INF1010 - Estruturas de dados avançadas
Introdução a C++

Implementando iteradores

PUC-Rio

2 de maio de 2018

Roteiro

Funções e classes friend

- A princípio, membros **private** e **protected** de uma classe, não podem ser acessados de fora da classe em que foram declarados.
- O conceito de *friendship* é uma exceção a essa regra. Funções ou classes declaradas como **friend** podem acessar membros declarados como private e protected.

Funções friend

- Exemplo:

```
class Rectangle
{
private:
  int width, height;

public:
  Rectangle() {}

  Rectangle (int x, int y)
    : width(x), height(y) {}

  friend Rectangle duplicate (const Rectangle&);
};
```

Funções friend

- Exemplo:

```
Rectangle duplicate (const Rectangle& param)
{
  Rectangle ret;
  ret.width = param.width*2;
  ret.height = param.height*2;
  return ret;
}

int main ()
{
  Rectangle ret1(2,3);
  Rectangle ret2 = duplicate(ret1);
  return 0;
}
```

Funções friend

- *duplicate* **não** é uma função membro da classe!
  Apenas tem acesso aos seus membros private e protected sem pertencer a ela.
- *duplicate* poderia estar implementada em outra classe.
- Uma aplicação típica para funções friend é em operações realizadas entre duas classes, acessando métodos protegidos de ambas.

Classes friend

- Exemplo:

```
class Rectangle
{
  int width, height;
public:
  void convert( Square a );
};

class Square
{
  friend class Rectangle;

  private:
    int side;

  public:
    Square(int s) : side(s){}
}
```

Classes friend

- Exemplo:

```
void Rectangle::convert(Square a)
{
  width = a.side;
  height = a.side;
}
```

A classe Rectangle consegue acessar a variável *side* da classe Square, mesmo esta sendo privada, pois a classe Square definiu a classe Rectangle como friend.

Sobrecarga de operadores

Exemplo: Classe para representar pontos/vetores no espaço 3d.

```
class Vector
{
public:

    Vector() : x(0.0f), y(0.0f), z(0.0f) {}

    Vector( float x, float y, float z)
        : x(x), y(y), z(z) {}

    float x;
    float y;
    float z;
};
```

Sobrecarga de operadores

Executar operações entre vetores gera muito código, e está sujeito a erros (*error-prone*):

```
//Dois objetos da classe Vector
Vector u( 1.0f, 2.0f, 3.0f );
Vector v( 5.0f, 2.0f, 4.5f );

//Soma de dois vetores:
Vector w1;
w1.x = u.x + v.x;
w1.y = u.y + v.y;
w1.z = u.z + v.z;

//Produto vetorial:
Vector w2;
w2.x = u.y * v.z - v.y * u.z;
w2.y = u.z * v.x - v.z * u.x;
w2.z = u.x * v.y - v.x * u.y;
```

Sobrecarga de operadores

E se fosse possível fazer as mesmas operações da seguinte forma?

```
//Dois objetos da classe Vector
Vector u( 1.0f, 2.0f, 3.0f );
Vector v( 5.0f, 2.0f, 4.5f );

//Soma de dois vetores:
Vector w1 = u + v;

//Produto vetorial
Vector w2 = u % v;
```

É possível desde que se faça a **sobrecarga** (overloading) dos operadores binários + e % na classe Vector.

Sobrecarga de operadores

Sobrescrevendo o operador +:

```
Vector operator + (const Vector& b)
{
  return Vector( x + b.x, y + b.y, z + b.z );
}
```

Sobrescrevendo o operador %:

```
Vector operator % (const Vector& b)
{
  return Vector( y * b.z - b.y * z,
                 z * b.x - b.z * x,
                 x * b.y - b.x * y
  );
}
```

Sobrecarga de operadores

Mais um exemplo...

- E se quiséssemos imprimir um Vector utilizando std::cout?

```
Vector u(1.0f, 2.0f, 3.0f);
std::cout << "u: " << u << std::endl;

//Exemplo de saida:
//u: (1, 2, 3)
```

Sobrecarga de operadores

Sobrescrevendo o operador <<:

```
friend std::ostream& operator <<
    (std::ostream& output, const Vector& vector)
{
  output << "(" << vector.x << ", "
                 << vector.y << ", "
                 << vector.z <<
            ")";

  return output;
}
```

Sobrecarga de operadores

E assim o código abaixo:

```
Vector u(1.0f, 2.0f, 3.0f);
Vector v(5.0f, 2.0f, 4.5f);

Vector w1 = u + v;
Vector w2 = u % v;

std::cout << "u: " << u << std::endl;
std::cout << "v: " << v << std::endl;
std::cout << "w1: " << w1 << std::endl;
std::cout << "w2: " << w2 << std::endl;
```

Gera a saída:
u: (1, 2, 3)
v: (5, 2, 4.5)
w1: (6, 4, 7.5)
w2: (3, 10.5, -8)

Sobrecarga de operadores

- Todos os seguintes 38 operadores podem ser sobrecarregados:

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

- Operadores que **não** podem ser sobrecarregados:

::	.	.*	?:

- **Novos** operadores **não** podem ser criados, tais como: **, <>, ou &|

Sobrecarga de operadores

- As três regras básicas da sobrecarga de operador em C++ [1]:
  - Se o significado de um operador não for obviamente claro e indiscutível, provavelmente ele não deveria estar sendo sobrecarregado. Forneça uma função para fazer a operação que você quer.
  - Atenha-se à semântica do operador. Seu compilador vai aceitar qualquer significado de operação, mas o usuário da operação vai esperar que a + b *some* os elementos a e b.
  - Forneça todas as operações de um conjunto de operações relacionadas. Por exemplo, se o seu tipo suporta a + b, o usuário espera que também exista a += b.

[1] Operator Overloading on StackOverflow C++ FAQ

Implementando um iterador simples para uma lista encadeada

- Objetivo: Percorrer a lista e imprimir seus elementos da seguinte forma:

```
List list;

(...)

for (List::iterator it = list.begin();
        it != list.end(); ++it)
{
  std::cout << *it << " ";
}
```

Implementando um iterador simples para uma lista encadeada

Para isso é preciso:

- Passo 1: Implementar uma classe **iterator** simples, que:
  - Aponte para um elemento da lista;
  - Sobrescreva o operador * retornando o valor do elemento apontado;
  - Sobrescreva o operador ++ que permita avançar para o próximo elemento;
  - Sobrescreva o operador != que permita dizer quando dois iteradores são diferetnes.

Implementando um iterador simples para uma lista encadeada

- Passo 2: Implementar/modificar a classe List, de forma que ela:
  - Mantenha um nó que represente o fim da lista. Normalmente, isso é feito utilizando um nó denominado **cabeça** (ou sentinela). Este é um nó que não pertence de fato à lista mas está ligado a ela de forma circular. O último elemento real da lista aponta para o nó cabeça, e o nó cabeça aponta para o primeiro elemento real da lista;
  - Contenha um método **begin()** que retorna um iterator para o primeiro elemento real (head->next);
  - Contenha um método **end()** que retorne um iterator que aponte o nó cabeça.

Passo 1:

```
class iterator
{
  friend List;

public:
  int& operator * ()
  { return node->val; }

  iterator& operator++()
  { node = node->next;
    return *this; };

  bool operator != (const iterator& other)
  { return node != other.node; }

private:
  iterator(Node* node) : node(node) {}
  Node* node;
};
```

Passo 2:

```
class List
{
public:
  (...)

  iterator begin()
  { return iterator(head->next); }

  iterator end()
  { return iterator(head); }

  (...)

private:
  Node* head;
};
```

Passo 2:

Método **push_back** utilizando o conceito de cabeça da lista:

```
void List::push_back(const int& element)
{
  Node* node = new Node();
  node->val = element;
  node->next = head;
  node->prev = head->prev;
  head->prev->next = node;
  head->prev = node;
}
```

Referências

- Tutoriais, referências de funções, fórum, etc: http://www.cplusplus.com
- Livros:
  - Drozdek, Adam. Data Structures and Algorithms in C++. Pacific Grove, CA: Brooks/Cole, 2001.
  - Paul J. Deitel. 2010. C++ how to Program. P.J. Deitel, H.M. Deitel (7th ed.). Pearson Education.
  - Scott Meyers. 1998. Effective C++ (2nd Ed.): 50 Specific Ways to Improve Your Programs and Designs. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
  - Scott Meyers. 2014. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 (1st ed.). O'Reilly Media, Inc.
- E muito material na Internet...