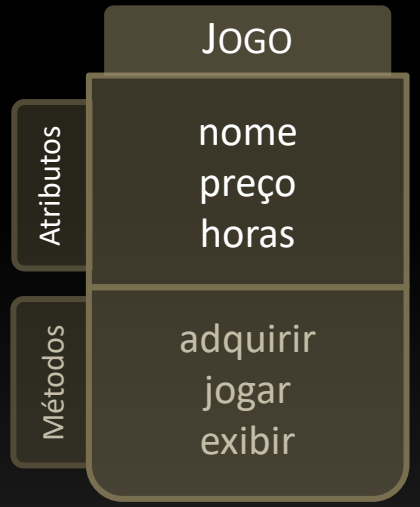
Programação Orientada a Objetos

SOBRECARGA DE OPERADORES

em C++

Introdução

- Uma classe define um novo tipo
  - Um tipo é composto por dados e operações
  - Variáveis desse novo tipo são chamadas de objetos




```
class Jogo
{
private:
   string nome;
   float preco;
   int horas;

public:
   void adquirir(const string & nome, float valor);
   void jogar(int tempo);
   void exibir();
};
```

```
int main()
{
   Jogo gears;
   Jogo doom;
}
```

Introdução

- A linguagem C++ tem como um de seus objetivos fazer **classes** e **objetos** trabalharem de forma similar às **variáveis** e **tipos primitivos**

```
int main()
{
   int n1 = 5;
   int n2 = 10;
   int num;

   // faz a soma
   num = n1 + n2;
}
```

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

public:
   ...
};
```

```
int main()
{
   Tempo t1 = {2, 10};
   Tempo t2 = {1, 30};
   Tempo t;

   // faz a soma?
   t = t1 + t2;
}
```

Sobrecarga de Operadores

- A sobrecarga de operador para classes funciona com a mesma ideia e da mesma forma que a sobrecarga de funções
  - Tem-se o mesmo nome de função
  - Agindo sobre diferentes tipos de dados

```
void exibir(string str, int val);
void exibir(double num, int val);
void exibir(long num, int val);
void exibir(int num);
void exibir(string str);
```

O Compilador consegue distinguir pelo tipo e quantidade de argumentos.

Sobrecarga de Operadores

- **C++** já faz a sobrecarga de vários operadores:
  - Multiplicação e operador de indireção (*)

```
int * ptr = new int;
int total = 5 * 4;          // * usado como multiplicação
*ptr = 10;                  // * usado como operador de indireção
```

  - AND bit-a-bit e operador de endereço (&)

```
int estado = 180;
int mascara = estado & 5;   // & usado como operador bit-a-bit
cout << &estado;            // & usado como operador de endereço
```

Sobrecarga de Operadores

- Para sobrecarregar um operador utiliza-se uma forma especial de função, chamada de função operador:

```
operator+()  // sobrecarrega operador de soma
operator*()  // sobrecarrega operador de multiplicação

operatorop(lista-de-argumentos)
```

Operador válido e existente em C++

Sobrecarga de Operadores

- Definir um operator+ para uma **classe**:
  - Define a operação de soma para objetos da classe
  - Permite o compilador trocar + por operator+()

Você escreve:

```
int main()
{
   Tempo t1 = {2, 10};
   Tempo t2 = {1, 30};
   // exibe a soma
   cout << t1 + t2;
}
```

Compilador entende:

```
int main()
{
   Tempo t1 = {2, 10};
   Tempo t2 = {1, 30};
   // exibe a soma
   cout << t1.operator+(t2);
}
```

A Classe Tempo

- Construindo um exemplo:

```
class Tempo
{
private:
   int horas;                          // quantidade de horas
   int minutos;                        // quantidade de minutos

public:
   Tempo();                            // construtor padrão
   Tempo(int h, int m = 0);            // construtor com horas e minutos

   void  Adicionar(int h, int m);      // adiciona horas e minutos
   void  Resetar(int h = 0, int m = 0);// modifica tempo armazenado
   Tempo Somar(const Tempo & t) const; // retorna soma dos tempos
   void  Exibir() const;               // exibe tempo na tela
};
```

» Tempo.h

A Classe Tempo

```cpp
// definição da classe Tempo
#include <iostream>
#include "Tempo.h"
using namespace std;

Tempo::Tempo()
{
    horas = 0;
    minutos = 0;
}

Tempo::Tempo(int h, int m)
{
    horas = h;
    minutos = m;
}

...
```

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

public:
   Tempo();
   Tempo(int h, int m = 0);

   void  Adicionar(int h, int m);
   void  Resetar(int h = 0, int m = 0);
   Tempo Somar(const Tempo & t) const;
   void  Exibir() const;
};
```

» Tempo.cpp

A Classe Tempo

```cpp
void Tempo::Adicionar(int h, int m)
{
    horas += h;
    minutos += m;

    // minutos excedentes viram horas
    horas += minutos / 60;
    minutos %= 60;
}

void Tempo::Resetar(int h, int m)
{
    horas = h;
    minutos = m;
}

...
```

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

public:
   Tempo();
   Tempo(int h, int m = 0);

   void  Adicionar(int h, int m);
   void  Resetar(int h = 0, int m = 0);
   Tempo Somar(const Tempo & t) const;
   void  Exibir() const;
};
```

» Tempo.cpp

A Classe Tempo

```cpp
Tempo Tempo::Somar(const Tempo & t) const
{
    Tempo soma;
    soma.horas = horas + t.horas;
    soma.minutos = minutos + t.minutos;

    // minutos excedentes viram horas
    soma.horas += soma.minutos / 60;
    soma.minutos %= 60;

    return soma;
}

void Tempo::Exibir() const
{
    cout << horas << " horas, "
         << minutos << " minutos";
}
```

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

public:
   Tempo();
   Tempo(int h, int m = 0);

   void  Adicionar(int h, int m);
   void  Resetar(int h = 0, int m = 0);
   Tempo Somar(const Tempo & t) const;
   void  Exibir() const;
};
```

» Tempo.cpp

A Classe Tempo

- O retorno da função Somar não pode ser uma referência

```
Tempo Tempo::Somar(const Tempo & t) const
{
    Tempo soma;
    soma.minutos = minutos + t.minutos;
    soma.horas = horas + t.horas;

    // minutos excedentes viram horas
    soma.horas += soma.minutos / 60;
    soma.minutos %= 60;

    return soma;
}
```

Não se deve retornar uma referência para uma variável local.

Ela deixa de existir no final da função.

A Classe Tempo

```cpp
#include <iostream>
#include "Tempo.h"
using std::cout;
using std::endl;

int main()
{
    Tempo projetando;
    Tempo codificando { 2, 40 };
    Tempo corrigindo { 5, 55 };
    Tempo total;

    cout << "Tempo projetando = ";
    projetando.Exibir();
    cout << endl;

    ...
```

```cpp
    ...

    cout << "Tempo codificando = ";
    codificando.Exibir();
    cout << endl;

    cout << "Tempo corrigindo = ";
    corrigindo.Exibir();
    cout << endl;

    total = codificando.Somar(corrigindo);
    cout << "Tempo Total = ";
    total.Exibir();
    cout << endl;

    return 0;
}
```

» PassaTempo.cpp

A Classe Tempo

- Saída do programa:

```
Tempo projetando = 0 horas, 0 minutos
Tempo codificando = 2 horas, 40 minutos
Tempo corrigindo = 5 horas, 55 minutos
Tempo Total = 8 horas, 35 minutos
```

  - O método somar não altera os objetos envolvidos na soma

```
total = codificando.Somar(corrigindo);
cout << "Tempo Total = ";
total.Exibir();
```

Operador de Adição

- O suporte ao operador de adição é obtido:
  - Apenas pela troca do nome do método Somar por operator+

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

public:
   ...
   Tempo Somar(const Tempo & t) const;
};
```

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

public:
   ...
   Tempo operator+(const Tempo & t) const;
};
```

Operador de Adição

- A implementação permanece a mesma
  - Apenas com a troca dos nomes

```
Tempo Tempo::Somar (const Tempo & t) const
{

    Tempo soma;
    soma.minutos = minutos + t.minutos;
    soma.horas = horas + t.horas;

    // minutos excedentes viram horas
    soma.horas += soma.minutos / 60;
    soma.minutos %= 60;

    return soma;

}
```

```
Tempo Tempo::operator+ (const Tempo & t) const
{

    Tempo soma;
    soma.minutos = minutos + t.minutos;
    soma.horas = horas + t.horas;

    // minutos excedentes viram horas
    soma.horas += soma.minutos / 60;
    soma.minutos %= 60;

    return soma;

}
```

Operador de Adição

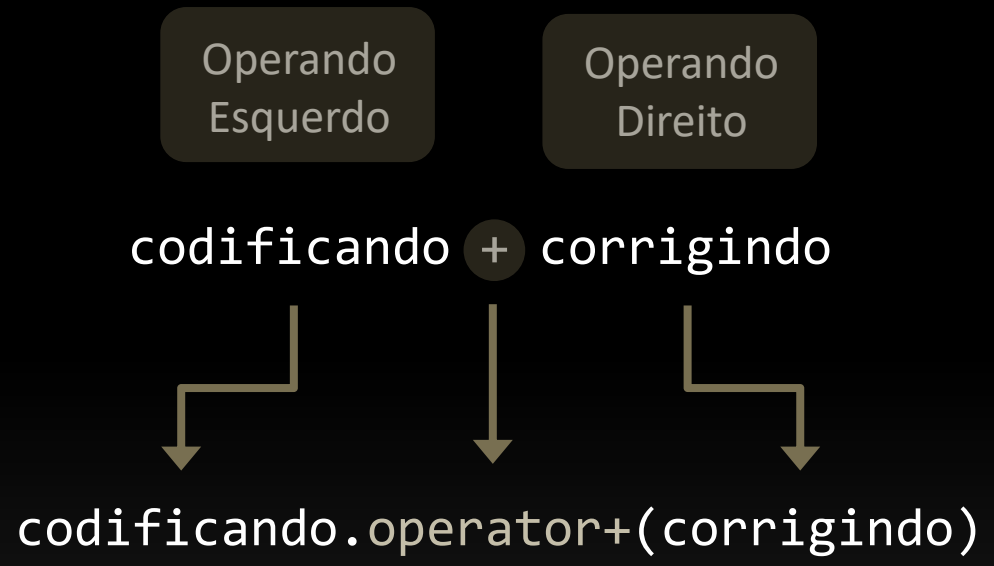
- A soma de tempos fica bem mais simples:

```
int main()
{
    Tempo projetando;
    Tempo codificando { 2, 40 };
    Tempo corrigindo { 5, 55 };
    Tempo total;


    total = codificando.Somar(corrigindo);

    cout << "Tempo Total = ";
    total.Exibir();
    cout << endl;

    ...
}
```

```
int main()
{
    Tempo projetando;
    Tempo codificando { 2, 40 };
    Tempo corrigindo { 5, 55 };
    Tempo total;

    // total = codificando.operator+(corrigindo);

    total = codificando + corrigindo;

    cout << "Tempo Total = ";
    total.Exibir();
    cout << endl;

    ...
}
```

Operador de Adição

- O uso do operador é equivalente a chamada da função



Internamente, o compilador converte o operador + em uma chamada de função.

Restrições

- Usar um operador como método de uma classe:
  - Obriga o operando esquerdo a ser um objeto

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

public:
   ...

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

};
```

```
Tempo a { 1, 30 };
Tempo b { 1, 20 };
Tempo t;
```

```
t = a + b;  ✓  // uso correto
               // a.operator+(b)

t = a + 2;  ✓  // uso correto
               // a.operator+(2);

t = 2 + a;  ✗  // uso inválido
               // 2 não é um objeto
               // 2.operator+(a)
```

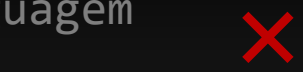
Restrições

- Os operadores não precisam ser métodos

```
Tempo operator+(const Tempo & a, const Tempo & b);
Tempo operator+(int a, const Tempo & b);
Tempo operator+(const Tempo & a, int b);
```

  - Mas pelo menos um dos operandos deve ser um objeto de uma classe definida pelo programador
    - Não é possível redefinir operações sobre tipos padrão

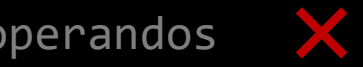
```
// argumentos são tipos padrão da linguagem
int operator+(int a, int b) const;
```



Restrições

- Não é possível:
  - Alterar as regras de uso do operador
    - Alterar a precedência, número de argumentos, etc.

```
+ total; // operador + requer dois operandos
```



  - Criar operadores
    - Só é possível sobrescrever operadores existentes

```
// operador @ não existe na linguagem C++
Tempo Tempo::operator@(const Tempo & t) const;
```

Restrições

- Nem todo operador pode ser sobrecarregado

sizeof	.	.*	::	?:
typeid	const_cast	dynamic_cast	reinterpret_cast	static_cast

- Os operadores abaixo só podem ser sobrecarregados por métodos de uma classe, não através de funções:

=	()	[]	->

Tabela de Operadores

- Os seguintes **operadores** podem ser sobrecarregados *:

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

* Use o bom senso, não modifique o significado do operador

Resumo

- A **sobrecarga de operadores** permite:
  - Tratar classes como tipos primitivos
  - Simplificar o uso de objetos dessas classes

```
Tempo a, b, total;

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

- A maior parte dos operadores podem ser sobrecarregados
  - Através de métodos ou funções