Apêndice H

Listas e iteradores: listagens

Este apêndice contém as listagens completas das várias versões do módulo físico `lista_int` desenvolvidas ao longo do Capítulo 10 e do Capítulo 11.

H.1 Versão simplista

Corresponde à versão desenvolvida no início do Capítulo 10, que não usa ponteiros nem variáveis dinâmicas, e na qual os itens são guardados numa matriz pela mesma ordem pela qual ocorrem na lista.

H.1.1 Ficheiro de interface: lista_int.H

Este ficheiro contém a interface do módulo `lista_int`. Contém também a declaração dos membros privados das classes, o que corresponde a uma parte da implementação:

```
#ifndef LISTA_INT_H
#define LISTA_INT_H

#include <iostream>

/** Representa listas de itens do tipo Item. Item é actualmente é um sinónimo
    de int, mas que pode ser alterado facilmente para o tipo que se entender.
    Por convenção, chama-se "frente" e "trás" ao primeiro e último item na lista.
    Os nomes "primeiro", "último", "início" e "fim" são reservados para iteradores. */
class ListaDeInt {
  public:

    /** Sinónimo de int. Usa-se para simplificar a tarefa de criar listas com
        itens de outros tipos. */
    typedef int Item;
```

```
/* Declaração de uma classe embutida que serve para percorrer e manipular
   listas: */
class Iterador;


// Construtores:

/// Construtor da classe, cria uma lista vazia.
ListaDeInt();


// Inspectores:

/// Devolve o comprimento da lista, ou seja, o seu número de itens.
int comprimento() const;

/// Indica se a lista está vazia.
bool estáVazia() const;

/// Indica se a lista está cheia.
bool estáCheia() const;

/** Devolve referência constante para o item na frente da lista.
    @pre PC ≡ ¬estáVazia(). */
Item const& frente() const;

/** Devolve referência constante para o item na traseira da lista.
    @pre PC ≡ ¬estáVazia(). */
Item const& trás() const;


// Modificadores:

/** Devolve referência para o item na frente da lista. Não modifica
    directamente a lista, mas permite modificações através da referência
    devolvida.
    @pre PC ≡ ¬estáVazia(). */
Item& frente();

/** Devolve referência para o item na traseira da lista.
    @pre PC ≡ ¬estáVazia(). */
Item& trás();

/** Põe novo item na frente da lista. Invalida qualquer iterador associado
    à lista.
```

```
    @pre  PC ≡ ¬estáCheia(). */
void põeNaFrente(Item const& novo_item);

/** Põe novo item na traseira da lista. Invalida qualquer iterador associado
    à lista.
    @pre  PC ≡ ¬estáCheia(). */
void põeAtrás(Item const& novo_item);

/** Tira o item da frente da lista. Invalida qualquer iterador associado
    à lista.
    @pre  PC ≡ ¬estáVazia(). */
void tiraDaFrente();

/** Tira o item da traseira da lista. Invalida qualquer iterador associado
    à lista.
    @pre  PC ≡ ¬estáVazia(). */
void tiraDeTrás();

/** Esvazia a lista. Invalida qualquer iterador associado
    à lista. */
void esvazia();

/** Insere novo item imediatamente antes da posição indicada pelo iterador
    i. Faz com que o iterador continue a referenciar o mesmo item que
    antes da inserção. Invalida qualquer outro iterador associado
    à lista.
    @pre  PC ≡ ¬estáCheia ∧ iterador ≠ fim() ∧ iterador é válido. */
void insereAntes(Iterador& iterador, Item const& novo_item);
/* Atenção! Em todo o rigor o iterador deveria ser constante, pois o mantém-se
   referenciando o mesmo item! */

/** Remove o item referenciado pelo iterador i. O iterador fica a referenciar
    o item logo após o item removido. Invalida qualquer outro iterador
    associado à lista.
    @pre  PC ≡ iterador ≠ início() ∧ iterador ≠ fim()∧
               iterador é válido. */
void remove(Iterador& i);


/* Funções construtoras de iteradores. Consideram-se modificadoras porque a lista
   pode ser modificada através dos iteradores. */

/** Devolve um novo iterador inicial, i.e., um iterador referenciando o item fictício
    imediatamente antes do item na frente da lista. */
Iterador início();
```

```
    /** Devolve um novo iterador final, i.e., um iterador referenciando o item fictício
          imediatamente após o item na traseira da lista. */
    Iterador fim();

    /** Devolve um novo primeiro iterador, i.e., um iterador referenciando o item na
          frente da lista. Note-se que se a lista estiver vazia o primeiro iterador é igual
          ao iterador final. */
    Iterador primeiro();

    /** Devolve um novo último iterador, i.e., um iterador referenciando o item na
          traseira da lista. Note-se que se a lista estiver vazia o último iterador é igual
          ao iterador inicial. */
    Iterador último();

  private:

    // O número máximo de itens na lista:
    static int const número_máximo_de_itens = 100;

    // Matriz que guarda os itens da lista:
    Item itens[número_máximo_de_itens];

    // Contador do número de itens na lista:
    int número_de_itens;

    /* Função auxiliar que indica se a condição invariante de instância da classe
         se verifica: */
    bool cumpreInvariante() const;

    // A classe de iteração tem acesso irrestrito às listas:
    friend class Iterador;

};

/** Representa iteradores para itens de listas do tipo ListaDeInt.

      Os iteradores têm uma característica infeliz: podem estar em estados inválidos.
      Por exemplo, se uma lista for esvaziada, todos os iteradores a ela associada
      ficam inválidos. É possível resolver este problema, mas à custa de um aumento
      considerável da complexidade deste par de classes. */
class ListaDeInt::Iterador {
  public:

    // Construtores:

    /** Construtor da classe. Associa o iterador com a lista passada como
```

```
        argumento e põe-no a referenciar o item na sua frente. */
explicit Iterador(ListaDeInt& lista_a_associar);


// Inspectores:

/** Devolve uma referência para o item referenciado pelo iterador.
    Note-se que a referência devolvida não é constante. É que um
    iterador const não pode ser alterado (avançar ou recuar), mas
    permite alterar o item por ele referenciado na lista associada.
    @pre PC ≡ O item referenciado não pode ser nenhum dos itens fictícios
              da lista (i.e., nem o item antes da frente da lista, nem o item
              após a sua traseira) e tem de ser válido. */
Item& item() const;

/** Indica se dois iteradores são iguais. Ou melhor, se a instância implícita
    é igual ao iterador passado como argumento. Dois iteradores são iguais
    se se referirem ao mesmo item da mesma lista (mesmo que sejam itens
    fictícios).
    @pre PC ≡ os iteradores têm de estar associados à mesma lista e ser
              válidos. */
bool operator == (Iterador const& outro_iterador) const;

/** Operador de diferença entre iteradores.
    @pre PC ≡ os iteradores têm de estar associados à mesma lista e ser
              válidos. */
bool operator != (Iterador const& outro_iterador) const;


// Modificadores:

/** Avança iterador para o próximo item da lista. Devolve o próprio iterador.
    @pre PC ≡ O iterador não pode ser o fim da lista associada e tem de ser
              válido. */
Iterador& operator ++ ();

/** Avança iterador para o próximo item da lista. Devolve um novo
    iterador com o valor do próprio iterador antes de avançado.
    @pre PC ≡ O iterador não pode ser o fim da lista associada e tem de ser
              válido. */
Iterador operator ++ (int);

/** Recua iterador para o item anterior da lista. Devolve o próprio iterador.
    @pre PC ≡ O iterador não pode ser o início da lista associada e tem de ser
              válido. */
Iterador& operator -- ();
```

```
        /** Recua iterador para o item anterior da lista. Devolve um novo iterador
            com o valor do próprio iterador antes de recuado.
            @pre PC ≡ O iterador não pode ser o início da lista associada e tem de ser
                      válido. */
        Iterador operator -- (int);

      private:

        // Referência para a lista a que o iterador está associado:
        ListaDeInt& lista_associada;

        // Índice do item da lista referenciado pelo iterador:
        int índice_do_item_referenciado;

        // Função auxiliar que indica se a condição invariante de instância
           da classe se verifica:
        bool cumpreInvariante() const;

        /* A classe ListaDeInt tem acesso irrestrito a todos os membros da classe
             Iterador. É importante perceber que as duas classes, ListaDeInt e
           ListaDeInt::Iterador estão completamente interligadas. Não há
           qualquer promiscuidade nesta relação. São partes do mesmo todo. */
        friend ListaDeInt;
};

#include "lista_int_impl.H"

#endif // LISTA_INT_H
```

H.1.2 Ficheiro de implementação auxiliar: `lista_int_impl.H`

Este ficheiro contem a definição de todas as rotinas e métodos em-linha do módulo `lista_int`:

```
#include <cassert>

inline ListaDeInt::ListaDeInt()
    : número_de_itens(0) {
    assert(cumpreInvariante());
}

inline int ListaDeInt::comprimento() const {
    assert(cumpreInvariante());

    return número_de_itens;
```

```
}

inline bool ListaDeInt::estáVazia() const {
    assert(cumpreInvariante());

    return comprimento() == 0;
}

inline bool ListaDeInt::estáCheia() const {
    assert(cumpreInvariante());

    return comprimento() == número_máximo_de_itens;
}

inline ListaDeInt::Item const& ListaDeInt::frente() const {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[0];
}

inline ListaDeInt::Item const& ListaDeInt::trás() const {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[número_de_itens - 1];
}

inline ListaDeInt::Item& ListaDeInt::frente() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[0];
}

inline ListaDeInt::Item& ListaDeInt::trás() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[número_de_itens - 1];
}

inline void ListaDeInt::põeAtrás(Item const& novo_item) {
    assert(cumpreInvariante());
    assert(not estáCheia());
```

```
    itens[número_de_itens++] = novo_item;

    assert(cumpreInvariante());
}

inline void ListaDeInt::tiraDeTrás() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    --número_de_itens;

    assert(cumpreInvariante());
}

inline void ListaDeInt::esvazia() {
    assert(cumpreInvariante());

    número_de_itens = 0;

    assert(cumpreInvariante());
}

inline ListaDeInt::Iterador ListaDeInt::início() {
    assert(cumpreInvariante());

    // Cria-se um iterador para esta lista:
    Iterador iterador(*this);

    iterador.índice_do_item_referenciado = -1;

    /* Em bom rigor não é boa ideia que seja um método da lista a verificar se a codição
       invariante de instância do iterador é verdadeira... Mais tarde se verá melhor
       forma de resolver o problema. */
    assert(iterador.cumpreInvariante());

    return iterador;
}

inline ListaDeInt::Iterador ListaDeInt::fim() {
    assert(cumpreInvariante());

    Iterador iterador(*this);

    iterador.índice_do_item_referenciado = número_de_itens;

    assert(iterador.cumpreInvariante());
```

```
    return iterador;
}

inline ListaDeInt::Iterador ListaDeInt::primeiro() {
    assert(cumpreInvariante());

    /* Cria-se um iterador para esta lista, que referencia inicialmente o item na frente
       da lista (ver construtor de ListaDeInt::Iterador), e devolve-se
       imediatamente o iterador criado: */
    return Iterador(*this);
}

inline ListaDeInt::Iterador ListaDeInt::último() {
    assert(cumpreInvariante());

    Iterador iterador(*this);

    iterador.índice_do_item_referenciado = número_de_itens - 1;

    assert(iterador.cumpreInvariante());

    return iterador;
}

inline bool ListaDeInt::cumpreInvariante() const {
    return 0 <= número_de_itens and
           número_de_itens <= número_máximo_de_itens;
}

inline ListaDeInt::Iterador::Iterador(ListaDeInt& lista_a_associar)
    : lista_associada(lista_a_associar),
      índice_do_item_referenciado(0) {
    assert(cumpreInvariante());
}

inline ListaDeInt::Item& ListaDeInt::Iterador::item() const {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.início() and
           *this != lista_associada.fim());

    return lista_associada.itens[índice_do_item_referenciado];
}

inline bool ListaDeInt::Iterador::
```

```
operator == (Iterador const& outro_iterador) const {
    assert(cumpreInvariante() and
           outro_iterador.cumpreInvariante());
    // assert(é válido and outro_iterador é válido);
    // assert(iteradores associados à mesma lista...);

    return índice_do_item_referenciado ==
           outro_iterador.índice_do_item_referenciado;
}

inline bool ListaDeInt::Iterador::
operator != (Iterador const& outro_iterador) const {
    assert(cumpreInvariante() and
           outro_iterador.cumpreInvariante());
    // assert(é válido and outro_iterador é válido);
    // assert(iteradores associados à mesma lista...);

    return not (*this == outro_iterador);
}

inline ListaDeInt::Iterador& ListaDeInt::Iterador::operator ++ () {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.fim());

    ++índice_do_item_referenciado;

    assert(cumpreInvariante());
    return *this;
}

inline ListaDeInt::Iterador ListaDeInt::Iterador::operator ++ (int) {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.fim());

    ListaDeInt::Iterador resultado = *this;
    operator ++ ();
    return resultado;
}

inline ListaDeInt::Iterador& ListaDeInt::Iterador::operator -- () {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.início());
```

```
    --índice_do_item_referenciado;

    assert(cumpreInvariante());
    return *this;
}

inline ListaDeInt::Iterador ListaDeInt::Iterador::operator -- (int) {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.início());

    ListaDeInt::Iterador resultado = *this;
    operator -- ();

    return resultado;
}

inline bool ListaDeInt::Iterador::cumpreInvariante() const {
    return -1 <= índice_do_item_referenciado and
        índice_do_item_referenciado <= lista_associada.número_de_itens;
}
```

H.1.3 Ficheiro de implementação: `lista_int.C`

Este ficheiro contém a função `main()` de teste do módulo `lista_int`. Contem também a definição de todas as rotinas e métodos que não são em-linha do módulo.

```
#include "lista_int.H"

void ListaDeInt::põeNaFrente(Item const& novo_item)
{
    assert(cumpreInvariante());
    assert(not estáCheia());

    for(int i = número_de_itens; i != 0; --i)
        itens[i] = itens[i - 1];

    itens[0] = novo_item;

    ++número_de_itens;

    assert(cumpreInvariante());
}

void ListaDeInt::insereAntes(Iterador& iterador,
```

```
                                Item const& novo_item)
{
    assert(cumpreInvariante());
    assert(not estáCheia());
    assert(iterador é válido);
    assert(iterador != início());

    // Há que rearranjar todos os itens a partir do referenciado pelo iterador:
    for(int i = número_de_itens;
        i != iterador.índice_do_item_referenciado;
        --i)
        itens[i] = itens[i - 1];

    // Agora já há espaço (não esquecer de revalidar o iterador!):
    itens[iterador.índice_do_item_referenciado++] = novo_item;

    assert(iterador.cumpreInvariante());

    // Mais um...
    ++número_de_itens;

    assert(cumpreInvariante());
}

void ListaDeInt::tiraDaFrente() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    --número_de_itens;
    for(int i = 0; i != número_de_itens; ++i)
        itens[i] = itens[i + 1];

    assert(cumpreInvariante());
}

void ListaDeInt::remove(Iterador& iterador) {
    assert(cumpreInvariante());
    assert(iterador é válido);
    assert(iterador != início() and iterador != fim());

    --número_de_itens;
    for(int i = iterador.índice_do_item_referenciado;
        i != número_de_itens;
        ++i)
        itens[i] = itens[i + 1];
```

```
    assert(cumpreInvariante());
}

#ifdef TESTE

// Macro definida para encurtar a escrita dos testes:
#define erro(mensagem)                              \
{                                                   \
    cout << __FILE__ << ":" << __LINE__ << ": " \
         << (mensagem) << endl;                     \
    ocorreram_erros = true;                         \
}

int main()
{
    bool ocorreram_erros = false;

    cout << "Testando módulo físico lista_int..." << endl;

    cout << "Testando classes ListaDeInt e ListaDeInt::Iterador..."
         << endl;
    // Definem-se itens canónicos para usar nos testes para que seja
    // fácil adaptar para tipos de itens diferentes:
    ListaDeInt::Item zero = 0;
    ListaDeInt::Item um = 1;
    ListaDeInt::Item dois = 2;
    ListaDeInt::Item tres = 3;
    ListaDeInt::Item quatro = 4;
    ListaDeInt::Item cinco = 5;
    ListaDeInt::Item seis = 6;
    ListaDeInt::Item sete = 7;
    ListaDeInt::Item oito = 8;
    ListaDeInt::Item nove = 9;
    ListaDeInt::Item dez = 10;

    int const número_de_vários = 11;
    ListaDeInt::Item vários[número_de_vários] = {
        zero, um, dois, tres, quatro, cinco,
        seis, sete, oito, nove, dez
    };

    ListaDeInt l;

    if(l.comprimento() != 0)
        erro("l.comprimento() dvia ser 0.");
```

```
l.põeAtrás(tres);
l.põeNaFrente(dois);
l.põeAtrás(quatro);
l.põeNaFrente(um);

if(l.comprimento() != 4)
    erro("l.comprimento() devia ser 4.");

if(l.frente() != um)
    erro("l.frente() devia ser um.");

if(l.trás() != quatro)
    erro("l.trás() devia ser um.");

ListaDeInt::Iterador i = l.início();

++i;

if(i != l.primeiro())
    erro("i devia ser l.primeiro().");

if(i.item() != um)
    erro("i.item() devia ser um.");

++i;

if(i.item() != dois)
    erro("i.item() devia ser dois.");

++i;

if(i.item() != tres)
    erro("i.item() devia ser tres.");

++i;

if(i.item() != quatro)
    erro("i.item() devia ser quatro.");

++i;

if(i != l.fim())
    erro("i devia ser l.fim().");

--i;
```

```
if(i != l.último())
    erro("i devia ser l.último().");

if(i.item() != quatro)
    erro("i.item() devia ser quatro.");

--i;

if(i.item() != tres)
    erro("i.item() devia ser tres.");

--i;

if(i.item() != dois)
    erro("i.item() devia ser dois.");

--i;

if(i.item() != um)
    erro("i.item() devia ser um.");

if(i != l.primeiro())
    erro("i devia ser l.primeiro().");

--i;

if(i++ != l.início())
    erro("i devia ser l.início().");

++i;

l.insereAntes(i, cinco);

if(i.item() != dois)
    erro("i.item() devia ser dois.");

--i;

if(i.item() != cinco)
    erro("i.item() devia ser cinco.");

i--;

l.remove(i);

if(i.item() != cinco)
```

```
    erro("i.item() devia ser cinco.");

if(--i != l.início())
    erro("i devia ser l.início().");

++i;
i++;
++i;
i++;

l.remove(i);

if(i != l.fim())
    erro("i devia ser l.fim().");

if(l.frente() != cinco)
    erro("l.frente() devia ser cinco.");

if(l.trás() != tres)
    erro("l.trás() devia ser tres.");

l.tiraDaFrente();

if(l.frente() != dois)
    erro("l.frente() devia ser dois.");

l.tiraDeTrás();

if(l.trás() != dois)
    erro("l.frente() devia ser dois.");

l.tiraDeTrás();

if(l.comprimento() != 0)
    erro("l.comprimento() devia ser 0.");

int número_de_colocados = 0;
while(número_de_colocados != número_de_vários and
      not l.estáCheia())
    l.põeAtrás(vários[número_de_colocados++]);

if(l.comprimento() != número_de_colocados)
    erro("l.comprimento() devia ser número_de_colocados.");

ListaDeInt::Iterador j = l.último();
while(not l.estáVazia()) {
```

```
        --número_de_colocados;

        if(l.trás() != vários[número_de_colocados])
            erro("l.trás() devia ser "
                 "vários[número_de_colocados].");

        if(j.item() != vários[número_de_colocados])
            erro("i.item() devia ser "
                 "vários[número_de_colocados].");

        --j;
        l.tiraDeTrás();
    }

    if(número_de_colocados != 0)
        erro("número_de_colocados devia ser 0.");

    if(j != l.início())
        erro("i devia ser l.início().");

    ++j;

    l.insereAntes(j, seis);

    --j;

    l.insereAntes(j, sete);

    l.insereAntes(j, oito);

    --j;

    l.insereAntes(j, nove);

    l.insereAntes(j, dez);

    l.remove(j);

    l.remove(j);

    --j;
    --j;
    --j;

    l.remove(j);
    l.remove(j);
```

```
    if(j.item() != dez)
        erro("j.item() devia ser dez.");

    if(l.trás() != dez)
        erro("l.trás() devia ser dez.");

    if(l.frente() != dez)
        erro("l.frente() devia ser dez.");

    if(l.comprimento() != 1)
        erro("l.comprimento() devia ser 1.");

    l.insereAntes(j, um);
    l.insereAntes(j, dois);
    l.insereAntes(j, tres);
    l.insereAntes(j, quatro);
    l.insereAntes(j, cinco);

    ListaDeInt const lc = l;

    l.esvazia();

    if(not l.estáVazia())
        erro("l.estáVazia() devia ser true.");

    if(lc.frente() != um)
        erro("lc.frente() devia ser um.");

    if(lc.trás() != dez)
        erro("lc.trás() devia ser dez.");

    cout << "Testes terminados." << endl;

    return ocorreram_erros? 1 : 0;
}

#endif // TESTE
```