

# Linguagem C++ II

06/12/2023

# Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- Diferentes **classes** simples, exemplificando o **overloading de operadores**, a criação de **classes derivadas**, o **polimorfismo**, e as **classes genéricas** --- que podem explorar e utilizar

# Sumário

- Classes
- Classes Derivadas – Herança
- Polimorfismo
- Classes Genéricas
- Referência

# Classes

# Classes e Objetos

- Atributos de instância vs Atributos de classe (**static**)
- Métodos de instância vs Métodos de classe (**static**)
- Visibilidade: **private**, **protected**, **public**
- O ponteiro **this** – referência para o objeto que invoca o método
- O qualificador **const**
- O qualificador **friend**

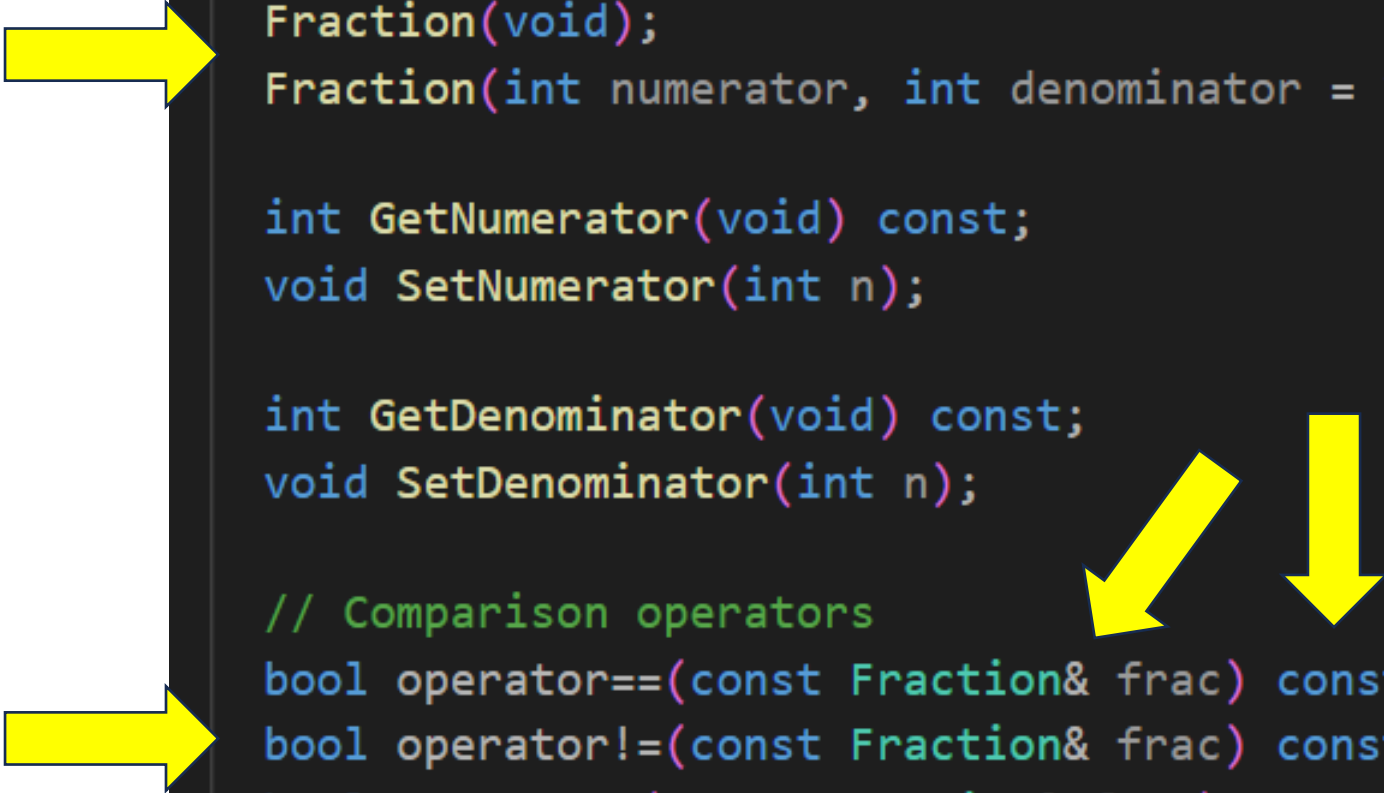
# Classes e Objetos

- Construtores e Destrutor
- Construtor de Cópia (**Copy Constructor**) – quando é necessário ?
- Operador de Atribuição (**operator=**) – quando é necessário ?
- Getters & Setters
- Overloaded operators : **operator ==**, **operator<**, etc.
- friend **operator <<**

# Exemplo – A classe Fraction


# Fraction.h

```
class Fraction {  
public:  
    Fraction(void);  
    Fraction(int numerator, int denominator = 1);  
  
    int GetNumerator(void) const;  
    void SetNumerator(int n);  
  
    int GetDenominator(void) const;  
    void SetDenominator(int n);  
  
    // Comparison operators  
    bool operator==(const Fraction& frac) const;  
    bool operator!=(const Fraction& frac) const;  
    bool operator<(const Fraction& frac) const;
```






# Fraction.h



```
// Binary operators
```


```
Fraction operator+(const Fraction& frac) const;
```




```
Fraction operator-(const Fraction& frac) const;
```

```
Fraction operator*(const Fraction& frac) const;
```

```
Fraction operator/(const Fraction& frac) const;
```




```
double ToDouble(void) const;
```



```
friend std::ostream& operator<<(std::ostream& out, const Fraction& frac);
```

```
private:
```



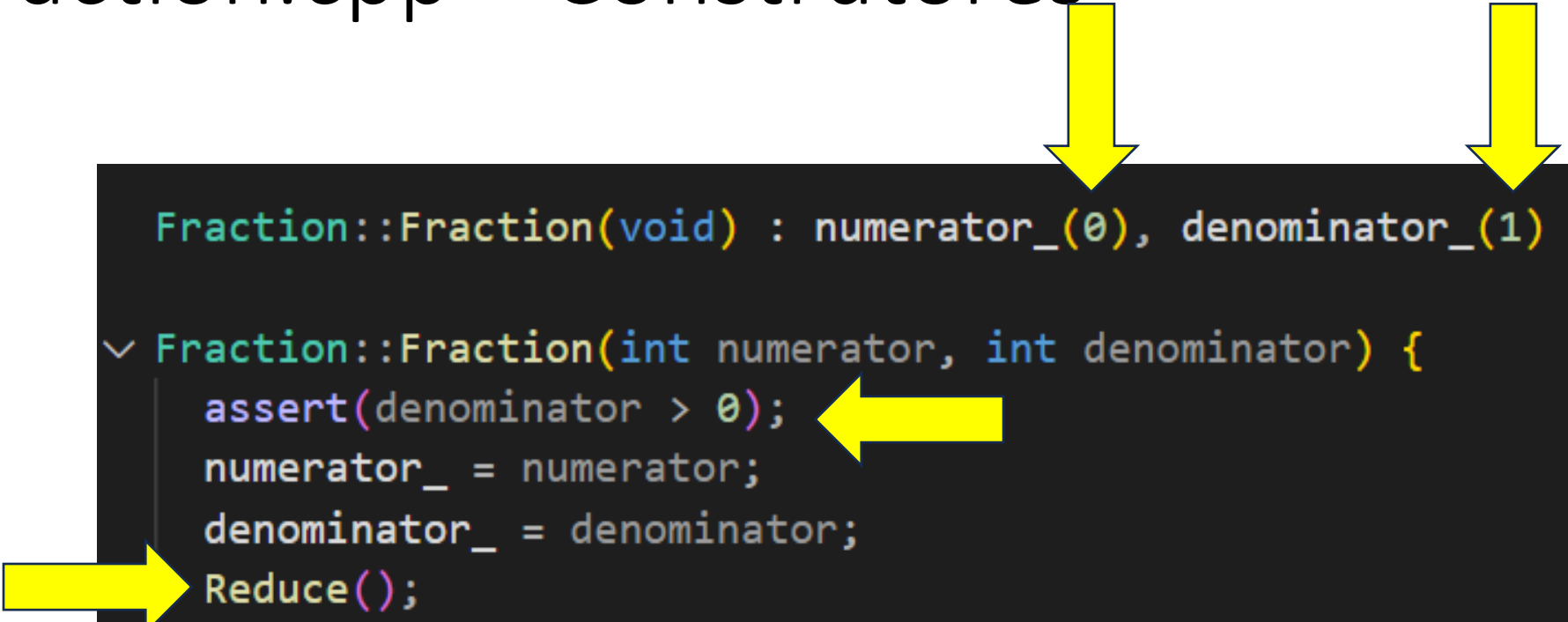
```
void Reduce(void);
```

```
int numerator_;
```

```
int denominator_; // ALWAYS POSITIVE !!!
```

```
};
```

# Fraction.cpp – Construtores



```
Fraction::Fraction(void) : numerator_(0), denominator_(1) {}  
  
✓ Fraction::Fraction(int numerator, int denominator) {  
    assert(denominator > 0);  
    numerator_ = numerator;  
    denominator_ = denominator;  
    Reduce();  
}
```

# Fraction.cpp – Getters & Setters

```
int Fraction::GetNumerator(void) const { return numerator_; }  
✓ void Fraction::SetNumerator(int n) {  
    numerator_ = n;  
    Reduce();  
}  
  
int Fraction::GetDenominator(void) const { return denominator_; }  
✓ void Fraction::SetDenominator(int n) {  
    assert(n > 0);  
    denominator_ = n;  
    Reduce();  
}
```

# Fraction.cpp – Operadores de comparação

```
// Comparison operators
```

```
bool Fraction::operator==(const Fraction& frac) const {  
    return (numerator_ == frac.numerator_) && (denominator_ == frac.denominator_);  
}
```

```
bool Fraction::operator!=(const Fraction& frac) const {  
    return !(*this == frac);  
}
```

```
bool Fraction::operator<(const Fraction& frac) const {  
    // Not the smartest way  
    return ToDouble() < frac.ToDouble();  
}
```

# Fraction.cpp – Operadores aritméticos

Unary operator


```
✓ Fraction Fraction::operator-(void) const {  
    Fraction res(-numerator_, denominator_);  
    return res;  
}
```

Binary operators


```
✓ Fraction Fraction::operator+(const Fraction& frac) const {  
    Fraction res(*this);  
    ✓ if (res.denominator_ == frac.denominator_) {  
        res.numerator_ += frac.numerator_;  
    } else {  
        res.numerator_ =  
            res.numerator_ * frac.denominator_ + frac.numerator_ * res.denominator_;  
        res.denominator_ *= frac.denominator_;  
    }  
    ✓ res.Reduce();  
    return res;  
}
```

# Fraction.cpp – Operadores aritméticos



```
Fraction Fraction::operator-(const Fraction& frac) const {  
    return *this + (-frac);  
}  
  
Fraction Fraction::operator*(const Fraction& frac) const {  
    Fraction res(*this);  
  
    res.numerator_ *= frac.numerator_;  
    res.denominator_ *= frac.denominator_;  
  
    res.Reduce();  
    return res;  
}
```



# Fraction.cpp – Operadores aritméticos



```
Fraction Fraction::operator/(const Fraction& frac) const {  
    assert(frac.numerator_ != 0);  
  
    Fraction res(*this);  
  
    res.numerator_ *= frac.denominator_;  
    res.denominator_ *= frac.numerator_;  
  
    // Ensure the denominator is POSITIVE  
    if (res.denominator_ < 0) {  
        res.numerator_ *= -1;  
        res.denominator_ *= -1;  
    }  
  
    res.Reduce();  
    return res;  
}
```



# Fraction.cpp – Métodos auxiliares


```
double Fraction::ToDouble(void) const {  
    return (double)numerator_ / (double)denominator_;  
}  
  
std::ostream& operator<<(std::ostream& os, const Fraction& frac) {  
    os << frac.numerator_ << " / " << frac.denominator_;  
    return os;  
}  
  
void Fraction::Reduce(void) {  
    int gcd = std::gcd(numerator_, denominator_); // Since C++17  
    if (gcd != 1) {  
        numerator_ /= gcd;  
        denominator_ /= gcd;  
    }  
}
```





# Exemplos de utilização

```
Fraction zero; // Has value ZERO
Fraction fraction_1;
Fraction fraction_2(5);
Fraction fraction_3(2, 4);

std::cout << "1st fraction: " << fraction_1 << " = " << fraction_1.ToDouble()
| | | | | << std::endl;
```




```
std::cout << fraction_1 << " is equal to " << fraction_3;
std::cout << " : " << std::boolalpha << (fraction_2 == fraction_3)
| | | | | << std::endl;
```



// Arithmetic operations

```
std::cout << fraction_2 << " + " << fraction_3;
std::cout << " = " << fraction_2 + fraction_3 << std::endl;
```



# Tarefas

- **Analisar** o código da classe Fraction
- **Desenvolver** outros exemplos de utilização

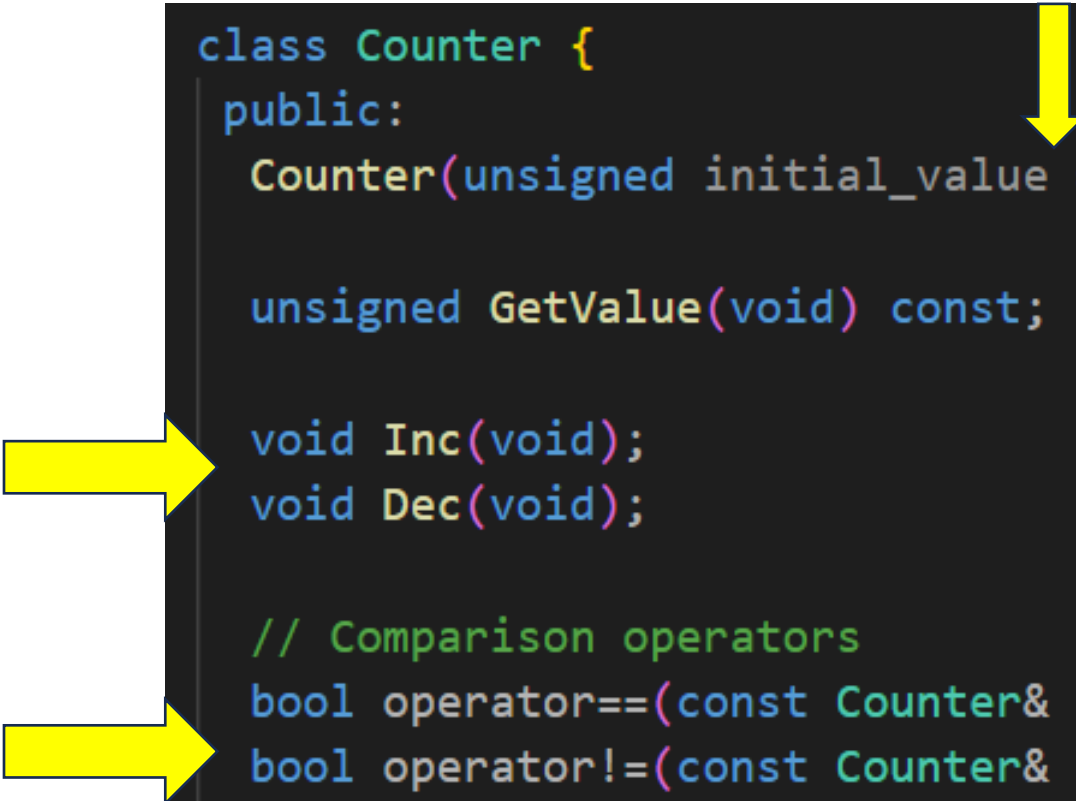
# Classes Derivadas

# Exemplo

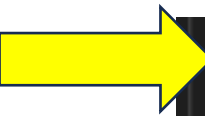
- Counter & LimitedCounter

# Counter.h – Classe de base

```
class Counter {  
    public:  
        Counter(unsigned initial_value = 0);  
  
        unsigned GetValue(void) const;  
  
        void Inc(void);  
        void Dec(void);  
  
        // Comparison operators  
        bool operator==(const Counter& c) const;  
        bool operator!=(const Counter& c) const;  
        bool operator<(const Counter& c) const;
```



# Counter.h – Classe de base




```
// Postfix operators
// Extra parameter to allow for prefix and postfix notations
Counter operator++(int);
Counter operator--(int);

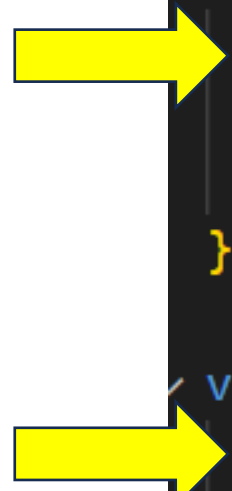
// Prefix operators
Counter& operator++(void);
Counter& operator--(void);

friend std::ostream& operator<<(std::ostream& out, const Counter& c);


protected:
    unsigned value_;
};
```




# Counter.cpp



```
void Counter::Inc(void) {  
    if (value_ < std::numeric_limits<unsigned int>::max()) {  
        value_++;  
    }  
}  
  
void Counter::Dec(void) {  
    if (value_ > 0) {  
        value_--;  
    }  
}
```





# Counter.cpp – c++ e c--



```
Postfix operators
// Extra parameter to allow for prefix and postfix notations


Counter Counter::operator++(int) {
    Counter old_counter = *this;
    Inc();
    return old_counter;
}

Counter Counter::operator--(int) {
    Counter old_counter = *this;
    Dec();
    return old_counter;
}
```







# Counter.cpp – ++c e --c

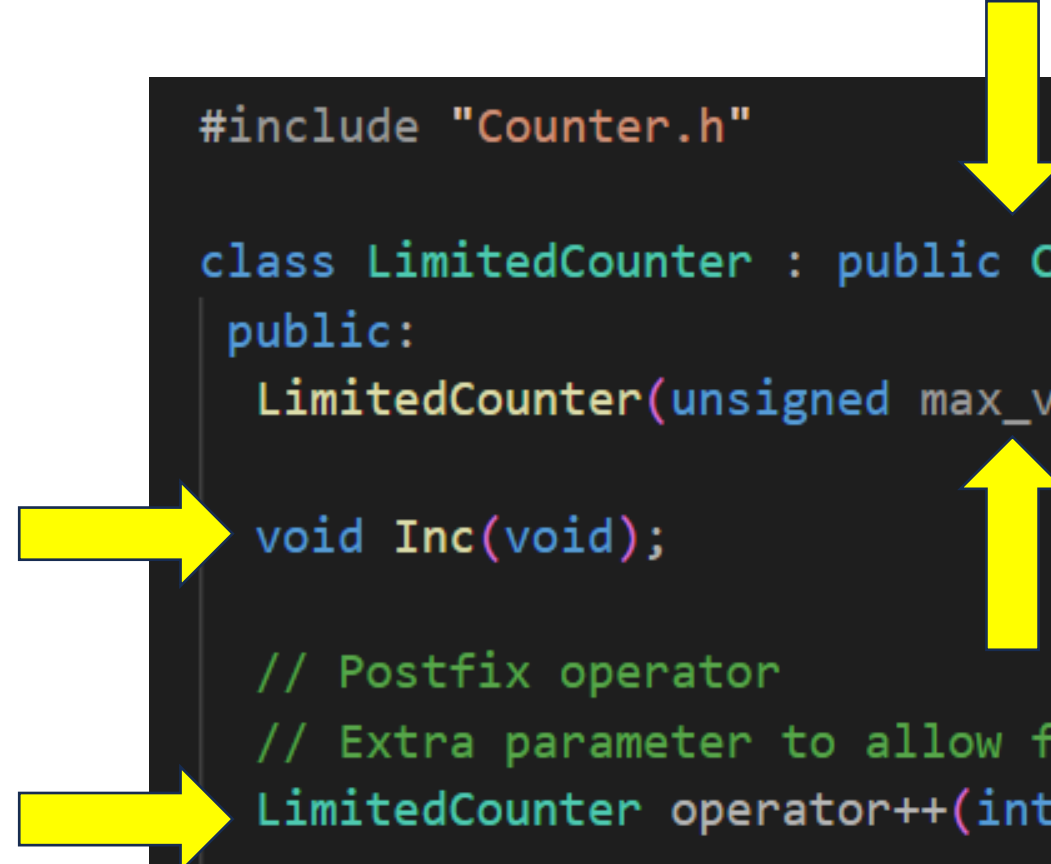


```
Prefix operators
Counter& Counter::operator++(void) {
    Inc();
    return *this;
}

Counter& Counter::operator--(void) {
    Dec();
    return *this;
}
```



# LimitedCounter.h – Classe derivada



The diagram illustrates the relationship between the `LimitedCounter` class and its base class `Counter`. A yellow arrow points from the title "LimitedCounter.h – Classe derivada" down to the `Counter` class name in the code. Another yellow arrow points from the left to the `Inc` method call. A third yellow arrow points from the left to the `operator++` call. A fourth yellow arrow points from the `Inc` method call up to the `operator++` call, indicating that `Inc` is implemented using the postfix increment operator.

```
#include "Counter.h"

class LimitedCounter : public Counter {
public:
    LimitedCounter(unsigned max_value, unsigned initial_value = 0);

    void Inc(void);

    // Postfix operator
    // Extra parameter to allow for prefix and postfix notations
    LimitedCounter operator++(int);

    // operator-- is inherited!!
}
```

# LimitedCounter.h – Atributo adicional

```
// Prefix operator
LimitedCounter& operator++(void);

// operator-- is inherited!!

friend std::ostream& operator<<(std::ostream& out, const LimitedCounter& c);

private:
    unsigned max_value_;
};
```

# LimitedCounter.cpp

```
LimitedCounter::LimitedCounter(unsigned max_value, unsigned initial_value)
    : Counter(initial_value)
    max_value_ = max_value;
}

void LimitedCounter::Inc(void) {
    if (value_ < max_value_) {
        value_++;
    }
}
```

# LimitedCounter.cpp – Method Overriding

```
// Postfix operators
// Extra parameter to allow for prefix and postfix notations

LimitedCounter LimitedCounter::operator++(int) {
    LimitedCounter old_LimitedCounter = *this;
    Inc();
    return old_LimitedCounter;
}

// Prefix operators
LimitedCounter& LimitedCounter::operator++(void) {
    Inc();
    return *this;
}
```

# Exemplos de utilização

```
LimitedCounter c_1(10);
LimitedCounter c_2(5);


std::cout << "1st counter: " << c_1 << std::endl;

std::cout << "2nd counter: " << c_2 << std::endl;

for (int i = 0; i < 12; i++) {
    c_1++;
    ++c_2;
}

std::cout << "1st counter: " << c_1 << std::endl;

std::cout << "2nd counter: " << c_2 << std::endl;
```



# Tarefas

- **Analisar** o código das classes Counter e LimitedCounter
- **Desenvolver** outros exemplos de utilização

# Polimorfismo



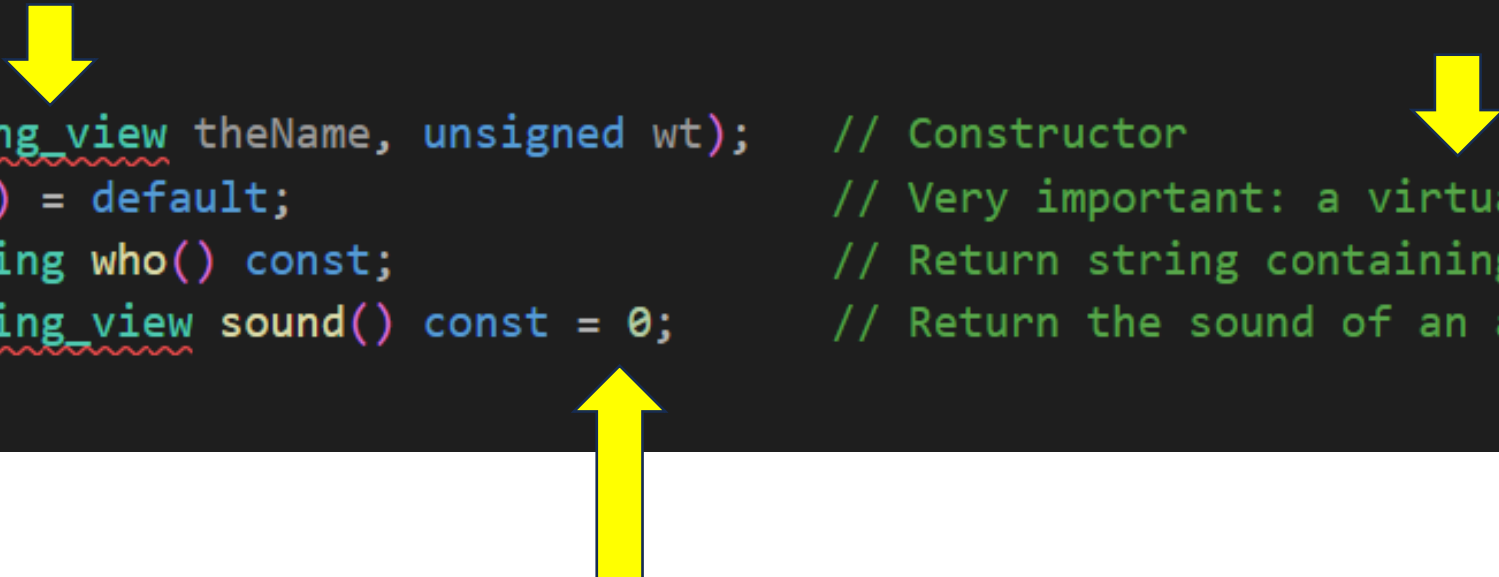
# Exemplo

## – Animais & Zoo

# Animal – Classe de base é abstrata

```
class Animal
{
private:
    std::string name;           // Name of the animal
    unsigned weight;           // Weight of the animal


public:
    Animal(std::string_view theName, unsigned wt); // Constructor
    virtual ~Animal() = default;                 // Very important: a virtual destructor!
    virtual std::string who() const;              // Return string containing name and weight
    virtual std::string_view sound() const = 0;   // Return the sound of an animal
};
```




# Animal – Método `who()` vai ser herdado

```
// Constructor
Animal::Animal(std::string_view theName, unsigned wt)
| | : name(theName), weight(wt)
| | {}

// Return string describing the animal
std::string Animal::who() const ←
|
| return "My name is " + name + ". My weight is " + std::to_string(weight) + " lbs.";
|
| }
```



# Classes Derivadas – Diferentes animais



```
class Sheep : public Animal
```


```
{
```

```
public:
```

```
    using Animal::Animal;
```


```
    std::string_view sound() const override;
```

```
};
```



```
// Inherit constructor
```

```
// Return the sound of a sheep
```



```
class Dog : public Animal
```


```
{
```

```
public:
```

```
    using Animal::Animal;
```

```
    std::string_view sound() const override;
```

```
};
```



```
// Inherit constructor
```

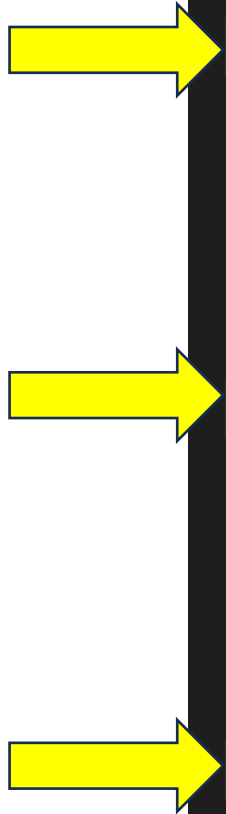
```
// Return the sound of a dog
```

# Classes Derivadas – Overriding

```
// Make like a sheep
std::string_view Sheep::sound() const
{
    return "Baaaa!!";
}

// Make like a dog
std::string_view Dog::sound() const
{
    return "Woof woof!!";
}

// Make like a cow
std::string_view Cow::sound() const
{
    return "Mooooo!!";
}
```



# Zoo – Coleção de (ponteiros para) Animais



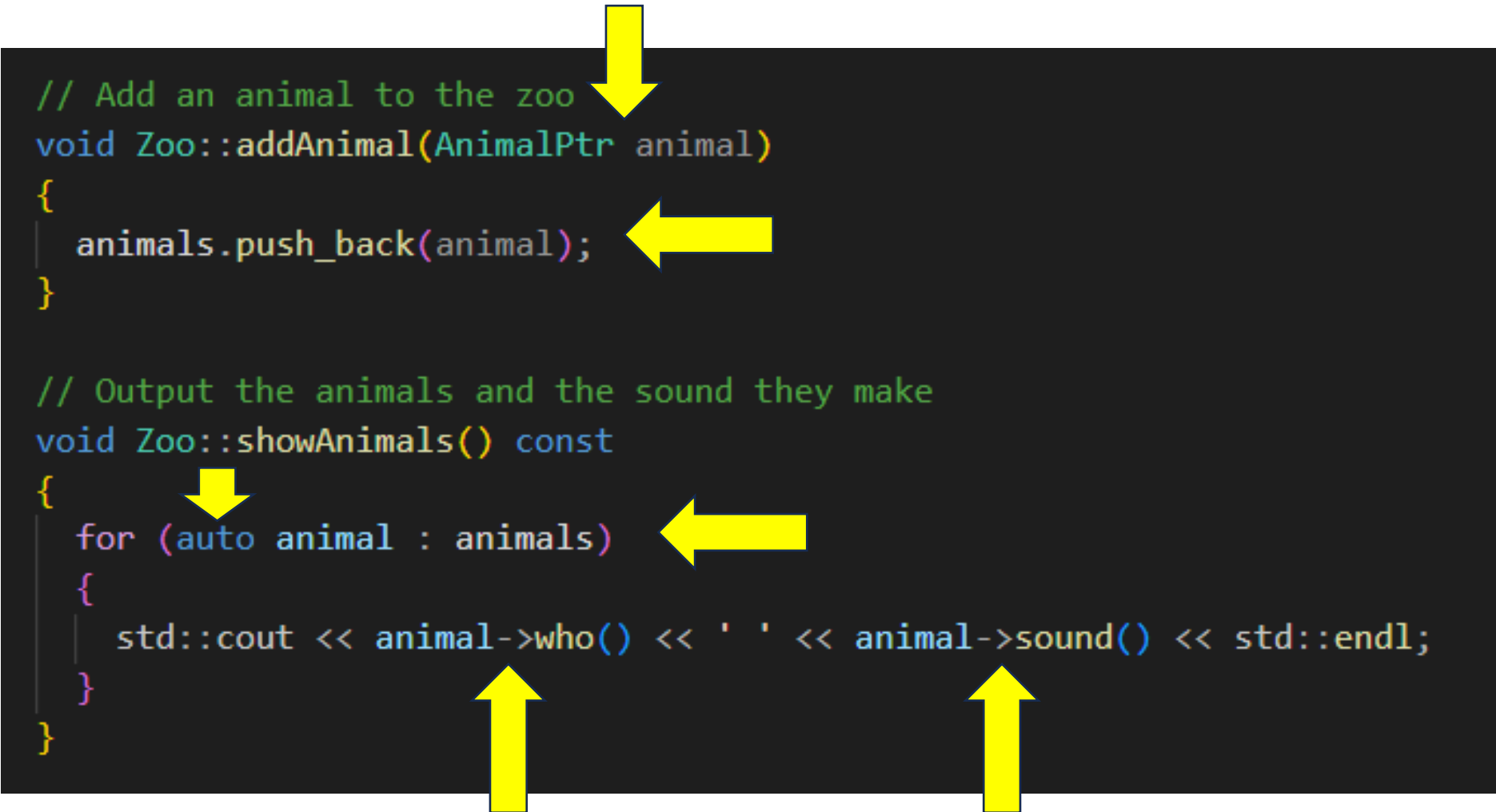
```
using AnimalPtr = std::shared_ptr<Animal>;           // Define a type alias for convenience

class Zoo
{
private:
    std::vector<AnimalPtr> animals;           // Stores pointers to the animals ←

public:
    Zoo() = default;                           // Default constructor for an empty zoo ←
    Zoo(const std::vector<AnimalPtr>& new_animals); // Constructor from a vector of animals
    virtual ~Zoo() = default;                   // Add a virtual destructor to allow classes to safely derive from
    // possible examples of Zoo specializations include SafariPark, Pe
    void addAnimal(AnimalPtr animal);           // Add an animal to the zoo
    void showAnimals() const;                   // Output the animals and the sound they make
};
```



# Zoo – Polimorfismo




```
// Add an animal to the zoo
void Zoo::addAnimal(AnimalPtr animal)
{
    animals.push_back(animal);
}

// Output the animals and the sound they make
void Zoo::showAnimals() const
{
    for (auto animal : animals)
    {
        std::cout << animal->who() << ' ' << animal->sound() << std::endl;
    }
}
```


# Exemplo de utilização

```
size_t nAnimals{}; // Number of animals to be created
std::cout << "How many animals in the zoo? ";
std::cin >> nAnimals;
```



```
Zoo zoo; // Create an empty Zoo
```


```
// Create random animals and add them to the Zoo
```



```
for (size_t i{}; i < nAnimals; ++i) {
```

```
    switch (random(3)) {
```

```
        case 0: // Create a sheep
```



```
        zoo.addAnimal(std::make_shared<Sheep>(
            sheepNames[random(sheepNames.size())],
            minSheepWt + random(maxSheepWt - minSheepWt + 1)));
```

```
        break;
```

```
        case 1: // Create a dog
```

```
        zoo.addAnimal(
```

```
            std::make_shared<Dog>(dogNames[random(dogNames.size())],
```



# Tarefas

- **Analisar** o código da classe de base (**Animal**) e das classes derivadas (**Cow, Dog, Sheep**)
- **Criar** classes derivadas adicionais: p. ex., **Cat, Tiger, Elephant**, etc.
- **Modificar** o exemplo de utilização para usar também essas classes adicionais

# Classes Genéricas

## - Template Classes

# Exemplo

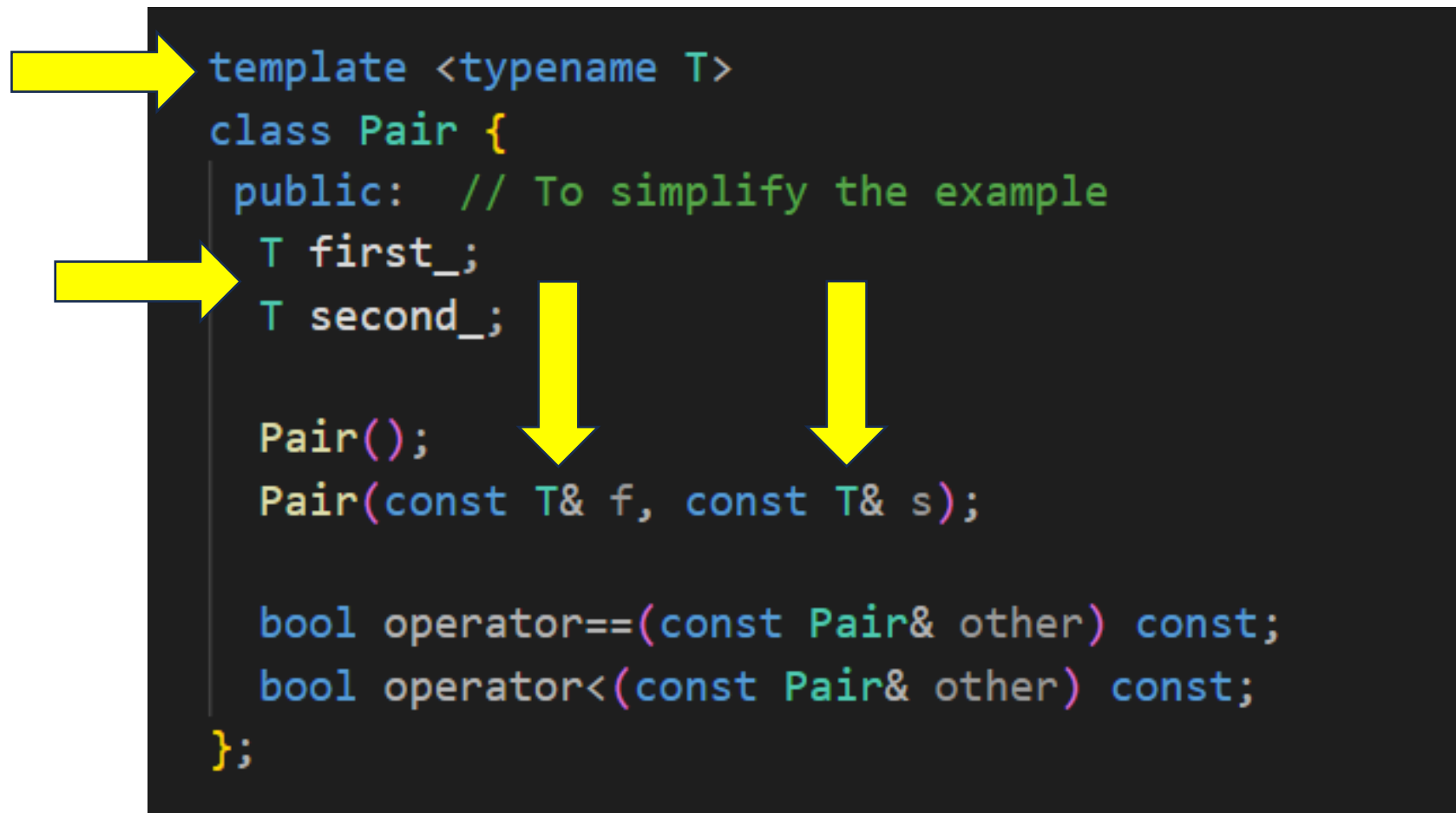
- A classe genérica `Pair<T>`

# Pair.h – Par de elementos do mesmo tipo

```
template <typename T>
class Pair {
public: // To simplify the example
    T first_;
    T second_;

    Pair();
    Pair(const T& f, const T& s);

    bool operator==(const Pair& other) const;
    bool operator<(const Pair& other) const;
};
```



# Pair.h – Operadores

```
// Comparison operators
```

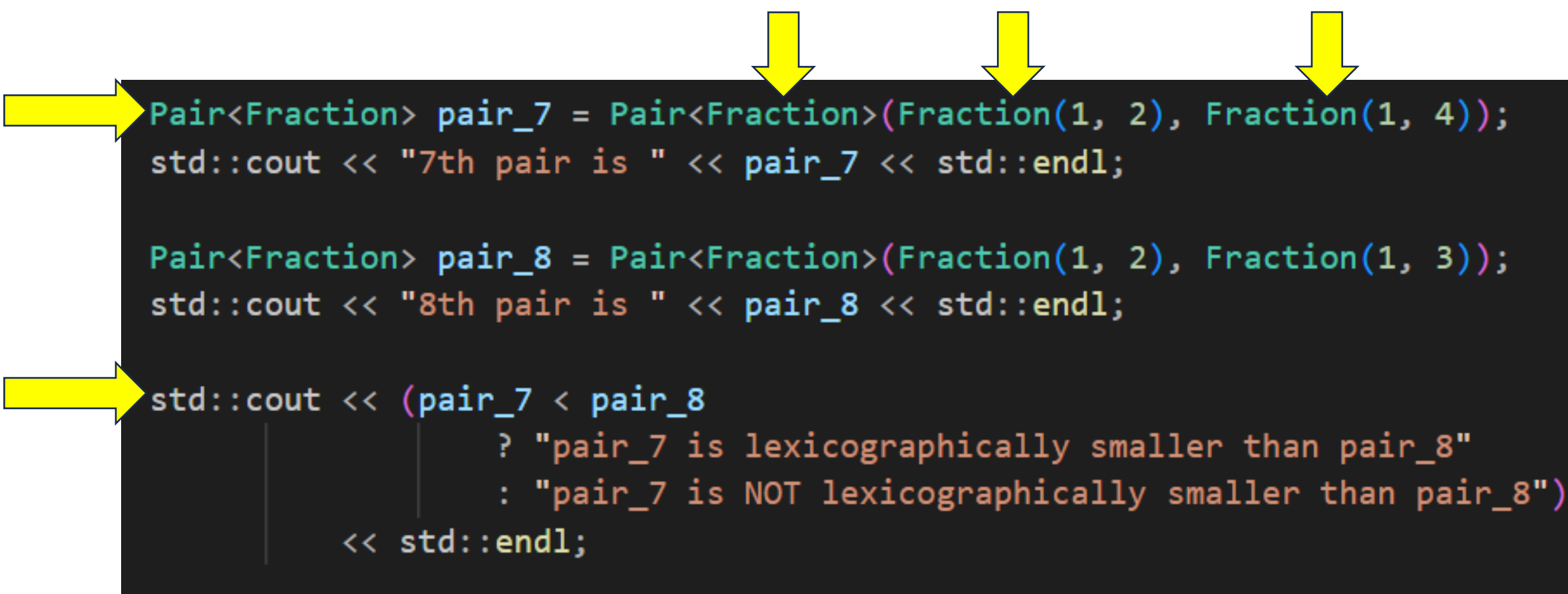
```
template <typename T>
```

```
bool Pair<T>::operator==(const Pair& other) const {  
    return first_ == other.first_ && second_ == other.second_;  
}
```

```
template <typename T>
```

```
bool Pair<T>::operator<(const Pair& other) const {  
    return first_ < other.first_ ||  
           (first_ == other.first_ && second_ < other.second_);  
}
```

# Exemplo de utilização – Pair<Fraction>



```
Pair<Fraction> pair_7 = Pair<Fraction>(Fraction(1, 2), Fraction(1, 4));
std::cout << "7th pair is " << pair_7 << std::endl;

Pair<Fraction> pair_8 = Pair<Fraction>(Fraction(1, 2), Fraction(1, 3));
std::cout << "8th pair is " << pair_8 << std::endl;

std::cout << (pair_7 < pair_8
               ? "pair_7 is lexicographically smaller than pair_8"
               : "pair_7 is NOT lexicographically smaller than pair_8")
            << std::endl;
```

# Tarefas

- **Analisar** o código da classe genérica
- **Analisar** os exemplos de utilização
- **Criar** uma **nova classe genérica**, que permita instanciar famílias de pares de elementos, em que o **tipo do 1º elemento é diferente** do tipo do 2º elemento
- **Modificar** o exemplo de utilização para usar também essa nova classe genérica

# Referência



# Referência

Tomás Oliveira e Silva, *AED Lecture Notes*, 2022