Aula 7

Funções amigas e Sobrecarga de operadores

Programação II

Prof. Sandino Jardim | CC-UFMT-CUA

Função Amiga

- Definição
  - Função definida fora de uma classe, mas que possui acesso aos membros privados e protegidos dela
  - Isto é, acessa a implementação como um membro da classe, sem o ser.
  - Mesmo definida fora, deve ter seu protótipo declarado internamente à classe
- Principais utilidades:
  - Casos especiais onde membros privados da classe precisam ser acessados sem o uso de objetos desta;
  - Sobrecarga de operadores;

Função Amiga

- Outro exemplo de uso:
  - Definição de um operador/função que multiplica uma matriz por um vetor, onde ambos são classes diferentes;
  - Cada um possui sua respectiva representação e provê um conjunto de operações para manipular objetos do mesmo tipo;
  - A rotina de multiplicação, se inserida como membro das classes, exigiria a implementação em ambas e acesso aos dados privados;
    - Também exigiria a invocação a partir de objetos de uma das duas classes
  - O acesso a atributos de classes não é recomendado ser público
  - Solução: declarar uma função de multiplicação como **amiga** de ambas

Exemplo

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

class Box {
 double width;
 public:
  friend void printWidth( Box box );
  void setWidth( double wid );
};


void Box::setWidth( double wid )
{ width = wid; }

void printWidth( Box box ) {
 cout << "Width of box : " << box.width <<endl;
}
```

```
int main() {
        Box box;
        box.setWidth(10.0);
        printWidth( box );
        return 0;
}
```

Sobrecarga de Operadores

- Caso especial de polimorfismo onde diferentes operadores possuem diferentes implementações dependendo de seus argumentos

```
class GenericObject{
        //...
}

int main(){
        GenericObject A, B, C;

        C = A + B; //ao invés de C = soma(A,B)
}
```

Conceitos básicos

- Funções com nomes especiais
  - Palavra-chave `operator` seguida do símbolo que se deseja sobrecarregar
  - Exige tipo de retorno, lista de parâmetros e corpo
- Possui o mesmo número de parâmetros do operador original
  - Operador unário – um parâmetro
  - Operador binário – dois parâmetros
    - Operando da esquerda deve ser o primeiro parâmetro
- Quando são definidos como métodos, primeiro parâmetro recebe o ponteiro `this`

Conceitos básicos (2)

- Funções devem ou ser membros de classes ou possuir ao menos um parâmetro de classe

```
// erro: tentativa de redefinição do operador de soma para inteiros
int operator+(int, int);
```

- Nem todos os operadores podem ser sobrecarregados
- Somente operadores existentes podem ser sobrecarregados
  - Não é possível inventar novos operadores (p.ex `operator**`)
- Alguns operadores são unários e binários (`+`, `-`, `*`, `&`)
- Ordens de precedência são mantidas

Conceitos básicos (3)

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

Operadores que não podem ser sobrecarregados			
::	.*	.	?:

Chamada de operador sobrecarregado

- Pode ser chamado utilizando o operador como é comumente utilizado ou como uma função tradicional

```
// equivalent calls to a nonmember operator function
data1 + data2;              // normal expression
operator+(data1, data2);    // equivalent function call
```

- O mesmo vale para sobrecarga como método de classe

```
data1 += data2;             // expression-based ‘‘call’’
data1.operator+=(data2);    // equivalent call
```

Operadores não recomendados

- Operadores lógicos (`&&` e `||`) sempre avaliam o operando da esquerda antes do da direita
- O operando da direita será avaliado *sse* o operando da esquerda não determina o resultado
- A sobrecarga destes operadores não preserva estas propriedades

Operador de saída <<

- Permite configurar uma saída padrão para objetos de uma classe:

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
        os << item.isbn() << " " << item.units_sold << " "
           << item.revenue << " " << item.avg_price();

        return os;
}
```

Operador de entrada >>

- Define como se dá a entrada padrão para a classe que o sobrecarrega

```
istream &operator>>(istream &is, Sales_data &item)
{
        double price; // no need to initialize;
        is >> item.bookNo >> item.units_sold >> price;
        if (is)       // check that the inputs succeeded
                item.revenue = item.units_sold * price;
        else
                item = Sales_data(); // input failed: default state

        return is;
}
```

Operadores aritméticos

- Normalmente são definidos como funções não-membros

```
// assumes that both objects refer to the same book
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
        Sales_data sum = lhs; // copy data members from lhs into sum
        sum.units_sold += rhs.units_sold;
        sum.revenue += rhs.revenue;

        return sum;
}
```

Operadores de igualdade

- Normalmente, verifica se membros de objetos são idênticos

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
        return lhs.isbn() == rhs.isbn() &&
        lhs.units_sold == rhs.units_sold &&
        lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
{
        return !(lhs == rhs);
}
```

Operadores compostos

- Devem retornar uma referência para o operando da esquerda

```
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
        units_sold += rhs.units_sold;
        revenue += rhs.revenue;
        return *this;
}
```