Introdução a Técnicas de Programação

Templates

Prof. André Campos
DIMAp/UFRN

Alguns problemas na reutilização de código...

Digamos que você precise definir uma função para encontrar o maior valor de uma sequência de dados:

```
int max_value(int n, int values[]) {
  int max = 0;
  if (n > 0) {
    max = values[0];
    for(int i = 1; i < n; i++) {
      if (values[i] > max) {
        max = values[i];
      }
    }
  }
  return max;
}
```

Alguns problemas na reutilização de código...

Mas a função anterior funciona APENAS para uma sequência de inteiros.
Digamos que você quer utilizar a mesma lógica para inteiros e doubles

```
double max_value(int n, double values[]) {
  double max = 0;
  if (n > 0) {
    max = values[0];
    for(int i = 1; i < n; i++) {
      if (values[i] > max) {
        max = values[i];
      }
    }
  }
  return max;
}
```

Sobrecarga de função

Precisamos de 2 funções diferentes por causa dos tipos de dados.

```
int max_value(int n, int values[]) {
  int max = 0;
  ...
}

double max_value(int n, double values[]) {
  double max = 0;
  ...
}
```

Função com mesmo nome, porém com parâmetros diferentes.
O compilador identifica qual das funções chamar de acordo com os parâmetros

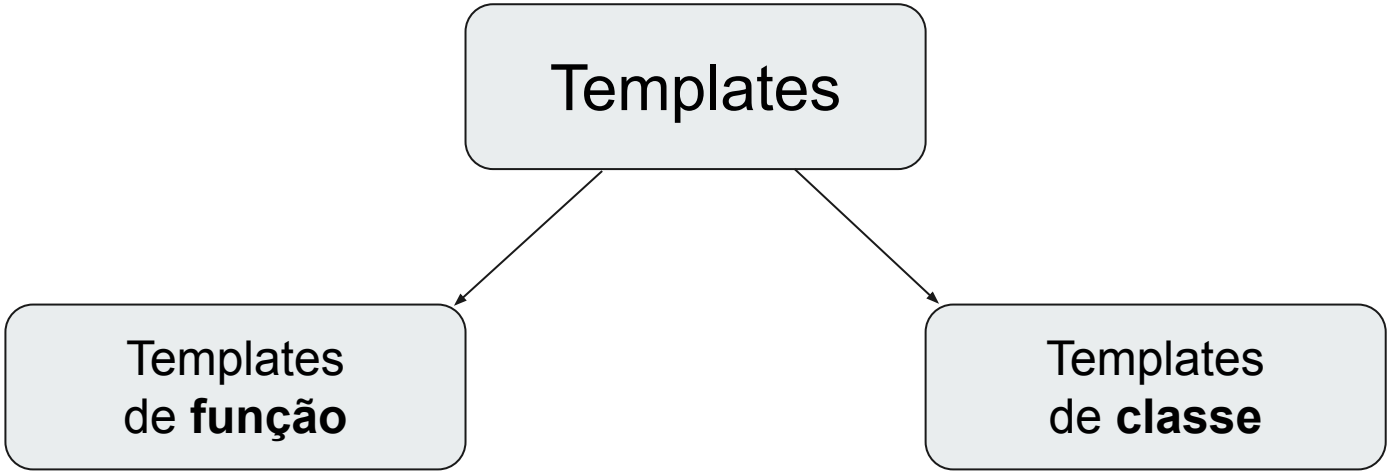
Templates

Há um mecanismo em C++ (e em outras linguagens) em que se pode definir a lógica de um algoritmo, variando os tipos de dados: **templates**

```
int max_value(int n, int values[]) {
  int max = 0;
  ...
}

double max_value(int n, double values[]) {
  double max = 0;
  ...
}
```

Útil quando a lógica é a mesma, mas os dados não.

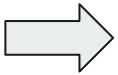
Tipos de templates



Template de função

É como se fosse um “gabarito” onde o tipo é preenchido quando a função for chamada. A função passa a ser **genérica**.

```
int max_value(int n, int values[]) {
  int max = 0;
  ...
}

double max_value(int n, double values[]) {
  double max = 0;
  ...
}
```



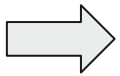
```
template <typename T>
T max_value(int n, T values[]) {
  T max = 0;
  if (n > 0) {
    max = values[0];
    for(int i = 1; i < n; i++) {
      if (values[i] > max) {
        max = values[i];
      }
    }
  }
  return max;
}
```

O termo **Template** é usado em C++, mas em outras linguagens é **Generics**.

Template de função

O compilador se encarrega de criar as várias versões da função

```
int main() {
  int vi[] = { 5, 4, 2, 8, 3, 1 };
  double vd[] = { 6.4, 2.3, 3.1 };

  int mi = max_value(6, vi);
  double md = max_value(3, vd);

  cout << mi << endl;
  cout << md << endl;

  return 0;
}
```

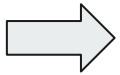


```
template <typename T>
T max_value(int n, T values[]) {
  T max = 0;
  ...
}
```

Template de função

O compilador se encarrega de criar as várias versões da função.
O tipo é “inferido” pelos parâmetros passados.

```
int main() {
  int vi[] = { 5, 4, 2, 8, 3, 1 };
  double vd[] = { 6.4, 2.3, 3.1 };

  int mi = max_value(6, vi);
  double md = max_value(3, vd);

  cout << mi << endl;
  cout << md << endl;

  return 0;
}
```



```
int max_value(int n, int values[]) {
  int max = 0;
  ...
}

double max_value(int n, double values[]) {
  double max = 0;
  ...
}
```

Template de função

Quando não é possível inferir o tipo pelos parâmetros, é necessário indicar o tipo através de `<tipo>` depois do nome da função.

```
template <typename T>
T** create_matrix(int num_lin, int num_col) {
  T **mat = new T*[num_lin];
  for (int i = 0; i < num_lin; i++) {
    mat[i] = new T[num_col];
  }
  return mat;
}

int main() {
  int **mat = create_matrix<int>(3, 4);
  ...
}
```

Template de classe

Os atributos e os métodos de uma classe também podem ser genéricos

```
class IntArray {
  int size;
  int *data;

public:
  IntArray(int s): size(s) {
    data = new int[s];
  }

  ~IntArray() {
    delete[] data;
  }
};
```

```
class DoubleArray {
  int size;
  double *data;

public:
  DoubleArray(int s): size(s) {
    data = new double[s];
  }

  ~DoubleArray() {
    delete[] data;
  }
};
```

Template de classe

Os atributos e os métodos de uma classe também podem ser genéricos. Quando não é possível inferir o tipo, usa-se também `<tipo>`.

```
template <typename T>
class Array {
  int size;
  T *data;

public:
  Array(int s = 10): size(s) {
    data = new T[s];
  }
  ~Array() {
    delete[] data;
  }
};
```

```
int main() {
  Array<int> a;
  Array<double> b(50);

  ...
}
```

Tipo padrão (default)

É possível definir um tipo a ser usado quando nenhum for identificado.

```
template <typename T = int>
class Array {
  int size;
  T *data;

public:
  Array(int s = 10): size(s) {
    data = new T[s];
  }
  ~Array() {
    delete[] data;
  }
};
```

```
int main() {
  Array a;
  Array<double> b(50);

  ...
}
```

Mais de um tipo como argumentos

É possível definir mais de um tipo genérico.

```
template <typename T, typename U>
class Pair {
  T v1;
  U v2;

public:
  Pair(T v1, U v2): v1(v1), v2(v2) {}

  T first() { return v1; }
  U second() { return v2; }
};
```

```
int main() {
  Pair p1(5UL, "fulano");
  Pair p2('A', 5.35);

  cout << p1.first() << endl;
  cout << p1.second() << endl;
  cout << p2.first() << endl;
  cout << p2.second() << endl;

  ...
}
```

Argumentos do template que não são tipos

É possível também definir valores constantes para o template.
Devem ser valores constantes para o compilador criar os diferentes tipos

```
template <typename T, int S>
class Array {
  T data[S];

public:
  Array() {
    // data = new T[S];
  }

  ~Array() {
    // delete[] data;
  }
};
```

```
int main() {
  Array<int, 10> a;
  Array<double, 50> b;

  ...
}
```

Algoritmos e containers da STL
Standart Template Library

Biblioteca STL do C++

STL (*Standard Template Library*) fornece um conjunto de templates de classes e funções para várias tarefas comuns de programação.

É uma coleção de **algoritmos e estruturas de dados** e outros elementos definidos de forma genérica (template) para facilitar o reuso:

- Algoritmos (ex: ordenação, busca, acumulação, ...)
- **Containers** (ex: array, vetor, fila, pilha, deque, mapa, ...)
- Objetos de função (functors)
- Iteradores (permite generalizar a forma de percorrer sequências)
- Utilitários (ex: pares, tuplas...)

A biblioteca de containers do C++

A biblioteca padrão fornece um conjunto de estruturas de dados úteis para a maioria dos programas e que faz uso extensivo de templates.

`array`	Sequência de tamanho fixo com elementos genéricos. Similar ao tipo array.
`vector`	Sequência de elementos genéricos cujo tamanho pode mudar dinamicamente.
`stack`	Container que implementa uma pilha de elementos genéricos (LIFO - *last-in first-out*)
`queue`	Container que implementa uma fila de elementos genéricos (FIFO - *first-in first-out*)
`set`	Container que guarda elementos genéricos únicos (não há repetições)
`map`	Container que associa seus elementos a uma chave única.
...	

array e vector

array

- Cria um array de tipo genérico de tamanho fixo (constante).
- É similar ao tipo array[], mas é uma classe com métodos e operadores que ajudam em várias situações.
- Precisa incluir o cabeçalho `#include <array>`

vector

- Cria um array cujo tamanho é dinâmico (cresce à medida que novos valores são inseridos)
- Precisa incluir o cabeçalho `#include <vector>`

array

```cpp
#include <iostream>
#include <array>
using namespace std;

int main() {
  array<int, 5> a, b;
  int c[5];

  cout << "A tem " << a.size() << " elementos" << endl;
  a.fill(1);
  a = b;
  cout << "A e B são " << (a == b ? "iguais" : "diferentes") << endl;
  b[2] = 100;
  cout << "A e B são " << (a == b ? "iguais" : "diferentes") << endl;

  return 0;
}
```

vector

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> a;

  cout << a.size() << "/" << a.capacity() << endl;
  for (int i = 0; i < 17; i++) {
    a.push_back(i);
    cout << a.size() << "/" << a.capacity() << endl;
  }
  a[2] = 100;
  return 0;
}
```

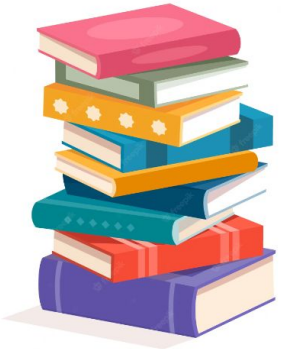
```
0/0
1/1
2/2
3/4
4/4
5/8
6/8
7/8
8/8
9/16
10/16
11/16
12/16
13/16
14/16
15/16
16/16
17/32
```

Uma implementação simples de vector

```
template <typename T>
class vector {
  int siz, cap;
  T *arr;
public:
  vector(): siz(0), cap(0), arr(nullptr) { }
  ~vector() { if (arr) delete[] arr; }

  int size() { return siz; }
  int capacity() { return cap; }

  T& at(int index) {
    if (index >= 0 && index < siz) {
      return arr[index];
    }
  }
  ...
```

```
  ...
  void push_back(const T& val) {
    if (siz == cap) {
      cap = cap == 0 ? 1 : cap * 2;
      T *new_arr = new T[cap];

      for (int i = 0; i < siz; i++) {
        new_arr[i] = arr[i];
      }
      if (arr) {
        delete[] arr;
      }
      arr = new_arr;
    }
    arr[siz++] = val;
  }
};
```

stack e queue



stack

- Implementa uma pilha (LIFO - last-in first-out)
- Precisa incluir o cabeçalho `#include <stack>`

queue

- Implementa uma fila (FIFO - first-in first-out)
- Precisa incluir o cabeçalho `#include <vector>`

stack

A pilha possui 3 operações básicas:

- `push()` - empilha
- `pop()` - desempilha
- `top()` - consulta o topo

O que esse código ao lado faz?

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
  int a[] = { 1, 2, 3, 4, 5 };
  stack<int> s;

  for (int v: a) {
    s.push(v);
  }
  for (int &v: a) {
    v = s.top();
    s.pop();
  }
  for (int v: a) {
    cout << v << " ";
  }
  cout << endl;
}
```

queue

A fila possui 4 operações básicas:

- `push()` - insere no final
- `pop()` - remove o primeiro
- `front()` - consulta o primeiro
- `back()` - consulta o último

O que esse código ao lado faz?

```
...
int main() {
  queue<int> q1, q2;
  vector<int> q;
  int a[]={ 17, 2, 151, -1, 14, -1, -1, 5 };

  for (int v: a) {
    if (v > 100) q2.push(v);
    else if (v > 0) q1.push(v);
    else if (!q2.empty()) {
      q.push_back(q2.front());
      q2.pop();
    }
    else if (!q1.empty()) {
      q.push_back(q1.front());
      q1.pop();
    }
  }
  for (int v: q) cout << v << " ";
  cout << endl;
}
```