Recursos Avançados de C++

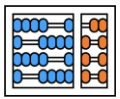
Módulo 3

Prof. Dr. Bruno B. P. Cafeo

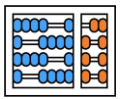
Instituto de Computação
Universidade Estadual de Campinas

Agenda

- Programação genérica
- Templates
- Smart pointers
- Semântica de transferência



Programação genérica

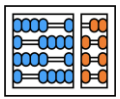


UNICAMP

Programação genérica

- Estilo de programação no qual algoritmos são escritos em termos de **tipos de dados a serem especificados posteriormente** e que são então instanciados quando necessários para tipos específicos fornecidos como parâmetros

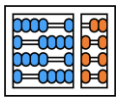
- A programação genérica desempenha um papel fundamental em linguagens de programação **tipadas**, fornecendo segurança de tipos e reutilização de código



Linguagens tipadas

- Linguagens tipadas são aquelas que realizam verificação de tipos em tempo de compilação, garantindo que as operações sejam aplicadas apenas a tipos de dados apropriados.

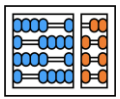
- Exemplos: Alguns exemplos de linguagens tipadas incluem C++, Java, C# e Rust, todas as quais enfatizam a segurança de tipos.



Segurança do tipos com linguagens tipadas

- **Verificação de Tipos Estática:** O tipo de cada variável, parâmetro e expressão é definido e verificado em tempo de compilação. Isso significa que o compilador avalia se as operações realizadas em variáveis são compatíveis com seus tipos. Se houver uma incompatibilidade de tipos, o compilador emitirá um erro e não permitirá a compilação do código. Isso evita que erros de tipo, como tentar somar uma string com um número, passem despercebidos até o tempo de execução.

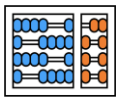
- **Eliminação de Erros de Conversão:** É necessário fazer conversões explícitas de tipo quando se deseja operar com tipos diferentes. Isso ajuda a evitar erros de conversão, garantindo que os tipos sejam compatíveis antes que as operações ocorram. Se uma conversão não for válida, o compilador detectará o erro e impedirá a compilação.



Segurança do tipos com linguagens tipadas

- **Restrições de Tipo:** Algumas linguagens tipadas permitem a definição de restrições de tipo específicas. Por exemplo, você pode criar tipos personalizados que só podem ser usados de maneiras específicas. Isso garante que o código seja usado de acordo com as regras definidas, evitando erros.

- **Polimorfismo Controlado:** O polimorfismo, que permite que um único código seja aplicado a diferentes tipos de dados, é controlado de forma segura em linguagens tipadas. O compilador garante que as operações polimórficas sejam realizadas apenas em tipos compatíveis.

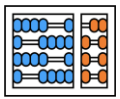


“Problema”

- Imagine que você está desenvolvendo um sistema de processamento de dados em uma linguagem tipada e precisa criar funções para realizar operações de agregação em diferentes tipos de dados: números inteiros, números de ponto flutuante e strings.

- Sem programação genérica, você teria que criar funções de agregação separadas para cada tipo de dado, o que resultaria em duplicação de código e aumento da complexidade.

- Com a programação genérica, você pode criar funções genéricas de agregação que funcionam para qualquer tipo de dado, tornando o código mais eficiente e fácil de manter.

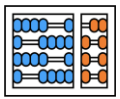


Por Que Programação Genérica em Linguagens Tipadas?

- **Segurança de Tipos:** Nas linguagens tipadas, a segurança de tipos é essencial para evitar erros de tempo de execução, permitindo a detecção de erros em tempo de compilação.

- **Reutilização de Código:** A programação genérica permite a criação de algoritmos e estruturas de dados que podem ser usados com vários tipos de dados, promovendo a reutilização do código.

- **Abstração:** A abstração é facilitada pela programação genérica, permitindo o encapsulamento de algoritmos e estruturas de dados em componentes independentes de tipos.

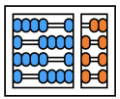


Exemplos em Linguagens Tipadas

- **Contêineres Genéricos**: Linguagens tipadas frequentemente implementam contêineres genéricos, como listas e dicionários, que podem armazenar uma variedade de tipos de dados.

- **Algoritmos Genéricos:** Algoritmos genéricos podem ser aplicados a diferentes tipos de coleções de dados, economizando tempo e esforço de desenvolvimento.

- **Bibliotecas e Estruturas:** Muitas bibliotecas e frameworks em linguagens tipadas fazem uso extensivo da programação genérica para fornecer funcionalidades amplamente aplicáveis.

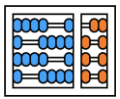


Desafios em Linguagens Tipadas

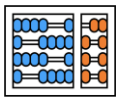
- **Overhead**: A verificação de tipos em linguagens tipadas pode introduzir um leve desempenho reduzido devido à necessidade de verificação adicional.

- **Complexidade:** O uso de genéricos pode tornar o código mais complexo, especialmente quando há restrições de tipos.

- **Equilíbrio**: Encontrar o equilíbrio certo entre segurança de tipos e flexibilidade de código é um desafio importante.



Templates

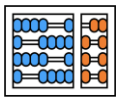


UNICAMP

O que são Templates em C++

- Templates habilitam a programação genérica em C++.

- É possível escrever algoritmos genéricos que funcionam com diferentes tipos de dados.

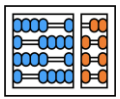


Sintaxe de Templates

- Classes, funções [e variáveis (a partir do C++14)]

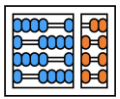
```
template<typename T>
```

```
template<class T>
```



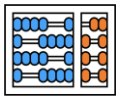
Exemplo (Método)

```
1   #include <iostream>
2
3   template <typename T>
4   T soma(T a, T b) {
5       return a + b;
6   }
7
8   int main() {
9       int resultadoInteiro = soma(5, 7);
10      double resultadoDouble = soma(3.14, 2.0);
11
12      std::cout << "Soma de inteiros: " << resultadoInteiro << std::endl;
13      std::cout << "Soma de doubles: " << resultadoDouble << std::endl;
14
15      return 0;
16  }
17
```



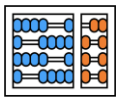
Exemplo (Classe)

```cpp
#include <iostream>
#include <vector>

template <typename T>
class Pilha {
public:
    Pilha() {}

    void empilhar(const T& elemento) {
        elementos.push_back(elemento);
    }

    void desempilhar() {
        if (!vazia()) {
            elementos.pop_back();
        }
    }

    T topo() {
        if (!vazia()) {
            return elementos.back();
        }
        throw std::runtime_error("A pilha está vazia.");
    }

private:
    std::vector<T> elementos;
};

int main() {
    Pilha<int> pilhaInt;
    pilhaInt.empilhar(5);
    pilhaInt.empilhar(10);
    pilhaInt.desempilhar();
    std::cout << "Topo da pilha de inteiros: " << pilhaInt.topo() << std::endl;

    Pilha<std::string> pilhaString;
    pilhaString.empilhar("Olá");
    pilhaString.empilhar("Mundo");

    return 0;
}
```



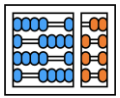
Exemplo (Variável)

```cpp
#include <iostream>

template <class T>
constexpr T pi = T(3.14159265358979323851L); // Variável template

template <class T>
T circular_area(T r) // Função template
{
    return pi<T> * r * r; // pi<T> é uma variável template
}

int main() {
    // Use a função template circular_area para calcular a área de um círculo
    double raio = 5.0;
    double area = circular_area(raio);
    std::cout << "Área do círculo: " << area << std::endl;

    // Use o valor de pi do tipo float
    float pi_float = pi<float>;
    std::cout << "Valor de pi (float): " << pi_float << std::endl;

    // Use o valor de pi do tipo double
    double pi_double = pi<double>;
    std::cout << "Valor de pi (double): " << pi_double << std::endl;

    return 0;
}
```



Dois “tipos” em template

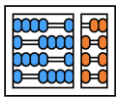
```
1   template <typename T, typename U>
2   void funcaoGenerica(T valor1, U valor2) {
3       std::cout << "Valor 1: " << valor1 << std::endl;
4       std::cout << "Valor 2: " << valor2 << std::endl;
5   }
6
7   int main() {
8       funcaoGenerica(42, 3.14);
9       funcaoGenerica("Olá, ", "mundo!");
10      return 0;
11  }
12
```



Como testar o tipo do template?

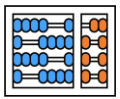
- Parte da biblioteca de metaprogramação

- A biblioteca <type_traits> (a partir do C++11) fornece utilitários para testar tipos
  - Tipos primários: is_void, is_integral, is_floating_point, is_class, is_function, ...
  - Tipos compostos (composite): is_scalar, is_object, is_reference, is_member_pointer,...
  - Propriedades de tipos: is_const, is_empty, is_polymorphic, ...
  - Tipos de relacionamento: is_same, is_base_of, ...



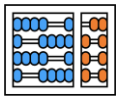
Como testar o tipo do template?

```cpp
#include <type_traits>

template <typename T>
void funcao(T valor) {
    if (std::is_integral<T>::value) {
        std::cout << "É um tipo inteiro." << std::endl;
    } else {
        std::cout << "Não é um tipo inteiro." << std::endl;
    }
}

int main() {
    funcao(42);      // Saída: É um tipo inteiro.
    funcao(3.14);    // Saída: Não é um tipo inteiro.
    return 0;
}
```

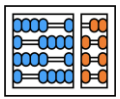


SFINAE (**S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror)

```
1   template <typename T>
2   typename std::enable_if<std::is_integral<T>::value, T>::type
3   funcao(T valor) {
4       return valor * 2;
5   }
6
7   template <typename T>
8   typename std::enable_if<!std::is_integral<T>::value, T>::type
9   funcao(T valor) {
10      return valor + 2.0;
11  }
12
13  int main() {
14      int x = funcao(42);     // Multiplica por 2
15      double y = funcao(3.14);  // Adiciona 2.0
16      return 0;
17  }
18
```



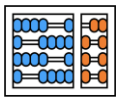
Semântica de Transferência



UNICAMP

O que é?

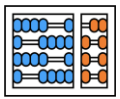
- A Semântica de Transferência é um conceito fundamental na programação em C++.
- Define como objetos são transferidos, copiados ou movidos.
- Embora seja uma técnica poderosa, a Semântica de Transferência pode ser complexa e desafiadora de dominar.



Por que?

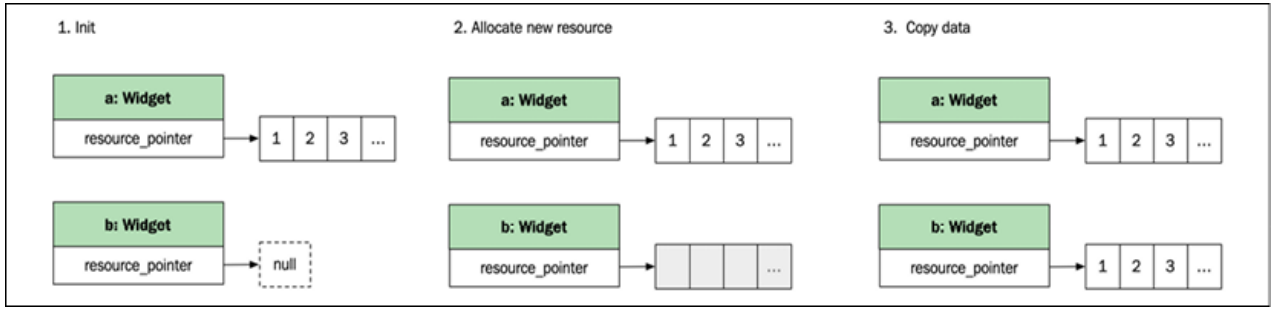
- C++ é uma linguagem baseada em valores

- Tipos de referências devem ser acessados explicitamente (ponteiros)



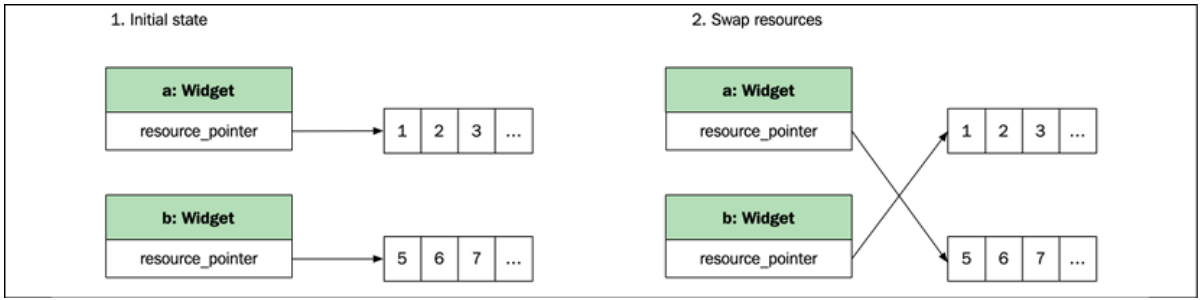
Por que? (Cópia)

```
1  auto a = Widget{};
2  auto b = a;          // Copy-construction
3
```



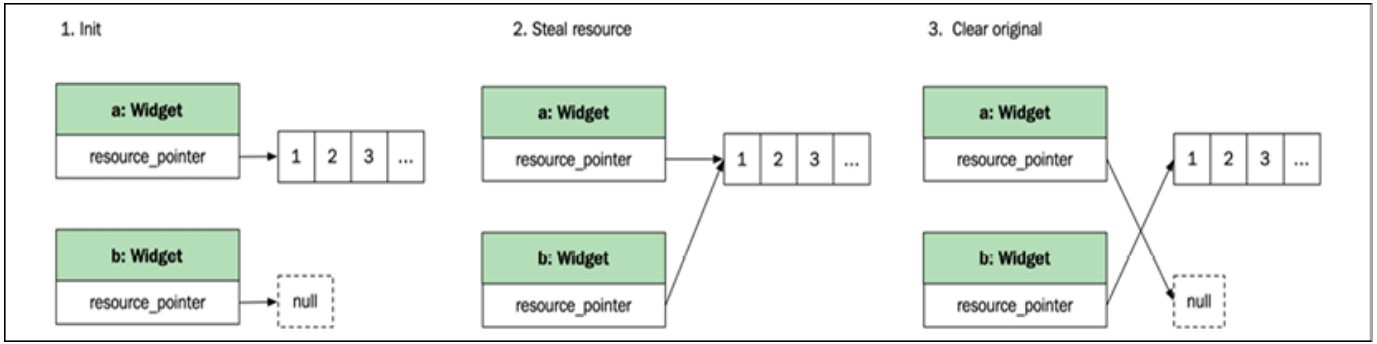
Por que? (Troca)

```
1  auto a = Widget{};
2  auto b = Widget{};
3  std::swap(a, b);
4
```



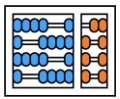
Por que? (Movimento)

```
1  auto a = Widget{};
2  auto b = std::move(a); // Tell the compiler to move the resource into b
3
```



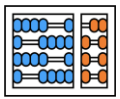
RAII (**R**esource **A**cquisition **I**s **I**nitialization)

- **Princípio Fundamental:** Os recursos (memória, arquivos, sockets, etc.) devem ser adquiridos durante a inicialização de um objeto.
- **Liberação Automática:** A liberação dos recursos é realizada automaticamente quando o objeto sai de escopo.
- **Garante Gerenciamento Seguro de Recursos:** Evita vazamentos de recursos e torna o código mais robusto.



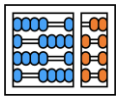
Regra dos três

- Copy constructor
- Copy assignment
- Destructor



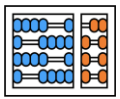
Regra dos três

```
1  class Buffer {
2  public:
3    // Constructor
4    Buffer(const std::initializer_list<float>& values): size_{values.size()} {
5      ptr_ = new float[values.size()];
6      std::copy(values.begin(), values.end(), ptr_);
7    }
8    auto begin() const { return ptr_; }
9    auto end() const { return ptr_ + size_; }
10
11   // OS PRÓXIMOS CÓDIGOS DEVEM VIR AQUI!!
12
13 private:
14   size_t size_{0};
15   float* ptr_{nullptr};
16 };
17
```



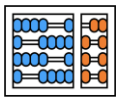
Regra dos três

```
1   // 1. Copy constructor
2   Buffer::Buffer(const Buffer& other) : size_{other.size_} {
3     ptr_ = new float[size_];
4     std::copy(other.ptr_, other.ptr_ + size_, ptr_);
5   }
6   // 2. Copy assignment
7   auto& Buffer::operator=(const Buffer& other) {
8     delete [] ptr_;
9     ptr_ = new float[other.size_];
10    size_ = other.size_;
11    std::copy(other.ptr_, other.ptr_ + size_, ptr_);
12    return *this;
13  }
14  // 3. Destructor
15  Buffer::~Buffer() {
16    delete [] ptr_;
17    ptr_ = nullptr;
18  }
19
```



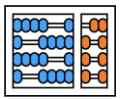
Regra dos três

```
1  int main() {
2      // Exemplo de construção de um buffer
3      Buffer buffer1(5);
4      for (int i = 0; i < 5; ++i) {
5          buffer1.ptr_[i] = static_cast<float>(i);
6      }
7
8      // Exemplo de copy constructor
9      Buffer buffer2 = buffer1; //Realizando a cópia
10     std::cout << "Buffer2: ";
11     buffer2.print();
12
13     // Exemplo de copy assignment
14     Buffer buffer3(3);
15     buffer3 = buffer1; //Realizando a atribuição
16     std::cout << "Buffer3: ";
17     buffer3.print();
18
19     return 0;
20 } // Com a saída do escopo, o destructor é automaticamente invocado
21
```



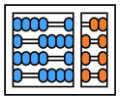
Regra dos **cinco** (Move semantics – A partir do C++11)

- Move constructor
- Move assignment



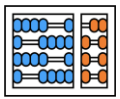
Regra dos **cinco** (Move semantics – A partir do C++11)

```
1   // 4. Move constructor
2   Buffer::Buffer(Buffer&& other) noexcept: size_{other.size_}, ptr_{other.ptr_} {
3     other.ptr_ = nullptr;
4     other.size_ = 0;
5   }
6   // 5. Move assignment
7   auto& Buffer::operator=(Buffer&& other) noexcept {
8     ptr_ = other.ptr_;
9     size_ = other.size_;
10    other.ptr_ = nullptr;
11    other.size_ = 0;
12    return *this;
13  }
14
```



Regra dos **cinco** (Move semantics – A partir do C++11)

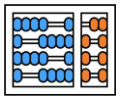
```
1   Buffer createBuffer() {
2       Buffer buffer(5);
3       return static_cast<Buffer&&>(buffer); // Aqui ocorre o retorno forçando o move constructor
4   }
5
6   int main() {
7       Buffer result = createBuffer();
8
9       /* Neste ponto, o retorno automático de mover otimiza a transferência de recursos
10      da função createBuffer() para a variável result. */
11
12      return 0;
13  }
14
```



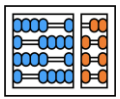
UNICAMP

Regra dos **cinco** (Move semantics – A partir do C++11)

```
1   Buffer createBuffer() {
2       Buffer buffer(5);
3       return buffer; // Aqui ocorre o retorno que usará implicitamente o move constructor
4   }
5
6   int main() {
7       Buffer result = createBuffer();
8
9       /* Neste ponto, o retorno automático de mover otimiza a transferência de recursos
10      da função createBuffer() para a variável result. */
11
12      return 0;
13  }
14
```



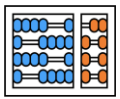
Smart Pointers



UNICAMP

Motivando

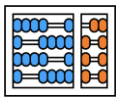
```
1   #include <iostream>
2   using namespace std;
3
4   class Rectangle {
5   private:
6       int length;
7       int breadth;
8   };
9
10  void foo()
11  {
12      Rectangle* p = new Rectangle();
13  }
14
15  int main()
16  {
17      while (1) {
18          foo();
19      }
20  }
21
```



Motivando

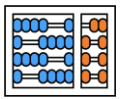
```
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle {
5  private:
6      int length;
7      int breadth;
8  };
9
12     Rectangle* p = new Rectangle();
13 }
14
15 int main()
16 {
17     while (1) {
18         foo();
19     }
20 }
21
```

Memory limit exceeded



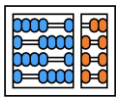
O que são Smart Pointers

- Objetos que gerenciam automaticamente a alocação e liberação de recursos.
- Prevenção de vazamentos de memória, simplificação da gestão de recursos.
- É um wrapper sobre um ponteiro com operadores * e -> sobrecarregados.
- Os objetos da classe de ponteiro inteligente se parecem com ponteiros convencionais. No entanto, ao contrário dos ponteiros convencionais, ele pode desalocar e liberar a memória do objeto destruído.



Tipos de Smart Pointers

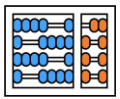
- auto_ptr (Depreciado a partir do C++11)
- unique_ptr
- shared_ptr
- weak_ptr



unique_ptr

- Tipo de smart pointer que gerencia recursos exclusivos.

```cpp
#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
    unique_ptr<Rectangle> P1(new Rectangle(10, 5));

    unique_ptr<Rectangle> P2 = std::move(P1);

    return 0;
}
```

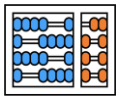


unique_ptr

- Tipo de smart pointer que gerencia recursos exclusivos.

```cpp
#include <iostream>
using namespace std;
// Dynamic Memory management library
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }

    int area() { return length * breadth; }
};

int main()
{
    unique_ptr<Rectangle> P1(new Rectangle(10, 5));

    unique_ptr<Rectangle> P2 = std::move(P1);

    return 0;
}
```

Semântica de transferência

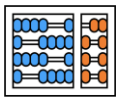


UNICAMP

shared_ptr

- Permite que vários smart pointers compartilhem a propriedade de um recurso.

```cpp
#include <iostream>
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b) : length(l), breadth(b) {}

    int area() { return length * breadth; }
};

int main() {
    std::shared_ptr<Rectangle> P1(new Rectangle(10, 5));

    std::shared_ptr<Rectangle> P2 = P1; // Compartilhando a propriedade com P1

    std::cout << "Área do retângulo em P1: " << P1->area() << std::endl;
    std::cout << "Área do retângulo em P2: " << P2->area() << std::endl;

    return 0;
}
```



shared_ptr

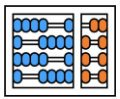
- Permite que vários smart pointers compartilhem a propriedade de um recurso.

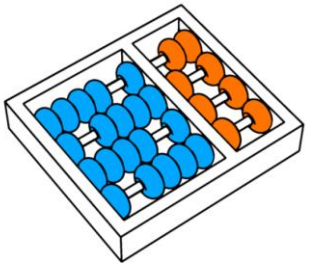
```cpp
#include <iostream>
#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b) : length(l), breadth(b) {}

    int area() { return length * breadth; }
};

int main() {
    std::shared_ptr<Rectangle> P1(new Rectangle(10, 5));

    std::shared_ptr<Rectangle> P2 = P1; // Compartilhando a propriedade com P1

    std::cout << "Área do retângulo em P1: " << P1->area() << std::endl;
    std::cout << "Área do retângulo em P2: " << P2->area() << std::endl;

    return 0;
}
```

Copy constructor

weak_ptr

- Mesma ideia do shared_ptr
- Não possui propriedade sobre o recurso
- Caso não existam proprietários do recurso referenciado por um weak_ptr, o recurso pode ser desalocado da memória





INSTITUTO DE COMPUTAÇÃO

UNICAMP

Prof. Dr. Bruno B. P. Cafeo

Sala 04
Instituto de Computação - Unicamp
Av. Albert Einstein, 1251
Cidade Universitária
Campinas – SP
13083-852

https://ic.unicamp.br/~cafeo/
cafeo@ic.unicamp.br