# Template

Roland Teodorowitsch

Programação Orientada a Objetos - ECo - Curso de Engenharia de Computação - PUCRS

5 de junho de 2024

# Programação Genérica

# Programação Genérica

- Os **genéricos** (ou ***templates***) são uma das mais poderosas maneiras de reuso de *software*
  - **Funções genéricas** e **classes genéricas** permitem que o programador especifique com apenas um segmento de código uma família de funções ou classes relacionadas (sobrecarregadas)
  - Esta tecnica e chamada **programação genérica**

# Programação Genérica

- Por exemplo, é possível criar uma **função genérica** que ordene um vetor
  - A linguagem se encarrega de criar especializações que tratarão vetores do tipo `int`, `float`, `string`, etc.
- É possível tambem criar uma classe genérica para a estrutura de dados Pilha
  - A linguagem se encarrega de criar as especializações para pilha de `int`, `float`, `string`, etc.
- O **genérico** define o formato, a **especialização** e conteúdo

# Funções Genéricas

# Funções Genéricas

- Funções sobrecarregadas normalmente realizam operações similares ou idênticas em diferentes tipos de dados
  - Soma de `int` e `float`
  - Se as operações são idênticas para diferentes tipos, elas podem ser expressas mais compacta e convenientemente atraves de **funções genéricas**
- O programador escreve a definição da função genérica
  - Baseado nos parâmetros explicitamente enviados ou inferidos a partir da chamada da função, o compilador gera as especializações para cada tipo de chamada

# Funções Genéricas

- Uma definição de **função genérica** começa com a palavra `template` seguida de uma lista de parâmetros genéricos entre < e >
- Cada parâmetro deve ser precedido por `class` ou `typename`
  - Especificam que os parâmetros serão de qualquer tipo primitivo
- Exemplos:

```
template <typename T>
template <class TipoElemento>
template <typename TipoBorda, typename TipoPreenchimento>
```

# Exemplo 1: definição

```
#include <iostream>

using namespace std;

// Definicao da template de funcao printArray
template <typename T>
void printArray(const T *array, int count) {
  for (int i=0; i<count; i++) {
      if ( i > 0 ) cout << " ";
      cout << array[i];
  }
  cout << endl;
}
// fim do template de funcao printArray

/*
  O template acima eh equivalente a ter 3 versoes de funcoes (uma para cada tipo):
  * void printArray(const int *, int);
  * void printArray(const double *, int);
  * void printArray(const char *, int);
*/
```

# Exemplo 1: uso e resultado

```
int main() {
  const int TAM_A = 5; // tamanho do array a
  const int TAM_B = 7; // tamanho do array b
  const int TAM_C = 6; // tamanho do array c
  int    a[TAM_A] = {1,2,3,4,5};
  double b[TAM_B] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7};
  char   c[TAM_C] = {'H','E','L','L','O','!'};

  cout << "O vetor a contem:" << endl;
  // chama a especializacao da template de funcao do tipo inteiro
  printArray(a,TAM_A);

  cout << "O vetor b contem:" << endl;
  // chama a especializacao da template de funcao do tipo double
  printArray(b,TAM_B);

  cout << "O vetor c contem:" << endl;
  // chama a especializacao da template de funcao do tipo caractere
  printArray(c,TAM_C);

  return 0;
}
```

```
O vetor a contem:
1 2 3 4 5
O vetor b contem:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
O vetor c contem:
H E L L O !
```

# Funções Genéricas

- Quando o compilador detecta a chamada a `printArray()`, ele procura a definição da função
  - Neste caso, a função genérica
  - Ao comparar os tipos dos parâmetros, nota que há um tipo genérico
  - Então, deduz qual deverá ser a substituição a ser feita:
    - `T` por `int`
  - O compilador cria três especializações:
    - `void printArray(const int *, int);`
    - `void printArray(const double *, int);`
    - `void printArray(const char *, int);`

# Classes Genéricas

# Classes Genéricas

- Para se compreender o funcionamento da estrutura de dados `Pilha`, não importa o tipo dos dados empilhados/desempilhados
  - No entanto, para implementar uma pilha, é necessário associá-la a um tipo
- Esta é uma das grandes oportunidade de **reuso de *software***
- O ideal é descrever uma pilha genéricamente, assim como ela é entendida
- Instanciar versões específicas desta **classe genérica** fica por conta do compilador

# Classes Genéricas

- **Classes Genéricas** (ou **Tipos Parametrizados**) requerem um ou mais parâmetros de tipo que especifiquem como customizá-las
  - Gerando assim **especializações de classes genéricas**
- O programador codifica apenas a definição da classe genérica
  - A cada vez que uma especialização diferente for necessária, o compilador codifica a especialização
  - Uma classe genérica `Pilha` vira uma coleção de classes especializadas:
    - Pilha de `int`, `float`, `string`, frações, restaurantes, etc.

# Classes Genéricas

- A definição de uma **classe genérica** é semelhante à definição de uma classe normal
  - Antes da definição da classe, adiciona-se o cabeçalho:
    `template <typename T>`
  - Em que o `T` representa o tipo que a classe manipula, passado como um parâmetro
    - Na verdade, qualquer identificador serve, mas `T` é padrão
  - Em caso de tipos definidos pelo programador, deve-se tomar cuidados com a sobrecarga de operadores e também garantir a existência de pelo menos um construtor (o *default*)

# Exemplo 2: definição

```
#include <iostream>

using namespace std;

template <typename T>
class Stack {
private:
  int size; // numero de elementos na Stack
  int top; // localizacao do elemento superior (-1 significa vazio)
  T *stackPtr; // ponteiro para a representacao interna da Stack
public:
  Stack(int = 10); // construtor padrao (tamanho de Stack 10)
  bool push(const T&); // insere (push) um elemento na Stack
  bool pop(T&); // remove (pop) um elemento da Stack

  ~Stack() {    // destrutor
    delete [] stackPtr; // desaloca o espaco interno para Stack
  }

  bool isEmpty() const {    // determina se a Stack esta vazia
    return top == -1;
  }

  bool isFull() const {    // determina se Stack esta cheia
    return top == size - 1;
  }
};
```

# Exemplo 2: implementação

```
// template construtora
template <typename T>
Stack<T>::Stack(int s):size((s>0)?s:10),       // valida o tamanho
                       top(-1),                // Stack inicialmente vazia
                       stackPtr(new T[size]) { // aloca memoria para elementos
  // corpo vazio
}

// insere elemento na Stack;
template <typename T>
bool Stack<T>::push(const T &pushValue) {
  if ( !isFull() ) {
     stackPtr[ ++top ] = pushValue; // insere item em Stack
     return true; // insercao bem-sucedido
  }
  return false; // insercao mal-sucedido
}

// remove elemento da Stack;
template <typename T>
bool Stack<T>::pop(T &popValue) {
  if ( !isEmpty() ) {
     popValue = stackPtr[top--]; // remove item da Stack
     return true; // remocao bem-sucedida
  }
  return false; // remocao mal-sucedida
}
```

# Exemplo 2: uso e resultado

```
int main() {
  Stack<double> doubleStack(5); // tamanho 5
  double doubleValue = 1.1;
  // insere 5 doubles em doubleStack
  while ( doubleStack.push(doubleValue) ) {
    cout << doubleValue << ' ';
    doubleValue += 1.1;
  }
  cout << endl;
  while ( doubleStack.pop(doubleValue) )
    cout << doubleValue << ' ';
  cout << endl;

  Stack<int> intStack; // tamanho padrao de 10
  int intValue = 1;
  // insere 10 inteiros em intStack
  while ( intStack.push(intValue) ) {
    cout << intValue << ' ';
    intValue++;
  }
  cout << endl;
  // remove elementos de intStack
  while ( intStack.pop(intValue) )
    cout << intValue << ' ';
  cout << endl;
  return 0;
}
```

```
1.1 2.2 3.3 4.4 5.5
5.5 4.4 3.3 2.2 1.1
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

# Classes Genéricas

- Os membros de uma **classe genérica** são **funções genéricas**
  - As definições que aparecem fora da classe devem ser precedidas pelo cabeçalho:
    `template <typename T>`
  - O operador de escopo utiliza o nome da **classe genérica**:
    `Stack <T>`
  - Na implementação, os elementos da pilha aparecem genéricamente como sendo do tipo `T`

# Classes Gené ricas

- Para instanciar um objeto de uma **classe genérica**, é preciso informar qual tipo deve ser associado à classe
  - No exemplo, foram declarados dois objetos, associando os tipos `double` e `int`
  - O compilador substituirá o tipo do `T` da definição da classe pelo tipo informado, e criará uma nova codificação da classe
  - Note que o código na função principal é idêntico para as duas pilhas criadas

# Outros Parâmetros e Parâmetros Padronizados

# Outros Parâmetros

- A **classe genérica** do exemplo anterior recebia apenas um parâmetro, que representava um tipo
  - No entanto, uma classe pode receber mais que um parâmetro
  - O parâmetro não precisa necessariamente representar um tipo
  - Por exemplo, uma **classe genérica** poderia receber também um inteiro que representasse o tamanho da pilha

```
template <typename T, int elementos>
class Stack {
```

# Parâmetros Padronizados

- Outra possibilidade é definir valores padrão para os parâmetros de uma **classe genérica**:

```
template <typename T = int>
class Stack {
```

- No exemplo acima, caso um tipo não seja especificado, será criada uma pilha de inteiros pela chamada abaixo:

```
Stack<> intStack(5);
```

- Parâmetros padronizados devem ser definidos mais à direita na lista de parâmetros
  - Quando se instancia uma classe com um ou mais parâmetros padronizados, se um parâmetro omitido não é o mais à direita, então todos os parâmetros depois dele também devem ser omitidos

# Classes Genéricas e Herança

# Classes Genéricas e Herança

- Os conceitos associados à **herança** podem ser utilizados em **classes genéricas**
  - Além de gerar classes, é possível gerar hierarquias de classes com templates
  - Cada hieraquia é parametrizada
  - **Funções virtuais** são permitidas na hierarquia, ou seja, é possível usar **polimorfismo**
  - Pode-se ter também **classes template abstratas**, etc.

# Exemplo 3

```
#include <iostream>

using namespace std;

template <class T>
class Base {
private:
  T tob;
public:
  Base(T par=0): tob(par) {}

  virtual void fpoli( ) {
    cout << "base " << tob << endl;
  }
};

template <class T>
class Deriv : public Base<T> {
private:
  T tod;
public:
  Deriv(T parb=0, T pard=0) : Base<T>(parb), tod(pard) { }

  void fpoli() {
    cout << "deriv " << tod << endl;
  }
};
```

```
template <class T>
void f(Base<T> *ptb) {
  ptb->fpoli();
}

int main() {
  Deriv<int> i0;
  i0.fpoli();
  Base<int> i1(1);
  i1.fpoli();
  Base<float> *ptbf = new Base<float>(10.);
  ptbf->fpoli();
  f(ptbf);
  delete ptbf;
  ptbf = new Deriv<float>(20.,30.);
  ptbf->fpoli();
  f(ptbf);
  delete ptbf;
  return 0;
}
```

# Considerações sobre o uso de const

# Considerações sobre o uso de const

- Observe o seguinte trecho de código, que envolve uma variável e um ponteiro

```
int *x;      // x eh um ponteiro para int
int a = 15; // a eh um inteiro

x = &a;
cout << "a = " << a << endl;
cout << "*x = " << *x << endl;
a++;
(*x)++;
cout << "a = " << a << endl;
cout << "*x = " << *x << endl;
int b = 20; // b eh um inteiro
x = &b;
cout << "b = " << b << endl;
cout << "*x = " << *x << endl;
```

- É possível ter um ponteiro variável apontando para um conteúdo constante

```
int const *y; // y eh um ponteiro para
              //   um int constante
// eh a mesma coisa que: const int *y;
const int c = 25; // c eh um inteiro constante

y = &c;
cout << "c = " << c << endl;
cout << "*y = " << *y << endl;
// NAO PERMITIDO: c++;
// NAO PERMITIDO: (*y)++;
int const d = 30; // d eh um inteiro constante
y = &d;
cout << "d = " << d << endl;
cout << "*y = " << *y << endl;
```

# Considerações sobre o uso de const

- Também seria possível ter um ponteiro constante apontando para um conteúdo variável

```
int e = 35; // e eh um inteiro
int * const z = &e; // z eh um ponteiro
                    //  constante para um int

cout << "e = " << e << endl;
cout << "*z = " << *z << endl;
e++;
(*z)++;
cout << "e = " << e << endl;
cout << "*z = " << *z << endl;
int f = 40; // f eh um inteiro
// NAO PERMITIDO: z = &f;
```

- Por fim, seria possível ter um ponteiro constante apontando para um conteúdo constante

```
const int g = 45; // g eh um inteiro constante
int const * const w = &g; // w eh um ponteiro
                          // constante para um
                          // int constante
// eh equivalente a: const int * const w = &g;

cout << "g = " << g << endl;
cout << "*w = " << *w << endl;
// NAO PERMITIDO: g++;
// NAO PERMITIDO: (*w)++;
const int h = 50; // h eh um inteiro constante
// NAO PERMITIDO: w = &h;
```

# Lista de Exercícios

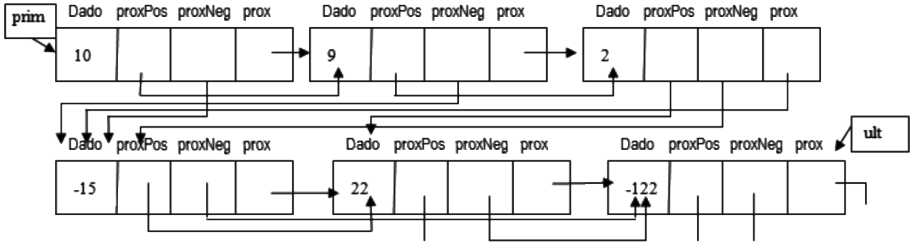
# Exercício 1

1. A função abaixo só ordena arranjos de elementos do tipo `int`.
   - Escreva a função genérica correspondente;
   - Escreva um programa que utiliza a função genérica.

```
void selection_sort(int v[], int N){
  for(int i=0; i<N-1; i++){
     int menor = i;
     for(int j=i+1; j<N; j++)
        if (v[j]<v[menor]) menor = j;
     if (menor!=i) {
        int aux = v[i];
        v[i] = v[menor];
        v[menor] = aux;
     }
  }
}
```

# Exercício 2

2. Modifique a classe `ListaLinkSinal` do exercício da aula sobre "Estruturas Encadeadas" para que ao invés de armazenar números inteiros (atributo Dado), ela possa armazenar um string ou um número real.



# Creditos

# Creditos

- Estas lâminas contêm trechos de materiais disponibilizados pelos professores Rafael Garibotti e Edson Moreno.

# Soluções

# Exercício 1

```
#include <iostream>

using namespace std;

template <typename T>
void selection_sort(T v[], int N){
  for(int i=0; i<N-1; i++){
     int menor = i;
     for(int j=i+1; j<N; j++)
        if (v[j]<v[menor]) menor = j;
     if (menor!=i) {
        T aux = v[i];
        v[i] = v[menor];
        v[menor] = aux;
     }
  }
}
```

```
int main() {
  int v1[] = { 10, 2, 9, 3, 1, 5, 6, 4, 7, 8};
  int t1 = sizeof(v1) / sizeof(v1[0]);
  selection_sort(v1,t1);
  for (int i=0; i<t1; ++i) {
        if (i>0) cout << " ";
        cout << v1[i];
  }
  cout << endl;

  double v2[] = { 3.3, 2.2, 5.5, 4.4, 1.1, 6.6};
  int t2 = sizeof(v2) / sizeof(v2[0]);
  selection_sort(v2,t2);
  for (int i=0; i<t2; ++i) {
        if (i>0) cout << " ";
        cout << v2[i];
  }
  cout << endl;

  return 0;
}
```