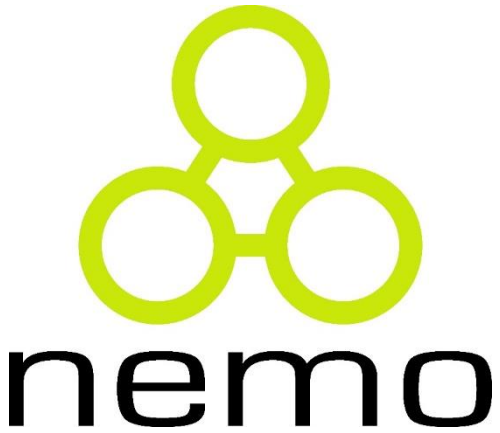


Programação III

Jordana S. Salamon

jssalamon@inf.ufes.br

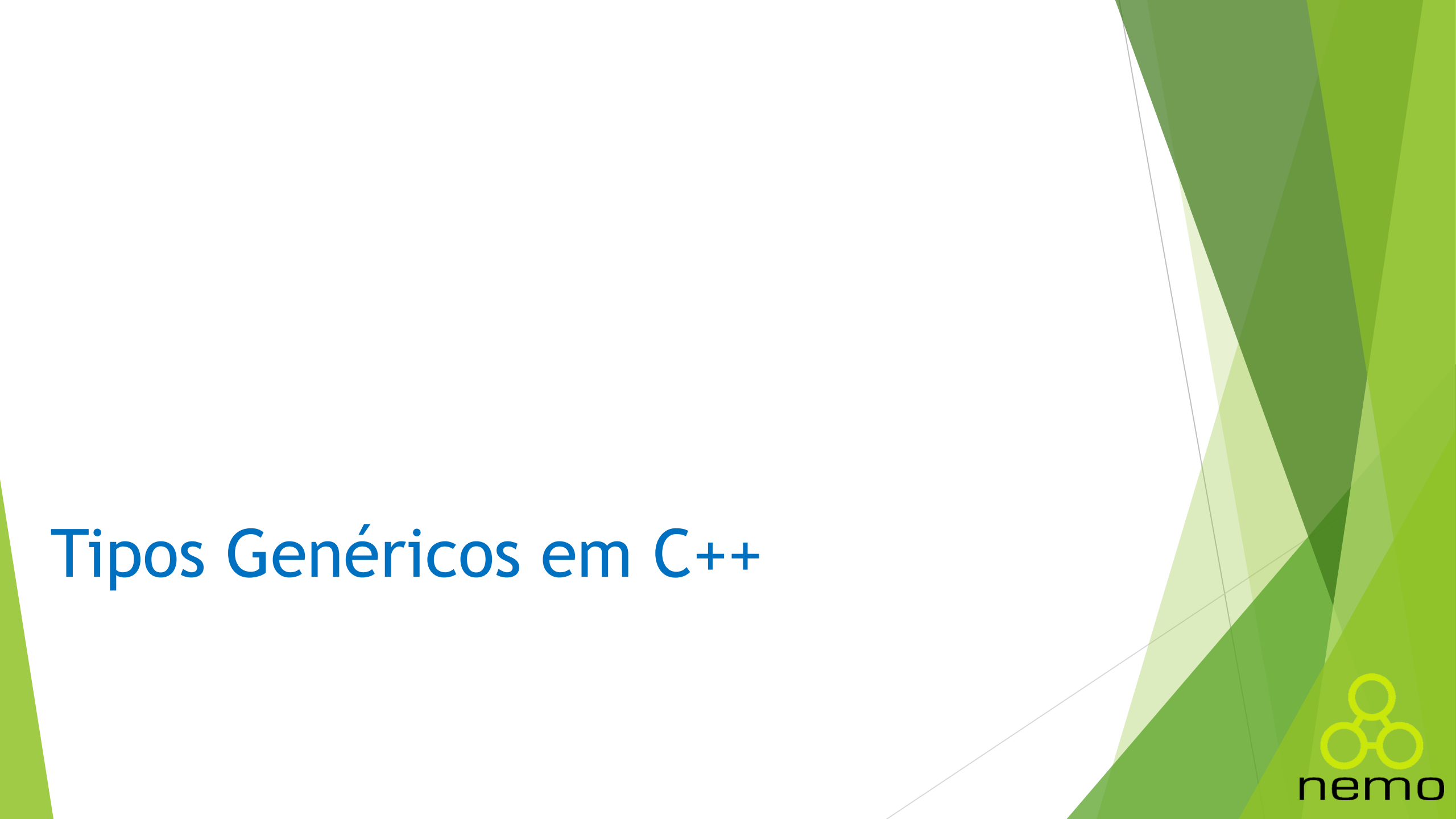
jordanasalamon@gmail.com

DEPARTAMENTO DE INFORMÁTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Tipos Genéricos em C++



Por que utilizar Tipos Genéricos?

- Suponhamos que temos uma função chamada *"compara_valores"*, que compara dois valores (recebidos por parâmetro) e retorna o maior valor.

```
compara_valores (a, b)
{
    se (a > b) retorna a;
    senão retorna b;
}
```

Tipo	Valor a	Valor b	Saída Esperada
Inteiro	15	20	20
Ponto Flutuante	10.55	10.85	10.85
String	"Aula"	"Prova"	"Prova"
Character	'a'	'c'	'c'

nemo

Por que utilizar Tipos Genéricos?

- Qual a solução devemos adotar nesse caso?
  - Construir uma função para cada tipo de valor?

```
int compara_valores (int a, int b)
{
    if(a > b) return a;
    else return b;
}
```

```
float compara_valores (float a, float b)
{
    if(a > b) return a;
    else return b;
}
```

```
char compara_valores (char a, char b)
{
    if(a > b) return a;
    else return b;
}
```

```
string compara_valores (string a, string b)
{
    if(a > b) return a;
    else return b;
}
```

- A linguagem permite sobrecarregarmos métodos, sendo possível assim criar métodos homônimos para manipular diferentes tipos de dados.

nemo

Por que utilizar Tipos Genéricos?

- Mas essa é a solução mais adequada?

```
int compara_valores (int a, int b)
{
    if(a > b) return a;
    else return b;
}
```

```
float compara_valores (float a, float b)
{
    if(a > b) return a;
    else return b;
}
```

- Observe como a implementação das funções são semelhantes, a única diferença é o tipo do parâmetro e o tipo do retorno que está sendo tratado.

nemo

Por que utilizar Tipos Genéricos?

- Neste exemplo a função possui apenas duas linha de processamento.

```
int compara_valores (int a, int b)
{
    if(a > b) return a;
    else return b;
}
```

```
float compara_valores (float a, float b)
{
    if(a > b) return a;
    else return b;
}
```

- Tipos em C++ não se restringe a apenas **tipos primitivos** mas também é possível definir novos tipos (e.g., classes, interfaces).

- Imagine criar funções com 100 linhas de processamento. E ser necessário implementar funções idênticas para os vários tipos possíveis.
  - Essa solução inicial começa a se tornar custosa.

nemo

Solução Ideal

- O ideal seria escrever apenas uma função genérica que seja independente de tipo.

```
tipo compara_valores (tipo a, tipo b)
{
    if(a > b) return a;
    else return b;
}
```

- Esse **tipo**, então, seria definido no momento em que a função fosse chamada.

- Na linguagem C++ denominamos esses tipos genéricos de **TEMPLATES.**

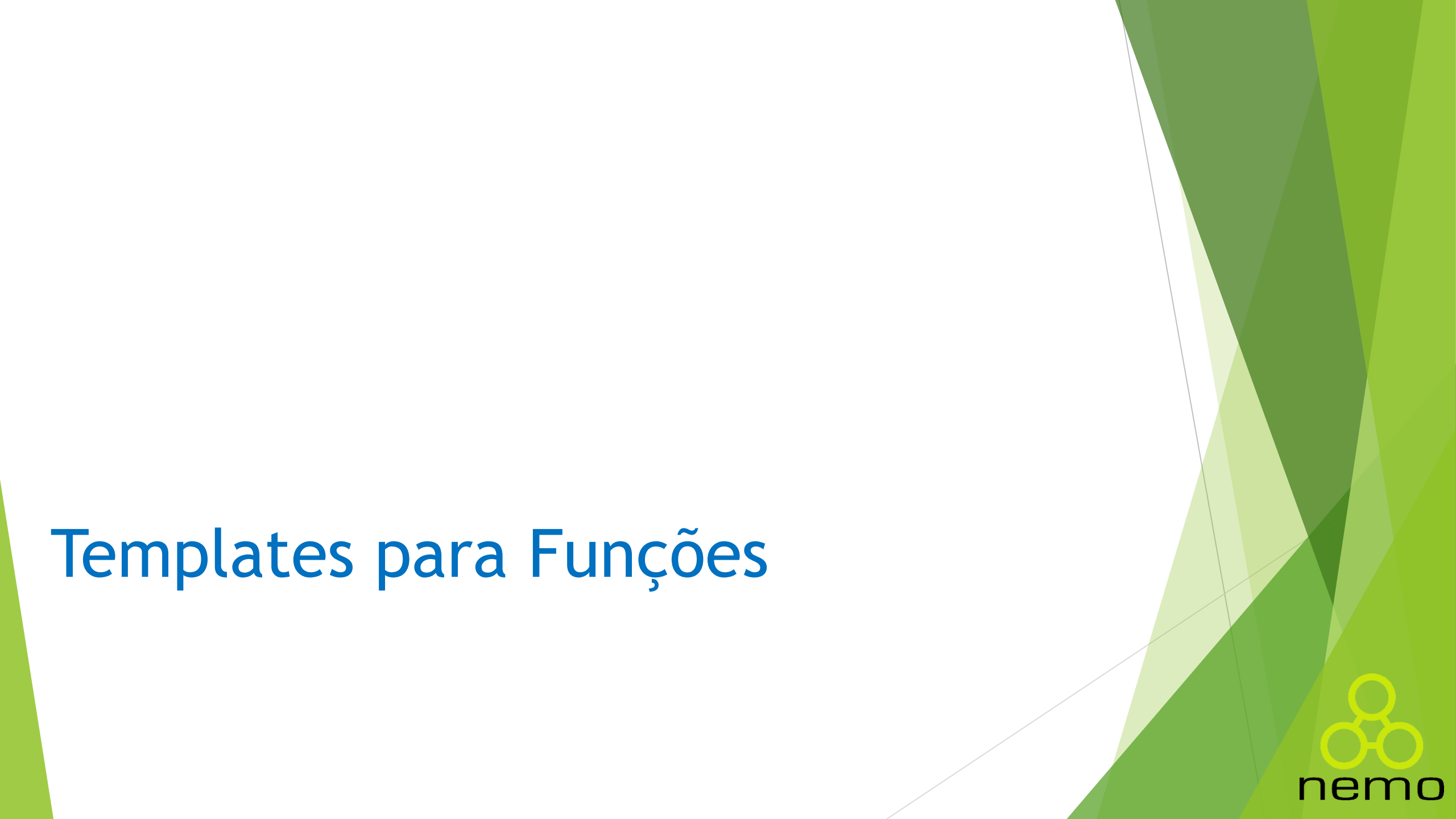
nemo

O que são Templates?

- São a fundamentação para programação genérica em C++.
- São baseados na ideia de **polimorfismo paramétrico**.
  - É uma forma de tornar a linguagem mais expressiva, permitindo escrever funções ou tipo de dados genericamente, suportando operações idênticas sem depender o tipo.
- Envolvem a escrita de código de uma forma independente de qualquer tipo específico.
- Eles são estruturas para a criação de **funções genéricas** e **classes genéricas**.

nemo

Templates para Funções



Templates de Funções

- Estrutura:

```
template <class/typename typel, ... typeN>
retorno nome_func (lista de parametros)
{
    // corpo da função
}
```

nemo

Templates de Funções

- Exemplo:

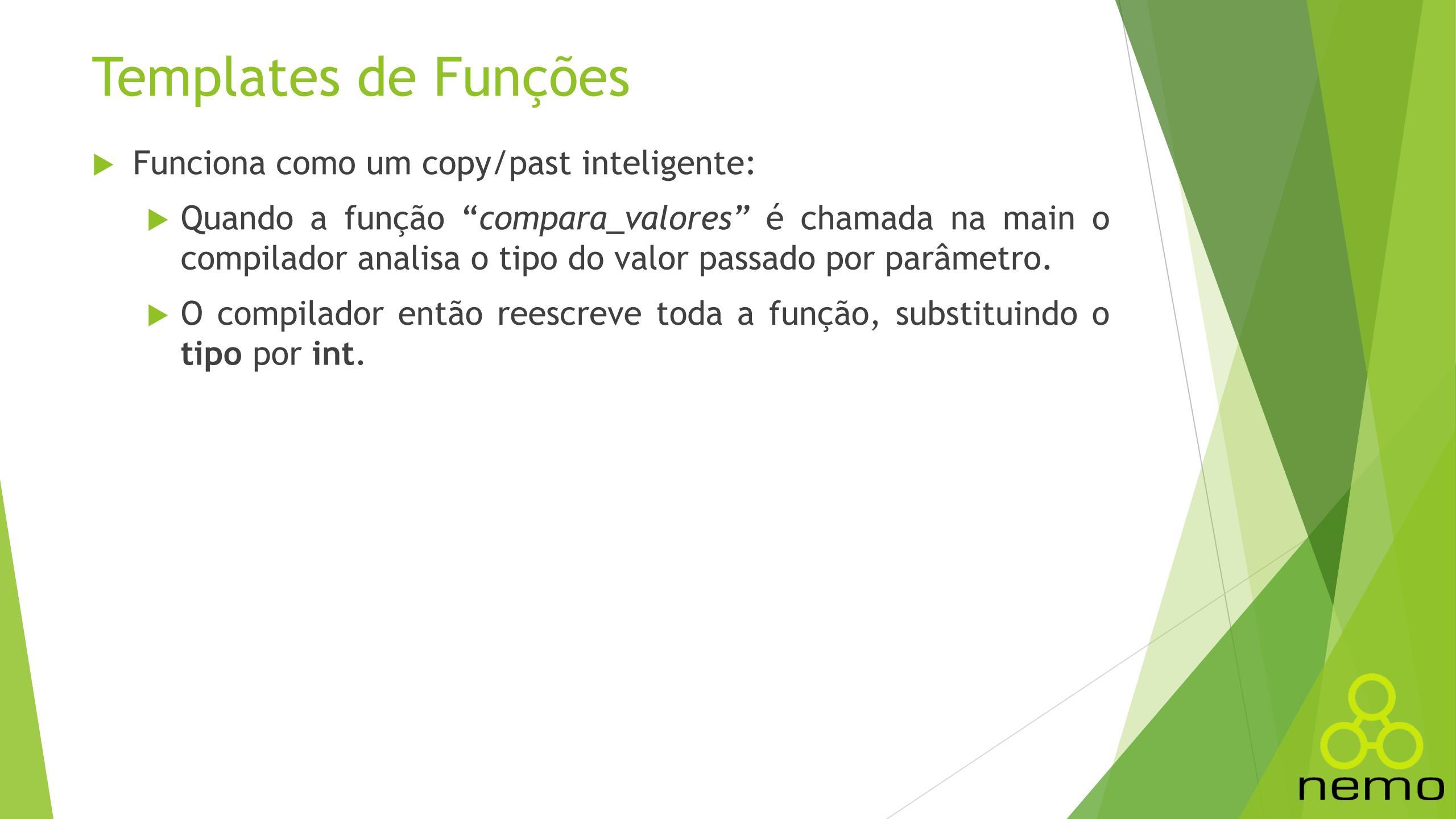
```
template <typename tipo>
tipo compara_valores (tipo a, tipo b){
    if(a > b) return a;
    else return b;
}

int main()
{
    int x = 15, y = 20;
    float z = 10.55, k = 10.85;
    string s1 = "Aula", s2 = "Prova";
    char c1 = 'a', c2 = 'c';

    cout << ">>---Comparacao---<<\n" << endl;
    cout << "Maior Int: " << compara_valores<int>(x,y) << endl;
    cout << "Maior Float: " << compara_valores<float>(z,k) << endl;
    cout << "Maior String: " << compara_valores<string>(s1,s2) << endl;
    cout << "Maior Char: " << compara_valores<char>(c1,c2) << endl;

    return 0;
}
```

```
>>---Comparação---<<

Maior Int: 20
Maior Float: 10.85
Maior String: Prova
Maior Char: c
```

nemo

Templates de Funções

- Funciona como um copy/past inteligente:
  - Quando a função “*compara_valores*” é chamada na main o compilador analisa o tipo do valor passado por parâmetro.
  - O compilador então reescreve toda a função, substituindo o **tipo** por **int**.



Templates de Funções

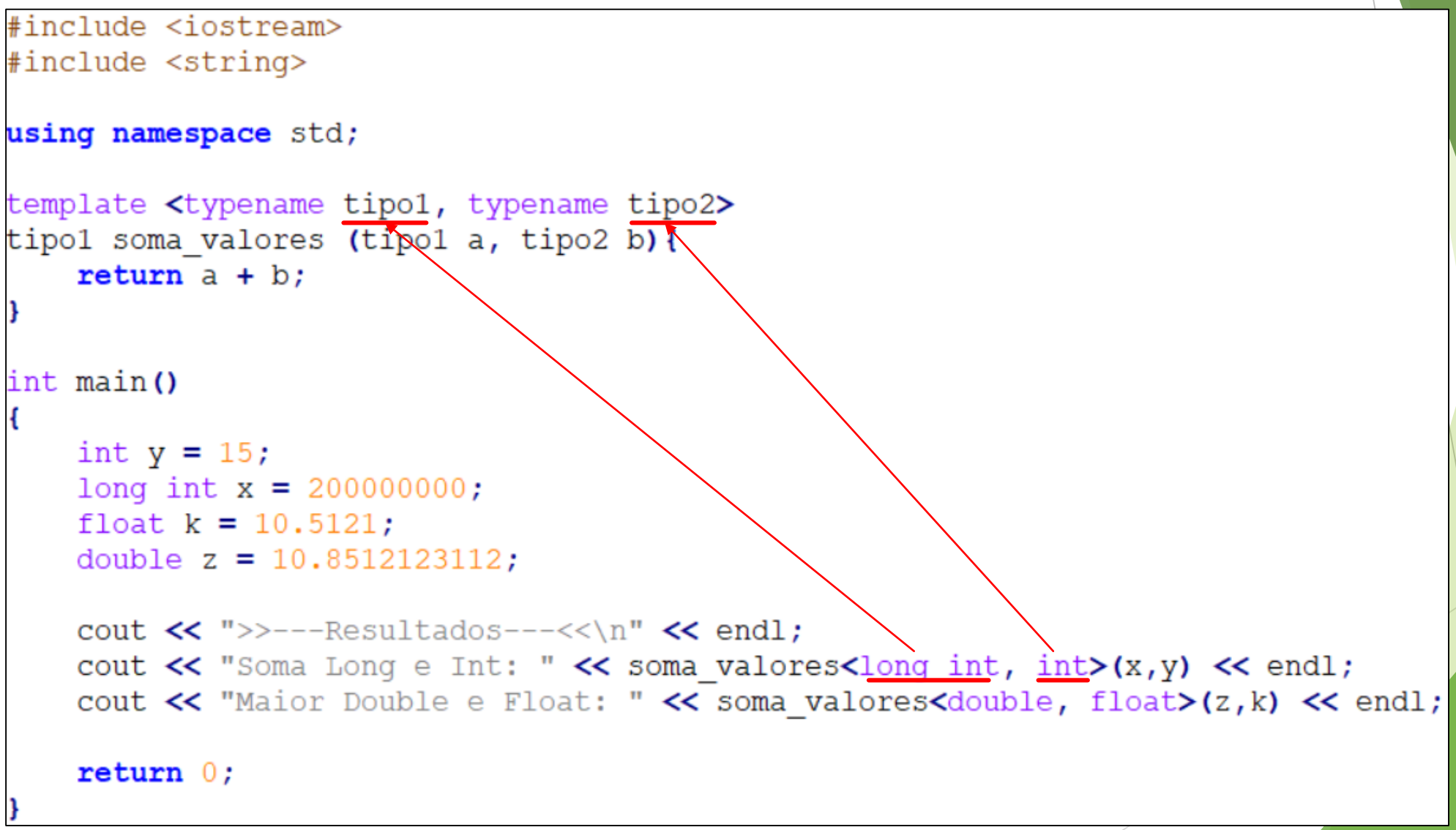
```
template <typename tipo>
tipo compara_valores (tipo a, tipo b){
    if(a > b) return a;
    else return b;
}

int main()
{
    int x = 15, y = 20;

    cout << "Maior Int: " << compara_valores<int>(x,y) << endl;

    return 0;
}
```

```
int compara_valores (int a, int b)
{
    if(a > b) return a;
    else return b;
}
```

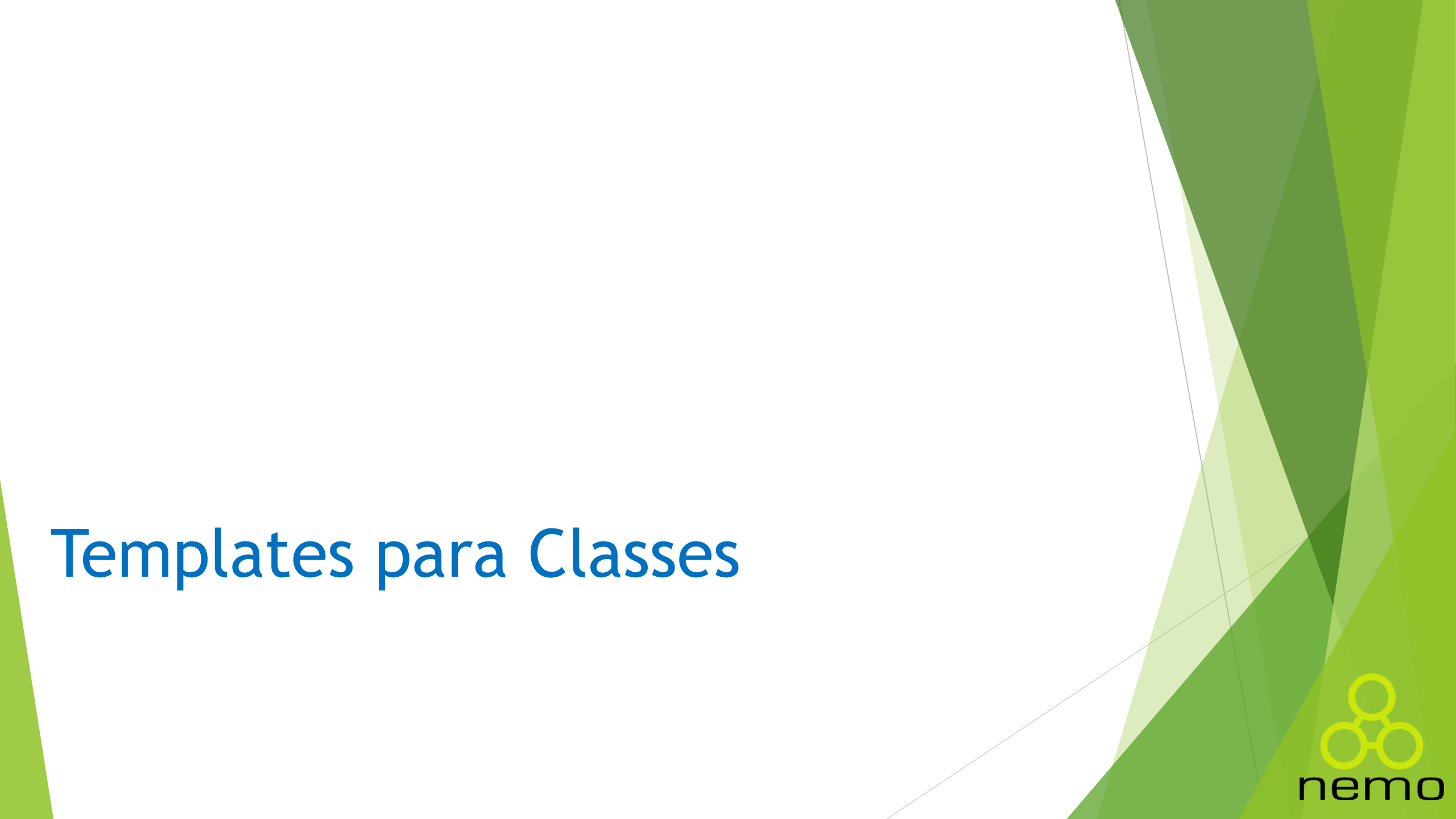
nemo

Templates de Mútiplos Tipos - Função

```cpp
#include <iostream>
#include <string>

using namespace std;

template <typename tipo1, typename tipo2>
tipo1 soma_valores (tipo1 a, tipo2 b){
    return a + b;
}

int main()
{
    int y = 15;
    long int x = 200000000;
    float k = 10.5121;
    double z = 10.8512123112;

    cout << ">>---Resultados---<<\n" << endl;
    cout << "Soma Long e Int: " << soma_valores<long int, int>(x,y) << endl;
    cout << "Maior Double e Float: " << soma_valores<double, float>(z,k) << endl;

    return 0;
}
```



Templates para Classes



Templates de Classes

- Estrutura:

```
template <class/typename typeI, ... typeN>
class nome_classe
{
    // atributos e metodos da classe
}
```

nemo

Templates de Classes

- Exemplo:

```
class Pessoa{
    public:
        int idade;
        string nome;
        double altura;
        double peso;
};

class Cachorro{
    public:
        int idade;
        string nome;
        double peso;
};
```

```
template <typename tipo>
class Coletivo
{
public:
    vector<tipo> membros;

    void push (tipo membro){
        membros.push_back(membro);
    }
};
```

```
int main ()
{
    Pessoa *p1;
    Cachorro *c1;
    p1 = new Pessoa();
    c1 = new Cachorro();

    p1->idade = 25;
    p1->nome = "Joãozinho";
    p1->altura = 1.8;
    p1->peso = 85.5;

    c1->idade = 15;
    c1->nome = "Rex";
    c1->peso = 5.65;

    Coletivo<Cachorro*> cachorros;
    Coletivo<Pessoa*> pessoas;
    cachorros.push(c1);
    pessoas.push(p1);

    return 0;
}
```

Implementado Método Fora da Classe

- Exemplo:

```
class Pessoa{
    public:
        int idade;
        string nome;
        double altura;
        double peso;
};

class Cachorro{
    public:
        int idade;
        string nome;
        double peso;
};
```

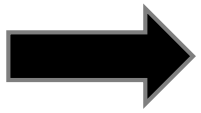
```
template <typename tipo>
class Coletivo
{
    public:
        vector<tipo> membros;

        void push(tipo membro);
};

template <typename tipo>
void Coletivo<tipo>::push (tipo membro){
    membros.push_back(membro);
}
```

```
int main ()
{
    Pessoa *p1;
    Cachorro *c1;
    p1 = new Pessoa();
    c1 = new Cachorro();

    p1->idade = 25;
    p1->nome = "Joãozinho";
    p1->altura = 1.8;
    p1->peso = 85.5;

    c1->idade = 15;
    c1->nome = "Rex";
    c1->peso = 5.65;

    Coletivo<Cachorro*> cachorros;
    Coletivo<Pessoa*> pessoas;
    cachorros.push(c1);
    pessoas.push(p1);

    return 0;
}
```

Templates de Classes

- Uma classe template implementa o conceito de uma classe genérica.
- Uma classe que pode ser construída para mais de um tipo, mas que tem a mesma forma (estrutura).

```
template <typename tipo>
class Coletivo
{
    public:
        vector<tipo> membros;

        void push(tipo membro);
};

template <typename tipo>
void Coletivo<tipo>::push (tipo membro){
    membros.push_back(membro);
}

int main ()
{
    Pessoa p1;
    Coletivo<Pessoa*> pessoas;
    pessoas.push(p1);
}
```



```
template <typename Pessoa*>
class Coletivo
{
    public:
        vector<Pessoa*> membros;

        void push(Pessoa* membro);
};

template <typename Pessoa*>
void Coletivo<Pessoa*>::push (Pessoa membro){
    membros.push_back(membro);
}
```

nemo

Templates de Mútiplos Tipos - Classes

```
template <typename tipo1, typename tipo2>
class Coletivo
{
    public:
        vector<tipo1> membros;
        tipo2 identificador;

    void push(tipo1 membro);
};

template <typename tipo1, typename tipo2>
void Coletivo<tipo1, tipo2>::push (tipo1 membro){
    membros.push_back(membro);
}
```

```
int main ()
{
    Pessoa p1;
    Cachorro c1;

    Coletivo<Cachorro*, string> cachorros;
    Coletivo<Pessoa*, int> pessoas;
    cachorros.identificador = "C1";
    pessoas.identificador = 1;

    cachorros.push(c1);
    pessoas.push(p1);

    return 0;
}
```

nemo

Exemplo Completo de Templates para Classes

That’s all Folks!