Friends e Sobrecarga

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP
Baseado nos slides do Prof. Marco Antônio Carvalho

Funções Amigas I

- Uma função amiga de uma classe é uma função definida completamente fora da classe
  - Porém, possui acesso aos membros públicos e não públicos de uma classe;
  - Funções isoladas ou mesmo classes inteiras podem ser declaradas como amigas de outra classe.

Funções Amigas II

- Funções amigas podem melhorar a performance de uma aplicação
- Também são utilizadas na sobrecarga de operadores e na criação de iteradores.

- Podemos também declarar uma classe amiga
  - Todos os métodos terão acesso aos membros da outra classe.

Funções Amigas III

- Para declararmos uma função amiga, utilizamos a palavra **friend** antes do protótipo da função dentro da classe
  - Note que a função não será um método.
- Se uma classe *ClasseUm* será declarada como amiga de uma classe *ClasseDois*, dentro de *ClasseUm* declaramos

```
friend class ClasseDois;
```

RelogioFriend.h I

```
#ifndef RELOGIO_H
#define RELOGIO_H
class Relogio
{
    int h, m, s;
public:
    Relogio(int=0, int=0, int=0);
    ~Relogio();
    void setHora(int, int, int);
    void printHora();
    friend void alteraHMS(Relogio&);
};
#endif
```

RelogioFriend.cpp I

```
#include <iostream>
#include "RelogioFriend.h"

Relogio::Relogio(int h, int m, int s) :h(h), m(m), s(s) {}
Relogio::~Relogio(){}

void Relogio::setHora(int h, int m, int s){
        this->h = h;     this->m = m;
        this->s = s;
}
void Relogio::printHora(){
        std::cout << h << ":" << m << ":"
                  << s << std::endl;
}
void alteraHMS(Relogio& r){
        r.h = 10; r.m = 5; r.s = 0;
}
```

Funções Amigas I

- Note que o objeto deve ser enviado como parâmetro para a função
  - Uma vez que o objeto não chama a função, é necessário indicar qual é o objeto cujos atributos serão alterados;
  - Para que a alteração tenha efeito, precisamos enviar o objeto por referência (**&**)
- Qualquer outra tentativa de acesso a membros privados por funções que não são amigas de uma classe resultará em erro de compilação.

Main.cpp I

```
#include  <iostream>
#include "RelogioFriend.h"

int main()
{
    Relogio R;
    R.setHora(12, 34, 27);
    R.printHora();
    alteraHMS(R);
    R.printHora();
    return 0;
}
```

Classes amigas (*friend*) I

- Programar usando orientação objeto significa encapsular num mesmo módulo:
  - dados,
  - informações e
  - operações que de alguma forma se relacionam.
- As vezes é preciso quebrar esse encapsulamento.
- Para isso C++ oferece o operador *friend*.

Classes amigas (*friend*) II

- Com *friend* podemos implementar associações mais íntimas entre as classes.
- Por exemplo, se duas classes X e Y têm uma associação do tipo Y acessa X, então os métodos a serem acessados por Y devem ser públicos em X.
- Problema: os métodos públicos de X ficam públicos para qualquer um que usar um objeto da classe X (quebrando o encapsulamento de X)

Classes amigas (*friend*) III

- Podemos manter o encapsulamento de X declarando a Y como amiga
- Com *friend* preservamos o encapsulamento.
- Liberamos o acesso aos membros da classe para classes/métodos/funções declaradas como amigas.

Classes amigas (*friend*) IV

- Serve para dizer para uma classe quais são sua (s) classe (s) amiga (s).
- As classes amigas podem acessar os atributos e métodos *private*.
- As classes *friends* não são herdáveis nem transitivas (Eu posso declarar que sou seu amigo, mas isso não significa que eu, necessariamente, acredite que seus filhos, ou seus outros amigos, sejam meus amigos.).

Classes amigas (*friend*) V

- Vamos permitir que uma função externa ganhe acesso aos membros *protected* e *private* de uma classe.

```
class Retangulo
{
        double largura, comprimento;
public:
        Retangulo(double=0.0, double=0.0);
        ~Retangulo();
        void setValores(double, double);
        double calculaArea();
        friend Retangulo duplicaAmiga(Retangulo);
};
```

Classes amigas (*friend*) VI

```
Retangulo::Retangulo(double largura,
    double comprimento) : largura(largura),
    comprimento(comprimento){}

Retangulo::~Retangulo(){}

void Retangulo::setValores(double largura,
    double comprimento){
        this->largura = largura;
        this->comprimento = comprimento;
}
```

Classes amigas (*friend*) VII

```
double Retangulo::calculaArea(){
        return largura * comprimento;
}

Retangulo duplicaAmiga(Retangulo R){
        Retangulo tmp;
        tmp.largura = 2 * R.largura;
        tmp.comprimento = 2 * R.comprimento;
        return tmp;
}
```

Classes amigas (*friend*) VIII

```
int main()
{
    Retangulo R1, R2;
    R1.setValores(2,3);
    R2 = duplicaAmiga(R1);
    cout << R2.calculaArea();

    return 0;
}
```

Mostraria na tela:
24

Classes amigas (*friend*) IX

- Utilizando a função `duplicaAmiga` conseguimos **acessar os membros** `largura` e `comprimento` dos objetos da classe `Retangulo`.
- Observe a que `duplicaAmiga` **não é um membro** da classe `Retangulo`.
- Assim como temos possibilidade de definir **funções amigas**, também podemos definir **classes amigas**

Classes amigas (*friend*) X

- Ao definir uma classe como amiga de outra permitimos que a segunda acesse os membros *protected* e *private* da primeira.
- Vejamos o seguinte exemplo: declarar uma classe quadrado e uma classe retângulo como amiga da primeira

Retangulo.h I

```
#ifndef RETANGULO_H
#define RETANGULO_H

class Quadrado;

class Retangulo
{
    int largura, comprimento;
public:
    Retangulo();
    ~Retangulo();
    int calculaArea();
    void converte(Quadrado);
};
#endif
```

Retangulo.cpp I

```
#include "Retangulo.h"
#include "Quadrado.h"

Retangulo::Retangulo(){}
Retangulo::~Retangulo(){}

int Retangulo::calculaArea(){
    return largura * comprimento;
}

void Retangulo::converte(Quadrado S){
    largura = S.lado;
    comprimento = S.lado;
}
```

Quadrado.h I

```
#ifndef QUADRADO_H
#define QUADRADO_H
class Retangulo;

class Quadrado
{
    int lado;
public:
    Quadrado();
    ~Quadrado();
    void setLado(int);
    friend class Retangulo;
};
#endif
```

Quadrado.h I

```
#include "Quadrado.cpp"

Quadrado::Quadrado() {}

Quadrado::~Quadrado() {}

void Quadrado::setLado(int lado){
    this->lado = lado;
}
```

Main.cpp I

```
#include <iostream>
#include "Retangulo.h"
#include "Quadrado.h"
using namespace std;


int main()
{
        Quadrado S;
        Retangulo R;
        S.setLado(4);
        R.converte(S);
        cout << R.calculaArea();

        return 0;
}
```

Mostraria na tela 16

Sobrecarga de Operadores I

- Como vimos até agora, operações relacionadas a objetos são realizadas através da chamada de métodos
- Porém, podemos utilizar os operadores da linguagem C++ para especificar operações comuns de manipulação de objetos;
- Novos operadores não podem ser criados;
- A adaptação dos operadores nativos da linguagem para nossas classes é chamada de **sobrecarga de operadores**.

Sobrecarga de Operadores II

- Por exemplo, os operadores ariméticos são sobrecarregados por padrão
  - Funcionam para quaisquer tipo de número (*int*, *float*, *double*, *long*, *etc.*);
  - Outro operador que é sobrecarregado é o de atribuição.
- Qualquer operação realizada por um operador sobrecarregado também pode ser realizada por um método sobrecarregado
  - No entanto, a notação de operador é muito mais simples e natural.

Sobrecarga de Operadores III

- A **principal convenção** a respeito da sobrecarga de operadores é a de que o **comportamento do operador** utilizado seja **análogo ao original**
  - Ou seja, $+$ só deve ser utilizado para realizar adição.

Sobrecarga de Operadores IV

- Para sobrecarregar um operador criamos um método ou função global cujo nome será **operator**, seguido pelo símbolo do operador a ser sobrecarregado
  - Por exemplo, **operator +()**;

Sobrecarga de Operadores V

- Os operadores '.' ,'.*', '?:' e '::' não podem ser sobrecarregados;
- Não é possível sobrecarregar um operador para lidar com tipos primitivos
  - Soma de inteiros não pode ser alterada.
- Não é possível alterar a precedência de um operador através de sobrecarga;

Sobrecarga de Operadores VI

- A continuação vejamos alguns operadores:

Operadores binários	+	-	+	/	%	^	&	|
	~	!	=	<	>	+=	-=	*=
	/=	%=	^=	&=	=—	<<	>>	>>=
	<<=	==	!=	<=	>=	&&	||	->*
	,	->	[]	()	new[]	delete[]	new	delete
Operadores unários	++	--	~	!				
Não podemos sobrecarregar	.	.*	::	?:				

Sobrecarga de Operadores VII

- O número de operandos de um operador não pode ser alterado através de sobrecarga;
- Sobrecarregar um operador não sobrecarrega os outros
  - Sobrecarregar $+$ não sobrecarrega $+=$;

Sobrecarga de Operadores VIII

- Veja a seguir o protótipo para sobrecarga de operador como função *friend* e método membro da classe.
- A declaração e a definição são iguais às de um método/função, apenas substituímos o nome do método/função pela palavra-chave **operator**

Sobrecarga de Operadores IX

```
class CNomeClasse
{
    Tipo atributo;
    // (1) Sobrecarga de operador unário como função friend
    friend CNomeClasse& operatorX (CNomeClasse& obj1);
    // (2) Sobrecarga de operador binário como função friend
    friend CNomeClasse& operatorX (CNomeClasse& obj1,
        CNomeClasse& obj2);
    // (3) Sobrecarga de operador unário como método membro
    CNomeClasse& operatorX();
    // (4) Sobrecarga de operador binário como método membro
    CNomeClasse& operatorX(CNomeClasse& obj2);
};
```

Ponto.h I

```
#ifndef PONTO_H
#define PONTO_H
class Ponto {
        int x, y;
public:
        Ponto(int=0, int=0); Ponto(const Ponto&);
        ~Ponto();
        void set(int, int); void set(const Ponto&);
        void print();
        // Declaração operadores unários como método membro
        Ponto& operator++();
        Ponto& operator++(int); // ++p
        // Declaração operadores binários como método membro
        Ponto operator+(const Ponto&) const;
        Ponto& operator=(const Ponto&);
    //Declaração operadores binários com função friend
        friend bool operator==(const Ponto&,
            const Ponto&);
};
#endif
```

Ponto.cpp I

```
#include <iostream>
#include "Ponto.h"

Ponto::Ponto(int x, int y) :x(x), y(y){}
Ponto::Ponto(const Ponto& P){
        x = P.x; y = P.y;
}

Ponto::~Ponto() {}

void Ponto::set(int x, int y){
        this->x = x;
        this->y = y;
}
void Ponto::set(const Ponto& P){
        this->x = P.x;
        this->y = P.y;
}
```

Ponto.cpp II

```
// Declaração operadores unários como método membro
Ponto& Ponto::operator++(){
        this->x++; this->y++; return *this;
}

Ponto& Ponto::operator++(int){ // ++p
        this->operator++(); return *this;
}

// Declaração operadores binários como método membro
Ponto Ponto::operator+(const Ponto& P2) const{
        Ponto tmp;
        tmp.x = this->x + P2.x;
        tmp.y = this->y + P2.y;
        return tmp;
}
```

Ponto.cpp III

```
Ponto& Ponto::operator=(const Ponto& P){
        if (this == &P)
                return *this;
        this->x = P.x; this->y = P.y;
        return *this;
}

//Declaração operadores binários com função friend
bool operator==(const Ponto& P1, const Ponto& P2){
        return (P1.x == P2.x && P1.y == P2.y);
}

void Ponto::print(){
    out << "(" << P.x << ","
        << P.y << ")\n";
}
```

Main.cpp I

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

int main()
{
        Ponto P(2, 3), P2(P), P3;
        P.print();
        P++; P.print();
        P3 = P + P2; P3.print();
        P3 = P;
        if (P3 == P)
                cout << "Iguais";
        else
                cout << "Diferentes";
        return 0;
}
```

Sobrecarga de entrada/saída de dados I

- Podemos, por exemplo, sobrecarregar o operador >> e << utilizado para leitura/escrita de dados
  - Note que o operando do lado esquerdo é um objeto da classe *istream/ostream*, da biblioteca padrão C++
  - Não podemos alterá-la para incluir a sobrecarga;
  - Mas podemos criar uma função que altere o operador do lado direito.

Sobrecarga de entrada/saída de dados II

- friend ostream& operator<<(ostream&, const Tipo&)
  - A sobrecarga é implementada através de uma função amiga
  - Retorna uma referência a um objeto *ostream* (fluxo de saída)
- friend istream& operator>>(ostream&, Tipo&)
  - A sobrecarga é implementada através de uma função amiga
  - Retorna uma referência a um objeto *istream* (fluxo de entrada)

Sobrecarga de entrada/saída de dados III

- O fato de ambas as funções retornarem um objeto, permite construções do tipo:

```
cin>>a;
cin>>a>>b>>c;
cout<<a;
cout<<a<<b<<c;
```

- Caso contrário não seria possível realizar chamadas "em cascata".

Ponto.h I

```
#ifndef PONTO_H
#define PONTO_H
class Ponto {
        int x, y;
public:
    . . .
    //Declaração operadores binários com função
    friend std::ostream& operator<<(std::ostream&,
    const Ponto&);

    friend std::istream& operator>>(std::istream&,
    Ponto&);
};
#endif
```

Ponto.cpp I

```
#include <iostream>
#include "Ponto.h"
     ...
std::ostream& operator<<(std::ostream& out,
     const Ponto& P){
     out << "(" << P.x << ","
              << P.y << ")\n";
     return out;
}

std::istream& operator>>(std::istream& in,
     Ponto& P){
          in >> P.x >> P.y;
          return in;
}
```

Main.cpp I

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

int main()
{
        Ponto P(2, 3), P2(P), P3;
        cout << "Digite coordenadas: ";
        cin >> P3;
        cout << P;
        cout << P++ << P + P2;
        return 0;
}
```

FIM