Programação Orientada a Objetos

FUNÇÕES AMIGAS

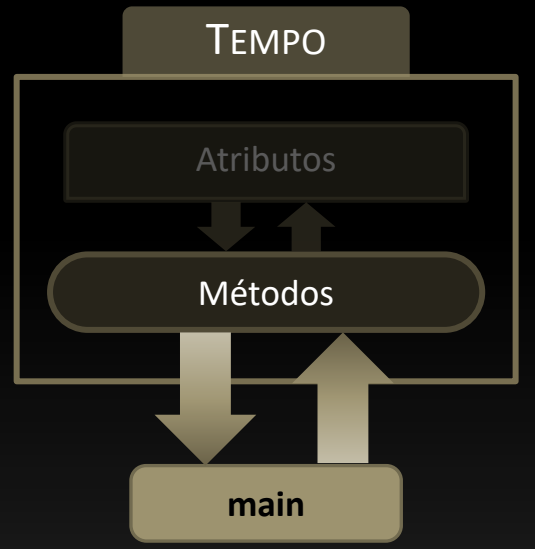
em C++

Introdução

- O controle de acesso em C++ garante que:
  - Membros privados fiquem ocultos
  - Sejam acessados apenas pelos métodos

```
class Tempo
{
private:
   int horas;
   int minutos;

public:
   Tempo();

   void Resetar(int h = 0, int m = 0);
   void Exibir() const;
};
```

```
int main()
{
   Tempo t = {2, 10};
   t.horas = 3;  ✗
   t.Resetar(3); ✓
   ...
}
```



Introdução

- Algumas vezes essa restrição é muito rígida
  - Para resolver isso C++ fornece:
    - Funções amigas
    - Classes amigas
    - Métodos amigos

Uma função amiga tem os mesmos privilégios de acesso que um método da classe.

Motivação

- A sobrecarga de operadores frequentemente gera a necessidade de **funções amigas**:

```
class Tempo
{
private:
   int horas;
   int minutos;

public:
   ...

   Tempo operator+(const Tempo & t) const;
   Tempo operator+(int num) const;

};
```

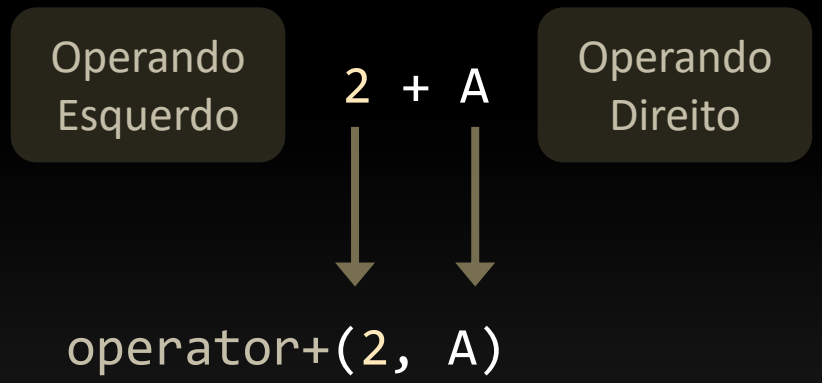
```
Tempo A { 1, 30 };
Tempo B { 1, 20 };
Tempo C;
```

```
C = A + B; ✓ // A.operator+(B)

C = A + 2; ✓ // A.operator+(2);

C = 2 + A; ✗ // 2 não é Tempo
             // 2.operator+(A)
```

Motivação

- Podemos resolver isso com uma função externa
  - Função que não é membro da classe

```
Tempo operator+(int num, const Tempo & t);
```




```
C = A + B; ✓ // A.operator+(B)

C = A + 2; ✓ // A.operator+(2);

C = 2 + A; ✓ // operator+(2, A);
```

Motivação

- Uma função externa não acessa membros privados:

Não é membro da classe Tempo

```
Tempo operator+(int num, const Tempo & t)
{
    Tempo soma;

    soma.horas = num + t.horas;   ✗

    soma.minutos = t.minutos;   ✗

    ...
```

Funções Amigas

- Podemos resolver esse problema:
  - Criando uma função amiga

```
class Tempo
{
private:
   int horas;
   int minutos;

public:
   ...
   friend Tempo operator+(int num, const Tempo & t);
   Tempo operator+(const Tempo & t) const;
   Tempo operator+(int num) const;

};
```

Protótipo da **função amiga** deve ficar dentro da classe.

Funções Amigas

- A função amiga:
  - Tem acesso aos membros privados
  - Mas não é um método da classe

✕ `friend Tempo operator+( ...`

✕ `Tempo Tempo:: operator+( ...`

```
Tempo operator+(int num, const Tempo & t)
{
    Tempo soma;
    soma.horas = num + t.horas;
    soma.minutos = t.minutos;
    return soma;
}
```

Utiliza-se friend apenas no protótipo

A função não pertence a classe Tempo

Funções Amigas

- Após a declaração e definição podemos ter:

```
class Tempo
{
private:
   int horas;
   int minutos;

public:

   ...

   friend Tempo operator+(int num, const Tempo & t);
   Tempo operator+(const Tempo & t) const;
   Tempo operator+(int num) const;
};
```

```
Tempo operator+(int num, const Tempo & t)
{
    Tempo soma;
    soma.horas = num + t.horas;
    soma.minutos = t.minutos;
    return soma;
}
```

```
Tempo A { 1, 30 };
Tempo B { 4, 20 };
Tempo C;

// operator+(2, A);
C = 2 + A;
```

Funções Amigas

- Funções amigas violam a Orientação a Objetos?

```
class Tempo
{
private:
   int horas;
   int minutos;

public:
   ...

};
```

```
Tempo operator+(int num, const Tempo & t)
{
    Tempo soma;

    soma.horas = num + t.horas ;

    soma.minutos = t.minutos;

    return soma;
}
```

O **mecanismo de amizade** parece violar a ocultação de dados

Funções Amigas

- Funções amigas ajudam a definir a interface da classe
  - Conceitualmente, não tem diferença entre somar um inteiro com tempo e somar um tempo com inteiro

```
Tempo A { 1, 30 };
Tempo B { 1, 20 };
Tempo C;

// operator+(2, A);
C = 2 + A;
C = A + 2;
```

```
class Tempo
{
private:
   int horas;
   int minutos;

public:
   ...
   friend Tempo operator+(int num, const Tempo & t);

};
```

A declaração da classe **ainda controla** quem pode acessar os dados.

Funções Amigas

- É possível resolver o problema sem funções amigas:

```
class Tempo
{
private:
   int horas;
   int minutos;

public:

   ...

   Tempo operator+(const Tempo & t) const;
   Tempo operator+(int num) const;
};
```

```
Tempo operator+(int num, const Tempo & t)
{
    // t.operator+(num);
    return t + num;
}
```

```
Tempo A { 1, 30 };
Tempo B { 1, 20 };
Tempo C;

// operator+(2, A);
C = 2 + A;
```

Operador de Inserção

- Mas existem casos em que elas são indispensáveis:
  - Sobrescrever a função `operator<<`

```
Tempo viagem { 2, 15 };
viagem.Exibir();

cout << viagem; // operator<<(cout, viagem);


void operator<< (ostream & os, const Tempo & t)
{
    os << t.horas << " horas, " << t.minutos << " minutos";
}
```

A função precisa acessar os membros privados de Tempo.

Operador de Inserção

- A função pode ser melhorada
  - Ela suporta usos simples:

```
Tempo viagem { 2, 15 };
cout << viagem;
```

  - Mas não suporta uma combinação dos operadores <<

```
Tempo viagem { 1, 15 };
cout << "Tempo da viagem: " << viagem << endl;
```

```
friend void operator<<(ostream & os, const Tempo & t);
```

Operador de Inserção

- O C++ processa a instrução cout:
  - Da esquerda para direita

```
int x = 5;
int y = 10;

cout << x << y;

(cout << x) << y;
```

Resultado é um objeto do tipo ostream

```
cout << x << y;

operator<<( operator<<(cout, x) , y);
```

ostream

```
ostream & operator<<(ostream & os, int n);
```

Retorno precisa ser do tipo ostream

Operador de Inserção

- A implementação do `operator<<`

```
class Tempo
{
private:
   int horas;
   int minutos;

public:
   ...
   friend ostream & operator<<(ostream & os, const Tempo & t);
};


ostream & operator<<(ostream & os, const Tempo & t)
{
    os << t.horas << " horas, " << t.minutos << " minutos";
    return os;
}
```

Operador de Inserção

- A função principal:

```
int main()
{
    Tempo ida { 1, 15 };
    Tempo volta { 1, 10 };

    Tempo viagem = ida + volta;

    cout << "Tempo da viagem: "  << viagem << endl;
};
```

  - Saída:

```
Tempo da viagem: 2 horas, 25 minutos
```

Sobrecarga de Operadores

- A sobrecarga de operadores pode ser feita com:
  - Funções membro
  - Funções não-membro

A função não-membro tem que ser amiga se ela precisar acessar os atributos da classe.

```
A = B + C
A = B.operator+(C);
A = operator+(B,C);
```

```
Tempo operator+(const Tempo & t) const;                 // membro
Tempo operator+(const Tempo & t1, const Tempo & t2);    // não-membro
```

Resumo

- As funções amigas:
  - Ajudam a especificar a interface das classes
    - Membros privados podem ser acessados:
      - Funções membro da classe
      - Funções amigas
  - Permitem sobrescrever operadores quando o operando esquerdo não é um objeto da classe

```
Tempo a, b, total;
cin >> a >> b;        // operator>>
total = a + b;        // operator+
cout << total;        // operator<<
```