

Templates em C++

Algoritmos e Estruturas de Dados

2020/2021



Mestrado Integrado em Engenharia Informática e Computação

Template de função

- Usada para implementar uma única função que executa operações idênticas para diferentes tipos de dados
- Função genérica definida em função de parâmetros que representam o tipo de dados a operar
- O tipo de dados é especificado (instanciado) quando a função é chamada
- Precede-se o cabeçalho da função da palavra-chave **template** seguida de uma lista de parâmetros entre **< >**, que representam o tipo de dados a instanciar mais tarde, e o seu nome.
 - cada parâmetro é precedido da palavra-chave **class**.



AED – 2020/21

2

Templates de funções

```
template <class T>
void printArray (const T * array, int size)
{
    for (int i=0; i<size; i++)
        cout << array[i] << " ";
    cout << endl;
}

main()
{
    const int aSize=5, bSize=4;
    int a[aSize] = {1, 2, 3, 4, 5};
    float b[bSize] = {1.1, 2.2, 3.3, 4.4};
    printArray(a, aSize);
    printArray(b, bSize);
    return 0;
}
```



Templates de classes

- Permite a generalização de classes. Classes similares que possuem diferentes tipos de dados para os mesmos membros, não necessitam ser definidas mais que uma vez.
- "Classe genérica" (também chamada classe parametrizada) definida em função de parâmetros a instanciar para se ter uma "classe ordinária"
- Precede-se a definição da "classe genérica" por:
template <class nome-de-parâmetro, ... >
- Para se obter uma classe ordinária, é necessário instanciar os parâmetros, numa lista entre <> a seguir ao nome da classe genérica



MemoryCell *template* - interface

```
// a class for simulating a memory cell

template <class Object>
class MemoryCell
{
public:
    explicit MemoryCell(const Object & initialValue = Object() );
    // valor defeito é construtor sem parâmetros
    const Object & read() const;
    void write(const Object & x);
private:
    Object storedValue;
};
```



MemoryCell *template* - implementação

```
template <class Object>
MemoryCell<Object>::MemoryCell(const Object & initialValue) :
    storedValue(initialValue)
{}

template <class Object>
const Object & MemoryCell<Object>::read() const
{
    return storedValue;
}

template <class Object>
void MemoryCell<Object>::write(const Object & x)
{
    storedValue = x;
}
```



MemoryCell *template* - teste

```
int main()
{
    MemoryCell<int> m1;
    MemoryCell<string> m2 ("hello");

    m1.write(37);
    m2.write(m2.read() + " world" );
    cout << m1.read() << endl << m2.read() << endl;

    return 0;
}
```



matrix *template*

```
template <class Object>
class matrix
{
    private:
        vector< vector<Object> > array;

    public:
        matrix(int rows, int cols): array(rows)
        {
            for (int i=0; i<rows; i++)
                array[i].resize(cols);
        }
}
```



matrix *template*

```
const vector<Object> & operator [] (int row) const
{ return array[row]; }

vector<Object> & operator [] (int row)
{ return array[row]; }

int numrows() const
{ return array.size(); }

int numcols() const
{ return numrows()>0 ? array[0].size() : 0 ; }

};
```



matrix - *operator []*

- operator [] tem duas versões
 - é const e retorna referência constante : acesso
 - não é const e retorna referência : mutação

Considere o seguinte método de cópia de matrizes:

```
void copy(const matrix<int> & from, matrix<int> & to)
{
    for(int i=0; i<to.numrows(); i++)
        to[i] = from[i];
}
```

- operator[] deve retornar uma referência
- mas assim, from[i] = to[i] é válido (não pode ser)
- Solução: operator[] deve retornar uma referência constante para from , mas uma referência não constante para to



Objetos funcionais

- É útil passar funções como argumentos

```
bool less_than_7(int v) {  
    return v<7;  
}  
  
void function1(vector<int> &v)  
{  
    vector<int>::iterator it = find_if(v.begin(), v.end(), less_than_7);  
    // ...  
}
```

- Mas muitas vezes é necessário que a função chamada guarde valores entre invocações sucessivas \Rightarrow **uso de classes**
- **Objeto funcional**
 - Objeto de uma classe com um operador de função



Objetos funcionais

```
template <class T>  
class Sum  
{  
    T res;  
public:  
    Sum (T i=0) : res(i) {}  
    void operator() (T x) { res += x; }  
    T result() const { return res; }  
};  
  
template <class T, class Op>  
Op for_each(T first, T last, Op f)  
{  
    while (first != last)  
        f(*first++)  
    return f;  
}
```



Objetos funcionais

```
int main()
{
    vector<int> v;
    v.push_back(2);
    v.push_back(5);
    v.push_back(8);
    v.push_back(3);
    Sum<int> s;
    s = for_each(v.begin(), v.end(), s);
    cout << "The sum is " << s.result(); << endl;
}
```

Resultado = ?

