin (>>): entrada padrão (geralmente através de um teclado)

Operadores de Inserção e Extração de Stream

Podemos enviar ou receber dados de qualquer tipo fundamental utilizando o operador de extração de stream (>>) e o operador de inserção de stream (<<).

As bibliotecas de C++ já sobrecarregam esses operadores para processar cada um dos tipos fundamentais, incluindo até mesmo ponteiros e strings char* no estilo C.

Também podemos sobrecarregar esses operadores para realizar a entrada e saída para nossos próprios tipos. Vejamos um exemplo:

```cpp
#ifndef POINT_H
#define POINT_H

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

class Point
{
    private:
        int x, y;
    public:
        Point(int xx = 0, int yy = 0) : x{xx}, y{yy} {}
        ~Point() {}

        friend ostream& operator<<(ostream&, const Point&);
        friend istream& operator>>(istream&, Point&);
};

#endif
```

```
Ponto 1: (10, 10)
Entre com os valores de p2 (x e y): 10 50
Ponto 2: (10, 50)
```

```cpp
#include <iostream>
#include "point.h"
using namespace std;

ostream& operator<<(ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

istream& operator>>(istream& in, Point& p) {
    in >> p.x >> p.y;
    return in;
}

int main() {
    Point p1{10,10};
    cout << "Ponto 1: " << p1 << "\n";

    Point p2;
    cout << "Entre com os valores de p2 (x e y): ";
    cin >> p2;
    cout << "Ponto 2: " << p2 << "\n";
}
```

Operadores de Inserção e Extração de Stream

Pontos importantes na implementação. Vamos destacá-los:

- A sobrecarga dos operadores de entrada e saída deve ser realizada através de funções globais ***friend***. Isso se deve ao fato de que o primeiro operando (ou seja, o da esquerda) é sempre o objeto que manipula o stream (cin, cout). Veja os exemplos:

```
cin >> p1;          // repare que tanto cin quanto cout
cin >> p2 >> p3;    // estão sempre do lado esquerdo
cout << p3;         // da operação
```

Além disso,

- É importante que as funções de sobrecarga recebam como parâmetros referências para **istream&**, **ostream&** e para próprio objeto (Point&). Isso é ainda mais importante no caso do istream. Caso seja passado o objeto por valor, as mudanças serão realizadas em uma cópia do objeto. Portanto, utilize referências.

- Também devemos retornar uma referência para o próprio objeto manipulador do stream ao final da função. Isso possibilita que sejam feitas chamadas encadeadas do operador, como vemos abaixo:

```
cout << “Dois pontos são ” << p1 << p2;
cin >> p3 >> p4;
```

Ao entrar na função sobrecarregada, os nomes dos parâmetros tornam-se “apelidos”, ou *alias*, para os objetos passados para a função. No caso do exemplo, temos:

```
istream& operator>>(istream& in, Point& p) {
    in >> p.x >> p.y;
    return in;
}
```

E na chamada da operação:

```
cin >> p2;
```

Portanto, dentro da função, *cin* passa a ser chamado de *in* (ou qualquer outro nome desejado) e *p2* passa a ser referenciado pelo nome *p*. Como são referências, estes novos nomes identificam o mesmo objeto, na mesma posição de memória.

Referências

- https://cplusplus.com/reference/

- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.