Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030
◁ Implementando Pilhas e Filas Genéricas com Templates ▷
7 de abril de 2016

Sumário

Apresentação

A presente tarefa tem por objetivo descrever a implementação das TADs **Pilhas** e **Filas** por meio de classes template. A utilização de template visa separar o processo de descrição do comportamento de uma classe de sua implementação com tipos de dados específicos. Esta capacidade de abstração é uma valiosa característica na implementação de estruturas de dados.

1 Estruturas de Dados, Classes e Gabaritos

As estruturas de dados ou TADs, em princípio, definem um *comportamento* de como certos elementos (dados) devem ser armazenados e manipulados. Tal comportamento, idealmente, deveria ser independente do tipo de dado que a estrutura de dados deve armazenar.

Uma **Pilha**, por exemplo, é um *container* com um comportamento de armazenamento bem definido (i.e. LIFO – *Last In, First Out*). Tal comportamento – empilhar – deve ser realizado de forma independente do tipo de informação que se deseja armazenar, seja ela, por exemplo, *inteiros*, *pessoas* ou mesmo outras *pilhas*.

O uso de templates é uma forma de viabilizar a separação entre comportamento e tipo de dados, ao permitir a definição de funções e classes que possuam *parâmetros para nomes de tipo*.

2 Templates de Classes

Templates de classe seguem a mesma idéia de template de funções e possuem uma sintaxe um pouco mais elaborada, como mostra a Listagem 1 que exibe uma classe que armazena um *par de valores*.

Código 1 Implementação (parcial) e uso da classe `Pair` com templates.

```
1  /**Classe para um par de valores genéricos T. */
2  template<typename T>
3    class Pair {
4      public:
5      Pair( T firstValue, T secondValue );
6        void setFirst( T newValue );
7        void setSecond( T newValue );
8        T getFirst( ) const;
9        T getSecond( ) const;
10     private:
11       T m_First;
12       T m_Second;
13   };
14
15   template<typename T>
16   Pair<T>::Pair( T firstValue, T secondValue ):
17     m_First( firstValue ), m_Second( secondValue )
18     { /* Empty */ }
19
20   template<typename T>
21   void Pair<T>::setFirst( T newValue ) {
22     m_First = newValue;
23   }
24   ...
25   int main( ) {
26     Pair<int> point;
27     point.setFirst( 1 );
28     ...
29     return EXIT_SUCCESS;
30   }
```

Note novamente o uso de `template <typename T>` antes da declaração da classe (linha 2) e antes da implementação de cada método da classe (linhas 15 e 20)[1].

3 Warming up: Implementar a Classe Pilha de Inteiros

Implemente a classe `StackInt` (pilha de inteiros) utilizando um vetor de inteiros com base nos métodos descritos na interface `AbsStack.h`, apresentada na Listagem 2 do Apêndice A.

A especificação da classe `StackInt` deve estar em um arquivo chamado `stackint.h`, enquanto que a implementação da classe deve estar em um arquivo chamado `stackint.cpp`.

Desenvolva um programa teste `drive_stackint.cpp` para testar de maneira completa todos os métodos da classe `StackInt`.

[1] A classe da Listagem 1 **não** está totalmente implementada.

Posteriormente, experimente fazer com que o vetor de armazenamento da pilha tenha um comportamento dinâmico, ou seja, o vetor deve dobrar de capacidade via *alocação dinâmica* sempre que seu limite máximo for alcançado. Com isso, a pilha passa a ter sua capacidade de armazenamento limitada apenas à memória disponível no dispositivo em que o programa for executado. Note que o processo de dinâmico de duplicação de memória deve ocorrer de maneira transparente para o usuário, o que implica que a pilha deve se preocupar em preservar os valores originalmente armazenados durante a expansão da memória.

4 Classe Pilha Genérica

Crie uma nova classe `StackAr` que implementa um pilha genérica capaz de empilhar objetos arbitrários, utilizando *templates*. Esta classe deve estender (herança) a classe interface definida em `AbsStack.h` e, consequentemente, implementar todos os métodos descritos na referida interface. Segue abaixo como o processo de extensão pode ser feito:

```
#ifndef STACKAR_H
#define STACKAR_H

#include "AbsStack.h" // Inclui a interface abstrata da pilha.

// Implementando interface via herança.
template <typename Object>
class StackAr : public AbsStack<Object> {
  public:
     ...
  private:
  ...
};
#endif
```

A classe `StackAr` deve ser implementada utilizando-se um arranjo unidimensional[2], com capacidade *default* de 10 elementos. O código cliente pode informar um tamanho alternativo, passando tal valor por parâmetro para o construtor.

Por utilizar um vetor com capacidade de armazenamento pré-determinada, a classe `StackAr` deve ser capaz de lançar a exceção `std::length_error` quando uma inserção exceder a capacidade de armazenamento da pilha ou quando uma remoção ou consulta é solicitada sobre uma pilha vazia. Esta exceção é definida em `<stdexcept>`.

Você também deve testar sua classe de forma sistemática, assegurando-se de que cada método será chamando nas mais variadas situações. A declaração da classe `StackAr` deve ser armazenada no arquivo `stackar.h`, a implementação no arquivo `stackar.inl`.

[2]Não serão aceitas soluções que utilizem classes do STL para armazenamento interno.

5 Classe Fila Genérica

Crie uma classe `QueueAr` que implementa um fila genérica capaz de enfileirar objetos arbitrários, utilizando *templates*. Esta classe deve estender (herança) a classe interface definida em `AbsQueue.h` (veja Listagem 3, no Apêndice B) e, consequentemente, implementar todos os métodos descritos na referida interface.

O armazenamento interno desta classe também deve ser feito através de um arranjo unidimensional. Contudo, a classe `QueueAr` deve ser mais versátil que a `StackAr` no que diz respeito ao gerenciamento de memória. Assim, ela deve apresentar uma capacidade de armazenamento que aumenta conforme necessário. Com esta medida, pode-se evitar a condição de *overflow* de fila. Note, entretanto, que a condição de *underflow* de fila permanece, ou seja, qualquer consulta ou tentativa de desenfileirar um elemento de uma fila vazia deve gerar a exceção `std::out_of_range`.

Novamente, lembre-se de testar sua classe de forma sistemática. A definição e implementação da classe `QueueAr` deve ser disponibilizada em um *único* arquivo, chamado `queuear.h`.

Considerções

Note que a declaração de uma classe template e sua codificação devem ser armazenados em um **mesmo** arquivo (com extensão **.h**), ao contrário do que normalmente é feito (i.e. declaração da classe no arquivo **.h** e implementação dos métodos no arquivo **.cpp**). Porém, isso pode ser contornado fazendo com que o arquivo **.h** inclua (via diretiva `#include "xxxxx.inl"`) o arquivo **.inl**[3]. Tal diretiva de inclusão deve ser colocada no final do arquivo **.h**.

Outra observação relevante é que uma classe template não pode ser compilada de forma isolada. Ela só vai ser 'compilada' quando for *definida* (i.e. utilizada) por algum programa. Isso pode ocorrer, por exemplo, quando algum objeto da classe for instanciado — somente nesse momento o código é gerado e a sintaxe analisada.

A Código da Classe Abstrata `AbsStack`

A declaração de função-membro na forma

```
virtual <tipo_retorno> <nome_função>( <lista_de_parâmetros> ) = 0;
```

é uma forma de implementar o conceitos de *interface*, e torna a classe ao qual o método pertence em uma *classe abstrata*. Declarando métodos como *funções virtuais puras*, é uma indicação de que se trata de uma classe-base para outras classes, ou seja, esta classe não pode ser instanciada, apenas serve de modelo para uma classe derivada. Com isso, a classe que for estender a `AbsStack` será **obrigada** a implementar todos os métodos virtuais, desta forma garantido a implementação da interface.

Note que a classe não possui nenhuma indicação de como os dados serão armazenados internamente, pois sua única função é definir a interface através da qual o *comportamento* é disponibilizado,

[3] É aconselhável que isso seja feito apenas para classes templates.

ou seja, a interface que o código cliente terá acesso para usar a pilha.

Código 2 Implementação da interface de uma pilha abstrata, `AbsStack.h`.

```
1  /**Classe interface pilha (não pode ser instanciada!).  */
2   template <class Object>
3   class AbsStack {
4     public:
5      AbsStack( ) { /* Empty */ }          // Default constructor
6      virtual ~AbsStack() { /* Empty */ } // Default destructor
7
8      // Basic members
9      virtual void push( const Object & x ) = 0;
10     virtual Object pop( ) = 0;
11     virtual Object top( ) const = 0;
12
13     virtual bool isEmpty( ) const = 0;
14     virtual void makeEmpty( ) = 0;
15
16    private:
17     // Disable copy constructor
18     AbsStack( const AbsStack & ) { /* Empty */ }
19  };
```

B Código da Classe Abstrata `AbsQueue`

Código 3 Implementação da interface de uma fila abstrata, `AbsQueue.h`.

```
1  /**Classe interface fila (não pode ser instanciada!).  */
2   template <class Object>
3   class AbsQueue{
4     public:
5      AbsQueue( ) { /* Empty */ }          // Default constructor
6      virtual ~AbsQueue() { /* Empty */ } // Default destructor
7
8      // Basic members
9      virtual void enqueue( const Object & x ) = 0;
10     virtual Object dequeue( ) = 0;
11     virtual Object getFront( ) const = 0;
12
13     virtual bool isEmpty( ) const = 0;
14     virtual void makeEmpty( ) = 0;
15
16    private:
17     // Disable copy constructor
18     AbsQueue( const AbsQueue & ) { /* Empty */ }
19  };
```