Capítulo 10

Listas e iteradores

!!Falar de promiscuidade e intimidade!

Antes de se avançar para as noções de ponteiro e variáveis dinâmicas, estudadas no próximo capítulo, far-se-á uma digressão pelos conceitos de lista e iterador. Estes dois conceitos permitem uma considerável diversidade de implementações, todas com a mesma interface mas diferentes eficiências. Neste capítulo ir-se-á tão longe quanto possível no sentido de uma implementação apropriada dos conceitos de lista e iterador. No próximo capítulo continuar-se-á este exercício, mas já usando ponteiros e variáveis dinâmicas. Assim, a implementação eficiente de listas e iteradores surgirá como justificação parcial para essas noções.

O estudo das listas e dos iteradores associados tem algumas vantagens adicionais, tais como abordar ao de leve questões relacionadas com as estruturas de dados e a eficiência de algoritmos, bem como habituar o leitor à compreensão de classes com um grau já apreciável de complexidade.

10.1 Listas

Todos nós conhecemos o significado da palavra "lista", pelo menos intuitivamente. As listas ocorrem na nossa vida prática em muitas circunstâncias: lista de compras, de afazeres, de pessoas... O Novo Aurélio [4] define lista da seguinte forma:

> **lista.** [Do fr. *liste* < it. *lista* < germ. **lista* (al. mod. *Leiste*).] ***S. f.*** **1.** Relação de nomes de pessoas ou de coisas: relação, rol, listagem. [...]

Escusado será dizer que as definições de "relação", "rol" e "listagem" do Novo Aurélio remetem de volta à noção de lista[1]! Tentar-se-á aqui dar uma definição do conceito de lista um pouco menos genérica.

Suponha-se uma lista de tarefas a realizar por um determinado aluno mais ou menos aplicado e com alguma ideia das prioridades:

[1] É uma fatalidade que tal aconteça num dicionário. Mas esperamos sempre que os ciclos de definições não sejam tão curtos...

- Comprar CD dos *Daft Punk*.
- Combinar ida ao concerto *Come Together*.
- Fazer trabalho de Sistemas Operativos.
- Estudar Programação Orientada para Objectos.

Uma lista parece portanto corresponder simplesmente a um *conjunto* de itens. No entanto, haverá alguma razão para, tal como acontece nos conjuntos, não existirem repetições de itens? Na realidade não. Note-se na seguinte lista de afazeres, bastante mais razoável que a anterior:

- Estudar Programação Orientada para Objectos.
- Comprar CD dos *Daft Punk*.
- Estudar Programação Orientada para Objectos.
- Combinar ida ao concerto *Come Together*.
- Estudar Programação Orientada para Objectos.
- Fazer trabalho de Sistemas Operativos.
- Estudar Programação Orientada para Objectos.

Assim, melhor seria dizer que uma lista é uma *colecção* de itens. Porém, a noção de colecção não é suficiente, pois ignora um factor fundamental: os itens de uma lista têm uma determinada ordem, que não tem forçosamente a ver com os valores dos itens (como é evidente no último exemplo)[2].

Assim, como definição tentativa, pode-se dizer que uma lista é uma sequência de itens com ordem arbitrária mas relevante.

Naturalmente que uma definição apropriada de lista terá de incluir as operações que se podem realizar com listas. Aliás, a noção de lista (e de iterador) podem ser totalmente definidas à custa das respectivas operações.

10.1.1 Operações com listas

A operação fundamental a realizar com qualquer tipo abstracto de dados é naturalmente a construção de instâncias desse tipo. Depois de construída uma lista, deve ser possível pôr novos itens na lista e tirar itens existentes da lista, aceder aos itens, e obter informação geral acerca do estado da lista. Grosso modo, as operações a realizar são:

- Construir nova lista.

[2] A ordem é normalmente determinada pela forma como os seus itens foram colocados na lista.

- Pôr novo item algures na lista. A lista não pode estar cheia, se tiver limite pré-estabelecido.
- Tirar algum item existente da lista. A lista não pode estar vazia.
- Aceder a um qualquer item da lista. A lista não pode estar vazia.
- Saber comprimento da lista.
- Saber se a lista está vazia.
- Saber se está cheia, se tiver limite pré-estabelecido.

Esta listagem de operações deixa um problema por resolver: pôr itens, tirar itens e aceder a itens implica ser capaz de lhes especificar a posição na lista. Naturalmente que uma possibilidade seria indicar as posições através de um número de ordem, por exemplo numerando os itens a partir de zero desde o início da lista, e usar depois índices para especificar posições, como se faz usualmente nas matrizes e vectores. Esta solução tem, no entanto, um problema. Esse problema só se tornará verdadeiramente claro quando se fizer uma implementação eficiente do conceito de lista em secções posteriores e tem a ver com a eficiência dos seus métodos: quando se melhorar a implementação inicial das listas (ainda por fazer...) de modo a permitir inserções eficientes de novos itens *em qualquer posição*, concluir-se-á que a utilização de índices se tornará extremamente ineficiente...

Porquê esta preocupação com a eficiência quando se está ainda a definir a interface das listas, ou seja, as suas operações e respectivos efeitos? Acontece que se pode e deve considerar que a eficiência faz parte do contrato das rotinas e métodos. Embora a questão da eficiência de algoritmos esteja fora do contexto desta disciplina, é fundamental apresentar aqui pelo menos uma ideia vaga destas ideias.

Suponha-se que se pretende desenvolver um procedimento para ordenar os itens de um vector por ordem crescente. A interface do procedimento inclui, como se viu em capítulos anteriores, o seu cabeçalho, que indica como o procedimento se utiliza, e um comentário de documentação onde se indicam a pré-condição e a condição objectivo do procedimento, ou seja o contrato estabelecido entre o programador que o produziu e os programadores que o consumirão (ver Secção 9.3.1):

```
/** Ordena os valores do vector v.
    @pre PC ≡ v = v.
    @post CO ≡ perm(v, v) ∧ (Q i, j : 0 ≤ i ≤ j < v.size() : v[i] ≤ v[j]).
void ordenaPorOrdemNãoDecrescente(vector<int>& v);
```

No entanto, esta interface não diz tudo. Se se mandar ordenar um vector com 1000 itens, quanto tempo demora o procedimento? E se forem o dobro, 2000 itens? Como cresce o tempo de execução com o número de itens? E quanta memória adicional usará o procedimento?

Estas questões são muitas vezes relevantes, embora nem sempre. Podem fazer a diferença entre um programa que demora dias ou meses (ou anos, séculos ou milénios...) a executar e outro programa que demora segundos. Ou entre um programa que pode ser executado em computadores com memória limitada e outro com um apetite voraz pela memória que o torna inútil na prática.

As eficiências em termos de tempo de execução e em termos de espaço de memória consumido são muito importantes e podem e devem ser considerados parte do contrato de uma rotina ou método de uma classe. No caso do procedimento de ordenação a interface, em rigor, deveria ser melhor especificada:

```
/** Ordena os valores do vector v. O tempo de execução cresce com número n
    de itens ao mesmo ritmo que a função n ln n. Ou seja, a eficiência temporal é
    O(n ln(n)). A ordenação é feita directamente sobre a matriz, recorrendo-se
    apenas a um pequeno número de variáveis auxiliares.
    @pre  PC ≡ v = v.
    @post CO ≡ perm(v, v) ∧ (Q i, j : 0 ≤ i ≤ j < v.size() : v[i] ≤ v[j]).
void ordenaPorOrdemNãoDecrescente(vector<int>& v);
```

É importante perceber-se que do contracto não fazem parte tempos de execução ao segundo. Há duas razões para isso. A primeira é prática: os tempos exactos de execução não são fáceis de calcular e, sobretudo, variam de computador para computador e de máquina para máquina. A segunda razão prende-se com o facto de ser muito mais importante saber que o tempo de execução ou a memória requerida por um algoritmo crescem de uma forma "bem comportada" do que saber o tempo exacto de execução[3].

Esta digressão veio como justificação para a utilização de critérios de eficiência na exclusão dos índices para indicar localizações de itens em listas. É que se pretende que as listas tenham métodos de inserção de novos itens em locais arbitrários (e respectiva remoção) tão eficientes quanto a sua inserção em (ou remoção de) locais "canónicos", tais como os seus extremos. Isso, como se verá, levará a uma "arrumação" dos itens tal que a sua indexação será inevitavelmente muito ineficiente.

Claro que uma possibilidade de resolução deste problema seria fornecer a possibilidade de usar índices para referenciar posições de itens na lista mas assinalar claramente como ineficientes as respectivas operações. No entanto, tal não será feito aqui, até porque se irá desenvolver uma forma mais apropriada de referenciar os itens de uma lista.

Há que distinguir entre as operações realizadas em locais "canónicos" das listas, que são os seus extremos, e aquelas que se realizam em locais arbitrários, que terão de fazer uso de uma generalização do conceito de indexação, a introduzir na próxima secção. As operações de inserção, remoção e acesso em locais canónicos são:

- Tirar o item da *frente* da lista. A lista não pode estar vazia.

[3] A título de exemplo, suponham-se dois algoritmos possíveis para a implementação do procedimento de ordenação acima. Suponha-se que o primeiro, mais simples, é caracterizado por o tempo de execução crescer com o quadrado do número de itens a ordenar e que o segundo, um pouco mais complicado, tem tempos de execução que crescem com o produto do número de itens a ordenar pelo seu logaritmo. A notação da algoritmia para a eficiência temporal destas algoritmos é $O(n^2)$ e $O(n \log n)$, respectivamente, em que n é o número de itens a ordenar. Cálculos muito simples permitem verificar que, para um número suficientemente grande de itens, o algoritmo mais complicado acaba sempre por ter uma eficiência superior.

- Tirar o item de *trás* da lista. A lista não pode estar vazia.
- Pôr um novo item na frente da lista. A lista não pode estar cheia, se tiver limite pré-estabelecido.
- Pôr um novo item na traseira da lista. A lista não pode estar cheia, se tiver limite pré-estabelecido.
- Aceder ao item na frente da lista (para modificação ou não). A lista não pode estar vazia.
- Aceder ao item na traseira da lista (para modificação ou não). A lista não pode estar vazia.

É possível imaginar muitas mais operações com listas. Algumas delas são realmente úteis, mas só serão introduzidas mais tarde, de modo a manter uma complexidade limitada na primeira abordagem. Outras serão porventura menos úteis em geral e poderão ser dispensadas. Não é fácil, como é óbvio, distinguir operações essenciais de operações acessórias. Algumas são essenciais para que se possa dar o nome de lista ao tipo em análise. As operações escolhidas neste e no próximo capítulo incluem as fundamentais para que se possa de facto falar em listas (como as operações de inserção e remoção de itens em locais canónicos) e também as mais comummente utilizadas na prática. Por outro lado, correspondem também a algumas das operações existentes na interface do tipo genérico `list` da biblioteca padrão do C++[4]. É o caso, por exemplo da seguinte operação, que fará parte da interface das listas:

- Esvaziar a lista, tirando todos os seus itens.

10.2 Iteradores

Como fazer para indicar um local arbitrário de inserção, remoção ou simples acesso a itens da lista? É conveniente aqui observar como um humano resolve o problema. Como o único humano disponível neste momento é o leitor, peço-lhe que entre no jogo e colabore. Considere a seguinte lista de inteiros

$$(1\ 2\ 3\ 11\ 20\ 0\ 354\ 2\ 3\ 45\ 12\ 34\ 30\ 4\ 4\ 23\ 3\ 77\ 4\ \ -1\ \ -20\ 46).$$

Considere um qualquer item da lista e suponha que pretende indicá-lo a um amigo. Suponha que esse amigo está ao seu lado e pode ver a lista. Indique-lhe o item que escolheu.

$\vdots$

$\vdots$

[4]Ou seja, reinventa-se de novo a roda... Ver Nota 1 na página 301.

Provavelmente o leitor indicou-o da forma mais óbvia: apontando-o com o dedo indicador... Pelo menos é assim que a maioria das pessoas faz... O objectivo agora é, pois, conseguir representar o conceito de "dedo indicador" em C++...

E se o leitor quiser indicar o local para inserir um novo item na lista? Provavelmente fá-lo-á indicando um intervalo entre dois itens (excepto se pretender inserir num dos extremos da lista). Aqui, no entanto, usar-se-á uma abordagem diferente: é sempre possível indicar um local de inserção indicando um item existente e dizendo para inserir o novo item imediatamente antes ou depois do item indicado. Isto, claro está, se a lista não estiver vazia! Mas mesmo nesse caso se encontrará uma solução interessante.

Para além da lista em si, tem-se agora um outro conceito para representar. Bons nomes para o novo conceito poderiam ser "indicador" (sugestivo dada a analogia do dedo) ou "cursor". Porém, o termo aqui usado é o usual em programação: *iterador*.

10.2.1 Operações com iteradores

Que operações se podem realizar com os iteradores? A primeira operação é, naturalmente, construir um iterador. É razoável impor que um iterador tenha de estar sempre *associado* a uma determinada lista. Assim, a construção de um iterador associa-o a uma lista e põe-no a *referenciar* um determinado item de essa lista, por exemplo o da sua frente (se existir...).

Imaginem-se agora as operações que se podem realizar com um dedo sobre uma lista. Identificam-se imediatamente as operações mais elementares: avançar e recuar[5] o iterador e aceder ao item referenciado pelo iterador. Por razões que se verá mais tarde, não será eficiente colocar um iterador numa posição arbitrária de uma lista (se exceptuarmos as posições canónicas) nem avançar ou recuar um iterador mais do que um item de cada vez. Estas restrições têm menos a ver com o conceito de iterador em si do que com o tipo de contentor de itens a que está associado[6]. Ou seja, as operações a realizar com iteradores são:

- Construir iterador associado a lista e referenciando o item na sua frente.
- Avançar o iterador. O iterador não pode referenciar o item de trás da lista.
- Recuar o iterador. O iterador não pode referenciar o item na frente da lista.
- Aceder ao item referenciado pelo iterador (para modificação ou não).

[5]Usa-se o termo "avançar" para deslocamentos da frente para trás e o termo "recuar" para deslocamentos de trás para a frente. Isto pode parecer estranho, mas é prática habitual e fará um pouco mais de sentido quando associado aos termos "início" e "fim". A ideia é que avançar é deslocar do início para o fim, tal como o leitor desloca o olhar, avançando, do início para o fim deste texto...

[6]Um *contentor* é um tipo abstracto de dados capaz de conter itens de um outro tipo. Existem variadíssimos contentores, que incluem listas, filas, pilhas, vectores, matrizes, colecções, conjuntos, mapas, etc. Cada um destes tipos de contentores (com excepção das pilhas e das filas) pode ser associado a um tipo específico de iterador. Os iteradores podem ser categorizados em *conceitos*, de acordo com as características específicas do respectivo contentor !!!!citarAustern. Por exemplo, vectores têm associados iteradores ditos de acesso aleatório, muito menos restrivos que os das listas, ditos bidireccionais.

Quantos iteradores se podem associar a uma lista? A solução mais restritiva é permitir apenas um iterador por lista. Esta solução, que não será seguida aqui, tem a vantagem de não exigir a definição de classes auxiliares e de resolver bem o problema da validade dos iteradores depois de alterações à lista que lhe está associada. A solução aqui seguida será a mais genérica: poderão existir um número arbitrário de iteradores associados à mesma lista.

A existência de vários iteradores associados à mesma lista torna necessários os operadores de igualdade e diferença entre iteradores:

- Verificação de igualdade entre iteradores. Os iteradores têm de estar associados à mesma lista.
- Verificação da diferença entre iteradores. Os iteradores têm de estar associados à mesma lista.

10.2.2 Operações se podem realizar com iteradores e listas

Para além das operações realizadas sobre os iteradores, há algumas operações que são realizadas sobre uma lista mas usando um iterador como indicador de uma posição arbitrária. Estas operações são as de inserção e remoção de itens em e de locais arbitrários mencionadas mais atrás. Por outro lado, era conveniente que a lista possuísse métodos construtores de iteradores para as suas posições canónicas.

Relativamente às operações de inserção e remoção, há que tomar algumas decisões.

Em primeiro lugar tem de se decidir se se insere antes ou depois do item referenciado por um iterador. Isto deve-se a que os iteradores referenciam itens, e não intervalos entre itens. Poder-se-ia escolher fornecer duas operações, uma inserindo antes e outra depois. Na prática considera-se suficente fornecer uma versão que insira antes de um iterador, pois durante inserções ordenadas é típico o iterador parar logo que se encontra o primeiro item à direita do qual o novo item não pode ser colocado (ver mais abaixo).

Em segundo lugar tem de se decidir o que fazer a um iterador depois da remoção do item por ele referenciado. É usual escolher-se avançar o iterador para o item imediatamente a seguir.

Ambas as opções, i.e., inserir antes e avançar ao remover, são típicas, embora razoavelmente arbitrárias.

Relativamente às posições canónicas, chamar-se-á *primeiro* a um iterador referenciando o item na frente da lista e *último* a um iterador referenciando o item na traseira da lista. Assim, as operações são:

- Inserir novo item antes do item referenciado por um iterador.
- Remover item referenciado por um iterador (e avançá-lo).
- Construir primeiro iterador, i.e., um iterador referenciando o item na frente da lista.
- Construir último iterador, i.e., um iterador referenciando o item na traseira da lista.

10.2.3 Itens fictícios

Para terminar a discussão acerca das operações de listas e iteradores há que resolver ainda alguns problemas. Nas operações apresentadas até agora teve-se sempre o cuidado de apresentar as respectivas pré-condições. Mas houve alguns "esquecimentos":

- Qual é o primeiro iterador de uma lista vazia? E o último?
- Que item referencia um iterador depois de, através dele, se ter removido o item da traseira da lista?

Outras perguntas igualmente interessantes podem ser feitas:

- Que item referencia um iterador associado a uma lista vazia?
- Como se insere na traseira da lista, através de um iterador, se existe apenas uma operação para inserir *antes* da posição referenciada por um iterador?
- Como se percorre uma lista do princípio ao fim com um iterador?

Para situar um pouco melhor estas perguntas e justificar uma possível resposta, suponha-se o problema de inserir ordenadamente um novo item numa lista de inteiros

$$\texttt{lista} = (1\ 2\ 6\ 9).$$

Suponha-se que o item a inserir é 5. Que fazer? Que algoritmo se pode usar que resolva o problema para qualquer lista ordenada e qualquer novo item?

Uma possibilidade é fazer uma pesquisa sequencial desde o item na frente da lista até encontrar o primeiro item maior ou igual ao novo item e inseri-lo imediatamente antes do item encontrado. Ou seja,

{ Algoritmo para inserir ordenadamente o novo item `novo_item` na lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto item referenciado por `i < novo_item` **faça-se:**
    avançar `i`
inserir `novo_item` na lista `lista` imediatamente antes do item referenciado por `i`

E se o novo item for 10? É claro que o algoritmo anterior não está correcto. A sua guarda tem de permitir a paragem mesmo que todos os itens da lista sejam inferiores ao novo item a inserir:

{ Algoritmo para inserir ordenadamente o novo item `novo_item` na lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto `i` não atingiu o fim da lista `lista` ∧
        item referenciado por `i` < `novo_item` **faça-se:**
    avançar `i`
inserir `novo_item` na lista `lista` imediatamente antes do item referenciado por `i`

O problema é o significado da frase "`i` não atingiu o fim da lista `l`". Que significa o fim da lista? Não é decerto o iterador referenciando o item na traseira. Porquê? Porque nesse caso o algoritmo que segue, que era suposto mostrar todos os itens da lista, pura e simplesmente não funciona: deixa o item de trás da lista de fora...

{ Algoritmo para mostrar todos os itens da lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto `i` não atingiu o fim da lista `lista` **faça-se:**
    mostrar item referenciado por `i`
    avançar `i`

Conclui-se facilmente que, a haver um iterador referenciando o fim da lista, ele está para além do último! Mas nesse caso, que item referencia? A esse item, que na realidade não existe na lista, dar-se-á o nome de item *fictício*. Assim, pode-se considerar que as listas, para além dos seus itens, possuem dois itens fictícios nos seus extremos. Estes itens são particulares por várias razões:

1. Existem sempre, mesmo com a lista vazia.
2. Não correspondem a itens reais, pelo que não têm valor.
3. São úteis apenas enquanto âncoras para os iteradores, que se lhes podem referir.

Chamar-se-á aos iteradores antes do primeiro e depois do último respectivamente *início* e *fim*. Veja-se a situação retratada na Figura 10.1.

Na realidade a noção de item fictício foi usada já extensamente quando se desenvolveram ciclos para percorrer matrizes. Repare-se no seguinte troço de programa:

```
double md[4] = {1.1, 2.2, 3.3, 4.4};

for(int i = 0; i != 4; ++i)
    cout << md[i] << endl;
```

Neste ciclo o índice toma todos os valores entre 0 e 4, sendo 4 a dimensão da matriz e, portanto, o índice de um elemento fictício final da matriz. Da mesma forma se pode considerar que uma matriz tem um item fictício inicial:

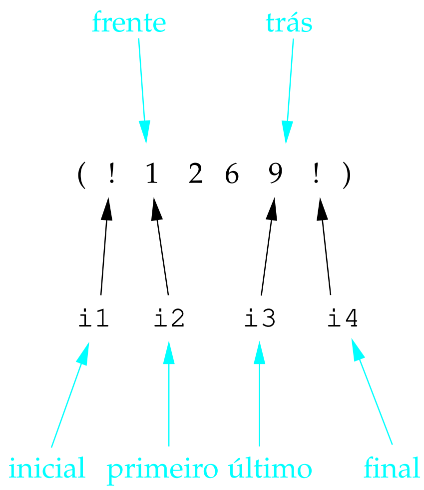

Figura 10.1: Definição de iteradores e itens canónicos das listas. Representam-se por um ponto de exclamação os itens fictícios da lista. Os iteradores primeiro (`i2`) e último (`i3`) referenciam respectivamente os itens na frente e na traseira da lista. Os iteradores início (`i1`) e fim (`i4`) são definidos como sendo respectivamente anterior ao primeiro e posterior ao último.

```
double md[4] = {1.1, 2.2, 3.3, 4.4};

for(int i = 3; i != -1; --i)
    cout << md[i] << endl;
```

Voltando às listas, e dados os dois novos iteradores início e fim, são úteis duas novas operações de construção para os obter:

- Construir iterador início, i.e., o iterador antes do primeiro.
- Construir iterador fim, i.e., o iterador depois do último.

Com estes novos iteradores, torna-se claro que os algoritmos propostos estão correctos. No caso da inserção ordenada, mesmo que o novo item seja superior a todos os itens da lista, o ciclo termina com o iterador final, pelo que inserir o novo item antes da posição referenciada pelo iterador leva-o a ficar na traseira da lista, como apropriado! É também interessante compreender que o posicionamento dos iteradores no caso de uma lista vazia, tal como indicado na Figura 10.2, apesar de contra-intuitivo, leva também ao correcto funcionamento dos algoritmos nesse caso extremo.

10.2.4 Operações que invalidam os iteradores

Que acontece a um iterador referenciando um item não-fictício de uma dada lista se essa lista for esvaziada? É evidente que esse iterador não pode continuar válido. Assim, é importante reconhecer que os iteradores podem estar em estados inválidos, nos quais a sua utilização é um erro. A questão mais importante associada à validade dos iteradores é a de saber que

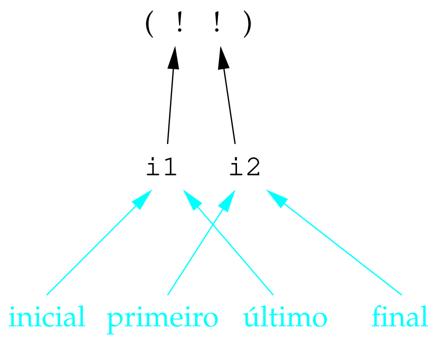

Figura 10.2: Definição de iteradores e itens canónicos de uma lista vazia. Note-se que os iteradores primeiro e último trocam as suas posições relativas usuais. Decorre daqui que são estes iteradores que se definem à custa dos iteradores início e fim, e não o contrário, como se sugeriu atrás.

operações invalidam que iteradores associados a uma lista. Esta questão e as pré-condições das operações discutidas apresentam-se na secção seguinte.

10.2.5 Conclusão

Operações apenas das listas:

- Construtoras:
  - Construir nova lista vazia.
- Inspectoras:
  - Saber comprimento da lista, i.e., quantos itens contém.
  - Saber se a lista está vazia.
  - Saber se está cheia, se tiver limite pré-estabelecido.
  - Saber valor do item na frente da lista. $PC \equiv$ a lista não pode estar vazia.
  - Saber valor do item na traseira da lista. $PC \equiv$ lista não pode estar vazia.
- Modificadoras:
  - Aceder ao item na frente da lista (para possível modificação). $PC \equiv$ a lista não pode estar vazia. Não invalida iteradores.
  - Aceder ao item na traseira da lista (para possível modificação). $PC \equiv$ a lista não pode estar vazia. Não invalida iteradores.
  - Pôr um novo item na frente da lista. $PC \equiv$ a lista não pode estar cheia, se tiver limite pré-estabelecido. Não invalida iteradores.
  - Pôr um novo item na traseira da lista. $PC \equiv$ a lista não pode estar cheia, se tiver limite pré-estabelecido. Não invalida iteradores.

  - Tirar o item da frente da lista. $PC \equiv$ a lista não pode estar vazia. Invalida todos os iteradores que referenciem o item da frente da lista.
  - Tirar o item de trás da lista. $PC \equiv$ a lista não pode estar vazia.Invalida todos os iteradores que referenciem o item de trás da lista.
  - Esvaziar a lista, tirando todos os seus itens. Invalida todos os iteradores associado à lista, com excepção dos que referenciam os seus itens fictícios.

Em todas as operações relacionadas com iteradores (exceptuando as construtoras de iteradores) considera-se como pré-condição que os iteradores têm de ser válidos. Esta condição não se apresenta explicitamente abaixo.

Operações dos iteradores:

- Construir iterador associado a lista (iterador é o primeiro, i.e., se a lista estiver vazia é o seu fim).
- Avançar o iterador. $PC \equiv$ o iterador não pode ser o fim da lista.
- Recuar o iterador. $PC \equiv$ o iterador não pode ser o início da lista.
- Aceder ao item referenciado pelo iterador (para possível modificação). $PC \equiv$ o iterador não pode ser nem o início nem o fim da lista.
- Verificação de igualdade entre iteradores. $PC \equiv$ os iteradores têm de estar associados à mesma lista.
- Verificação da diferença entre iteradores. $PC \equiv$ os iteradores têm de estar associados à mesma lista.

Operações das listas relacionadas com iteradores:

- Inserir novo item antes do item referenciado por um iterador. $PC \equiv$ o iterador não pode ser o início da lista. Nâo invalida iteradores.
- Remover item referenciado por um iterador (e avançá-lo).$PC \equiv$ o iterador não pode ser nem o início nem o fim da lista. Invalida todos os iteradores que referenciem o item removido (com excepção do iterador que foi usado para a remoção, que avança).
- Construir primeiro iterador, i.e., o iterador depois do início da lista.
- Construir último iterador, i.e., o iterador antes do fim da lista.
- Construir iterador início.
- Construir iterador fim.

10.3 Interface

Os conceitos de lista e iterador, para serem verdadeiramente úteis, têm de ser concretizados na forma de classes. As operações estudadas nas secções anteriores correspondem à sua interface. Para simplificar o desenvolvimento supor-se-á que as listas guardam inteiros. Assim, chamar-se-á `ListaDeInt` à classe que concretiza o conceito de lista de inteiros e `Iterador` à classe que concretiza o conceito de iterador de uma lista de inteiros.

Nas secções abaixo vai-se definindo a interface da classe passo por passo. Os contractos das operações declaradas são paresentados informalmente. No final é apresentada uma veroa completa, com os contratos das operações na forma de comentários de documentação.

10.3.1 Interface de `ListaDeInt`

Esta classe é a concretização do conceito de lista de inteiros:

```
class ListaDeInt {
  public:
```

Tipos

Há dois tipos definidos pela classe. O primeiro, `Item`, não é verdadeiramente um novo tipo, pois `Item` é sinónimo de `int`:

```
typedef int Item;
```

Este sinónimo tem a vantagem de simplificar consideravelmente a tarefa de criar listas com itens de outros tipos.

!!!!Dar maior ênfase a isto!

O segundo tipo é a classe `Iterador`:

```
class Iterador;
```

!!!!Reescrever

De modo a que o identificador `Iterador` não fique gasto, uma vez que se podem definir iteradores para muitos outros tipos de contentores para além das listas, é conveniente que a classe `Iterador` seja embutida dentro da classe `ListaDeInt`.

Métodos construtores

Declara-se apenas um método construtor que constrói uma lista vazia:

```
ListaDeInt();
```

Métodos inspectores

Declaram-se três métodos inspectores. Este inspectores podem ser usados em qualquer circunstância para indagar do estado da lista.

O primeiro devolve o número de itens da lista, ou seja, o seu comprimento:

```
int comprimento() const;
```

O segundo serve para verificar se a lista está vazia:

```
bool estáVazia() const;
```

O terceiro serve para verificar se a lista está cheia:

```
bool estáCheia() const;
```

Declaram-se adicionalmente dois métodos inspectores que apenas podem ser invocados se a lista não estiver vazia e que servem para aceder (sem poder modificar) aos itens nas posições canónicas da lista. Ambos têm como pré-condição:

$$PC \equiv \neg\texttt{estáVazia}().$$

O primeiro devolve uma referência constante para o item na frente da lista:

```
Item const& frente() const;
```

O segundo devolve uma referência constante para o item na traseira da lista:

```
Item const& trás() const;
```

Métodos modificadores

Os dois primeiros métodos modificadores declarados são semelhantes aos inspectores dos itens nas posições canónicas da lista. Tal como eles, apenas podem ser invocados se a lista não estiver vazia

$$PC \equiv \neg\texttt{estáVazia}().$$

Ao contrário deles, no entanto, devolvem referências para os itens, o que permite que estes sejam modificados. O primeiro devolve um referência para o item na frente da lista:

```
Item& frente();
```

O segundo devolve uma referência para o item na traseira da lista:

```
Item& trás();
```

Os dois métodos modificadores seguintes permitem pôr um novo item nas posições canónicas da lista. Ambos requerem que a lista não esteja cheia, i.e.,!!!

$$PC \equiv \neg\texttt{estáCheia}().$$

!!!!!Vou aqui

O primeiro dos métodos põe o novo item na frente da lista:

```
void põeNaFrente(Item const& novo_item);
```

O segundo põe o novo item na traseira da lista:

```
void põeAtrás(Item const& novo_item);
```

Declaram-se também dois métodos modificadores que permitem tirar um item das posições canónicas da lista. Ambos requerem que a lista não esteja vazia, i.e.,

$$PC \equiv \neg\texttt{estáVazia}().$$

Ambos invalidam qualquer iterador que esteja associado à lista.

O primeiro dos métodos tira o item da frente da lista:

```
void tiraDaFrente();
```

O segundo tira o item da traseira da lista:

```
void tiraDeTrás();
```

Finalmente, declara-se um método modificador de que não se falou ainda. Este método serve para esvaziar a lista, i.e., para descartar todos os seus itens:

```
void esvazia();
```

Este método invalida qualquer iterador que esteja associado à lista.

Métodos modificadores envolvendo iteradores

Declaram-se dois métodos modificadores que envolvem a especificação de posições através de iteradores.

O primeiro insere um novo item imediatamente antes da posição indicada por um iterador `iterador`, exigindo-se para isso que a lista não esteja cheia, que o iterador seja válido e esteja associado à lista em causa e que o iterador não referencie o iterador início[7]

$PC \equiv$ `iterador` é válido$\wedge$`iterador` associado à lista$\wedge\neg$`estáCheia`()$\wedge$`iterador` $\neq$ `início`().

Este método garante que o iterador continua a referenciar o mesmo item que antes da inserção[8]:

```
void insereAntes(Iterador& iterador, Item const& novo_item);
```

Qualquer outro iterador associado à lista é invalidado por este método.

O segundo método remove o item indicado pelo iterador iterador, exigindo-se que este seja válido e esteja associado à lista em causa e que referencie um dos itens válido da lista, pois não faz sentido remover os itens fictícios[9]

$PC \equiv$ `iterador` é válido$\wedge$`iterador` associado à lista$\wedge$`iterador` $\neq$ `início`()$\wedge$`iterador` $\neq$ `fim`().

Este método garante que o iterador fica a referenciar o item logo após o item removido. Por isso o iterador tem de ser passado por referência não-constante.

```
void remove(Iterador& iterador);
```

Qualquer outro iterador associado à lista é invalidado por este método.

[7] Ver declaração do método `início()` mais à frente. Repare-se que não se incluiu na pré-condição nenhum termo que garante que o iterador está de facto associado à lista em causa. Isso será feito mais tarde, quando se falar de ponteiros.

[8] Há aqui uma incoerência que o leitor mais atento detectará. Se o iterador se mantém referenciando o mesmo item que antes da inserção, não deveria ser passado por valor ou referência constante? É verdade que sim. Acontece que a primeira implementação simplista destas classes, que será feita na Secção 10.4, implicará uma alteração física ao iterador para ele conceptualmente se possa manter constante... Este tipo de problemas resolve-se normalmente à custa do qualificador `mutable`, que aplicado a uma variável membro de uma classe, permite que ela seja alterada mesmo que a instância que a contém seja constante. No entanto essa solução levaria a uma maior complexidade da classe e, além disso, o problema resolver-se-á naturalmente quando se melhorar a implementação da classe na Secção 10.5, pelo que mais tarde a declaração do método será mudada paulatinamente de modo ao iterador ser passado por referência constante...

[9] Note-se que não é necessário incluir na pré-condição a garantia de que a lista não está vazia, visto que se o iterador não se refere a nenhum dos itens fictícios, então certamente que a lista não está vazia.

Métodos construtores de iteradores

Estes métodos consideram-se modificadores porque a lista pode ser modificada através dos iteradores devolvidos. Existem quatro métodos construtores de iteradores, correspondentes aos quatro iteradores canónicos definidos na Figura 10.1. Os primeiros métodos devolvem iteradores referenciando os itens fictícios da lista:

```
    Iterador início();
    Iterador fim();
```

Os dois seguintes devolvem iteradores imediamente depois do início e antes do fim, i.e., nas posições conhecidas por "primeiro" e "último":

```
    Iterador primeiro();
    Iterador último();
```

Com é evidente os iteradores construídos por qualquer destes métodos são válidos.

Estes dois métodos completam a interface da classe `ListaDeInt`. Mais tarde, quando se passar à implementação, discutir-se-ão os membros privados da classe:

```
  private:
    ... // a completar mais tarde, pois é parte da implementação da classe.
};
```

Interface completa em C++

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Que fazer ao contrato? Convinha apresentá-lo!

!!!!Ir buscar tudo de novo aos guiões. Não esquecer de acrescentar invalidação de iteradores nos contratos das operações!

```
class ListaDeInt {
  public:
    typedef int Item;

    class Iterador;

    ListaDeInt();

    int comprimento() const;

    bool estáVazia() const;
    bool estáCheia() const;
```

```
        Item const& frente() const;
        Item const& trás() const;

        Item& frente();
        Item& trás();

        void põeNaFrente(Item const& novo_item);
        void põeAtrás(Item const& novo_item);

        void tiraDaFrente();
        void tiraDeTrás();

        void esvazia();

        void insereAntes(Iterador& iterador, Item const& novo_item);
        void remove(Iterador& iterador);

        Iterador início();
        Iterador fim();

        Iterador primeiro();
        Iterador último();

      private:
        ... // a completar mais tarde, pois é parte da implementação da classe.
    };
```

10.3.2 Interface de **ListaDeInt::Iterador**

Esta classe, `ListaDeInt::Iterador`, é a concretização do conceito de iterador de uma lista de inteiros:

```
    class ListaDeInt::Iterador {
      public:
```

Métodos construtores

Esta classe tem apenas um construtor. Como um iterador tem de estar sempre associado a uma qualquer lista, é natural que o construtor receba a lista como argumento. O iterador construído fica válido e associado à lista passada como argumento e referenciando o item na sua frente. I.e., o iterador construído é o primeiro da lista. Como através do iterador se podem alterar os itens da lista, a lista é passada por referência não-constante:

```
        explicit Iterador(ListaDeInt& lista_a_associar);
```

Note-se a utilização da palavra-chave `explicit`. A sua intenção é impedir que o construtor, podendo ser invocado com um único argumento, introduza uma conversão implícita de lista para iterador. Uma instrução como

```
ListaDeInt lista;
...
ListaDeInt i(lista);
...
if(i == lista) // comparação errónea se não houver conversão implícita.
    ...
```

resulta portanto num erro de compilação, ao contrário do que sucederia se a conversão implícita existisse[10].

Métodos inspectores

Declara-se um único método inspector, que serve para aceder ao item referenciado pelo iterador. Este método devolve uma referência para o item referenciado pelo iterador, de modo a que seja possível alterá-lo. É importante perceber-se que este método é constante, apesar de permitir alterações ao item referenciado: a constância de um iterador refere-se ao iterador em si, e não à lista associada ou aos itens referenciados, que não pertencem ao iterador, mas sim à lista associada[11]. Um iterador constante (`const`) não pode ser alterado (e.g., avançar ou recuar), mas permite alterar o item por ele referenciado na lista associada.

O método requer que o iterador seja válido e não seja nem o início nem o fim da lista, pois não faz sentido aceder aos itens fictícios

$$PC \equiv \text{ é válido} \wedge \text{não é iterador inicial da lista} \wedge \text{não é iterador final da lista.}$$

A declaração do método é:

```
Item& item() const;
```

Operadores inspectores

Declaram-se dois operadores inspectores, i.e., dois operadores que não modificam a instância implícita, que é sempre o seu primeiro operando. São os operadores de igualdade (==) e diferença (`!=`). Estes operadores são membros da classe porque exigem acesso aos atributos privados da classe[12].

[10]Na realidade essa conversão poderia contribuir para aproximar o modelo das listas e iteradores do modelo de matrizes e ponteiros do C++. Ver Capítulo 11.

[11]Este facto, de resto, irá obrigar mais tarde à definição de uma classe adicional de iterador, menos permissiva, que garanta a constância da lista associada e dos itens referenciados.

[12]Uma alternativa possível, embora indesejável, seria definir os operadores como rotinas normais (não-membro) e torná-los amigos da classe `ListaInt::Iterador`. Isso só seria justificável se fosse importante que ocorressem conversões implícitas em qualquer dos seus operandos, como sucedia no exemplo dos números racionais (ver Capítulo 13). Neste caso essas conversões não são necessárias. Além disso, como se verá mais tarde (ver !!), a tranformação destas classes em classes modelo (genéricas) exige que os operadores de uma classe embutida sejam definidos como membros.

O primeiro operador verifica se a instância implícita (primeiro operando) é igual ao iterador passado como argumento (segundo operando). Dois iteradores são iguais se se referirem ao mesmo item. O segundo verifica se são diferentes:

```
bool operator == (Iterador const& outro_iterador) const;
bool operator != (Iterador const& outro_iterador) const;
```

Só faz sentido invocar estes operadores para iteradores válidos e associados à mesma lista

$$PC \equiv \text{é válido} \wedge \texttt{outro\_ iterador}\ \text{é válido} \wedge \text{associados à mesma lista}.$$

Métodos modificadores

Os únicos métodos modificadores de iteradores são os que permitem avançar e recuar um iterador de um (e um só) item na lista[13]. Em vez de se declararem métodos modificadores com nomes como `avança()` e `recua()`, optou-se por declarar os operadores de incrementação (++) e decrementação (−−), quer prefixos quer sufixos, de modo a que a utilização de iteradores fosse o mais parecida possível quer com a utilização de índices inteiros, quer com a utilização de ponteiros, a estudar no próximo capítulo.

Os operadores de incrementação prefixa e sufixa

```
Iterador& operator ++ ();
Iterador operator ++ (int);
```

exigem ambos que o iterador seja válido e não seja o fim da lista

$$PC \equiv \text{é válido} \wedge \text{não é iterador final da lista}.$$

Os operadores de decrementação prefixa e sufixa

```
Iterador& operator -- ();
Iterador operator -- (int);
```

exigem ambos que o iterador não seja o início da lista

$$PC \equiv \text{é válido} \wedge \text{não é iterador inicial da lista}.$$

Estes dois métodos completam a interface da classe `ListaDeInt::Iterador`. Mais tarde, quando se passar à implementação, discutir-se-ão os membros privados da classe:

```
    private:
      ... // a completar mais tarde, pois é parte da implementação da classe.
};
```

[13]Esta limitação deve-se às mesmas razões pelas quais não se usam índices para localizar itens em listas: implementações adequadas das listas tornariam essas operações onerosas.

Interface completa em C++

!!!O mesmo que para as listas!

```
class ListaDeInt::Iterador {
  public:
    explicit Iterador(ListaDeInt& lista_a_associar);

    Item& item() const;

    bool operator == (Iterador const& outro_iterador) const;
    bool operator != (Iterador const& outro_iterador) const;

    Iterador& operator ++ ();
    Iterador operator ++ (int);

    Iterador& operator -- ();
    Iterador operator -- (int);

  private:
    ... // a completar mais tarde, pois é parte da implementação da classe.
};
```

10.3.3 Usando a interface das novas classes

Uma vez definidos os conceitos de classe e iterador e definidas as respectivas interfaces em C++, pode-se passar directamente ao desenvolvimento de programas que os usem. É claro que não se pode ainda gerar programas executáveis, pois falta a implementação das classes. Como todas as interfaces são contratos, não apenas no caso das rotinas mas também no das classes, o que se possui para já relativamente às classes `ListaDeInt` e `Iterador` é um contrato de promessa. O programador produtor compromete-se a, num determinado prazo, fornecer ao programador consumidor uma implementação para as classes sem qualquer alteração à interface acordada. I.e., compromete-se a, a partir de determinada data, garantir o correcto funcionamento das classes com a interface acordada desde que o programador consumidor garanta, por seu lado, que invoca todos os métodos e rotinas com argumentos que verificam as respectivas pré-condições.

Este contrato permite ao programador consumidor ir desenvolvendo programas usando as classes *ainda antes de estas estarem completas*. É extremamente importante conseguir fazê-lo não apenas porque permite alguns ganhos em produtividade no seio das equipas, mas sobretudo, numa fase de formação, por fazê-lo estimula a capacidade de abstração: se a implementação das classes ainda não existe não é possível que nos distraia ao construirmos código que as usa!

Começa-se por um exemplo simples. Suponha-se uma dada lista `lista`, por exemplo

$$\texttt{lista} = (!\ 1\ 2\ 3\ 11\ 20\ 0\ 354\ 2\ 3\ 45\ 12\ 34\ 30\ 4\ 4\ 23\ 3\ 77\ 4\ \ -1\ \ -20\ 46).$$

Como mostrar os itens desta lista no ecrã? Claramente é necessário percorrê-la, o que só pode ser realizado usando iteradores. O algoritmo é o seguinte:

{ Algoritmo para mostrar todos os itens da lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto `i` não atingiu o fim da lista `lista` **faça-se:**
    mostrar item referenciado por `i`
    avançar `i`

A tradução deste algoritmo para C++ pode ser já feita recorrendo às classes cuja interface se definiu nas secções anteriores:

```
ListaDeInt lista;

... // Aqui preenche-se a lista...

ListaDeInt::Iterador i = lista.primeiro();
while(i != lista.fim()) {
    cout << i.item() << endl;
    ++i;
}
```

ou, usando um ciclo `for`,

```
ListaDeInt lista;

... // Aqui preenche-se a lista...

for(ListaDeInt::Iterador i = lista.primeiro(); i != lista.fim(); ++i)
    cout << i.item() << endl;
```

Um problema mais complexo é o de inserir um item `novo_item` por ordem numa lista `lista`. O algoritmo já foi visto atrás:

{ Algoritmo para inserir ordenadamente o novo item `novo_item` na lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto `i` não atingiu o fim da lista `lista` ∧
        item referenciado por `i` < `novo_item` **faça-se:**
    avançar `i`
inserir `novo_item` na lista `lista` imediatamente antes do item referenciado por `i`

A sua tradução para C++ é mais uma vez imediata:

```
ListaDeInt::Iterador i = lista.primeiro();
while(i != lista.fim() and i.item() < novo_item)
    ++i;

lista.insereAntes(i, novo_item);
```

Assim, é possível escrever código usando classes das quais se conhece apenas a interface. Aliás, esta é a tarefa do programador consumidor, que mesmo que conheça uma implementação, deve-se abstrair dela. É necessário agora implementar estas classes. A responsabilidade de o fazer é do programador produtor. Mas como pode o programador produtor garantir que cumpre a sua parte do contrato? Como pode ter alguma segurança acerca da qualidade dos seus produtos? Para além do cuidado posto no desenvolvimento disciplinado do código, o programador produtor tem de testar o código produzido.

10.3.4 Teste dos módulos

Como pode o programador produtor testar os módulos desenvolvidos, neste caso as classes `ListaDeInt` e `ListaDeInt::Iterador`? Escrevendo código com o objectivo simples de usar os módulos num conjunto de casos considerados interessantes, quer pela sua frequência na prática, quer pela sua natureza extrema. A ideia é levar os módulos desenvolvidos ao limite e verificar se resultam naquilo que se espera.

Tal como o programador consumidor pode começar a escrever código quando ainda só tem disponível a interface de um módulo, também o programador produtor o pode fazer: os testes de um módulo podem ser escritos antes mesmo da sua implementação. Aliás, a palavra "podem" é demasiado fraca: os testes *devem* ser escritos antes da sua implementação. São o ponto de partida para a implementação.

As razões são várias:

- Ao escrever os testes detectam-se erros, inconsistências, faltas e excessos na interface do módulo respectivo.
- Uma vez prontos, os testes podem ser usados, mesmo com implementações incompletas (nesse caso o programador produtor espera e antecipa a ocorrência de erros, naturalmente). Note-se que a execução dos testes com uma implementação incompleta implica que todas as rotinas e métodos do módulo estejam definidos em esqueleto pelo menos[14] (i.e., tem de ser possível construir um executável...).
- Se os testes completos existirem desde o início todas as alterações à implementação são feitas com maior segurança, uma vez que o teste pode ser realizado sem qualquer trabalho e, dessa forma, pode-se confirmar se as alterações tiveram sucesso.

[14]O nome usual para estes esqueletos que compilam embora não façam (ainda) o que é suposto fazerem é *stubs*.

As características desejáveis de um bom teste são pelo menos as seguintes:

- O nível mais baixo ao qual devem existir testes é o de módulo físico. Como é usual que cada módulo físico defina uma única classe (ou pelo menos várias classes totalmente interdependentes), os testes dos módulos físicos normalmente confundem-se com os testes das classes. Este requisito tem apenas a ver com a facilidade com que se podem executar automaticamente todos os testes de um projecto. Por vezes é desejável fugir a esta regra e fazer testes individualizados para algum ou todos os módulos definidos por um módulo físico, mas nesse caso o teste (global) do módulo físico deve-os invocar um por um.
- Os testes devem estar tanto quanto possível embebidos no próprio módulo físico. É usual que em C++ tenham a forma de uma função `main()` colocada dentro do ficheiro de implementação (`.C`) e protegida por uma directiva de compilação condicional à definição de uma macro de nome `TESTE`

  ```
  #ifdef TESTE

  ... // Preâmbulo do teste (#include, etc.)

  int main()
  {
      ... // Conjunto de testes...
  }

  #endif
  ```

  Isto permite facilmente executar os testes sempre que se faz alguma alteração ao módulo.
- Os testes não devem gerar senão mensagens muito simples, informando que se iniciou o teste de um dado módulo físico e de cada um dos seus módulos. No final devem terminar com uma mensagem assinalando o fim dos testes do módulo físico.
- Só em caso de erro devem ser escritas mais mensagens, que deverão se claras e explicativas. As mensagens de erro devem incluir o nome do ficheiro e o número da linha onde o erro foi detectado.
- Caso ocorra algum erro a função `main()` deve devolver 1. Dessa forma podem ser facilmente desenvolvidos programas que geram e executam os testes de todos os módulos físicos de um projecto e geram um relatório global.

!!!!!Valerá a pena manter o apêndice?

Usando as ideias acima desenvolveu-se o programa de teste que se pode encontrar na Secção H.1.3. As listagens completas do módulo físico `lista_int`, que define as classes `ListaDeInt` e `ListaDeInt::Iterador`, encontram-se na Secção H.1.

10.4 Implementação simplista

!!!!!Dizer algures que não se cumpre parte dos contratos! Nomeadamente no que diz respeito à invalidação dos iteradores.

Finalmente chega a altura de se pensar numa implementação para as classes `ListaDeInt` e `ListaDeInt::Iterador`. Mais vale tarde que nunca, dir-se-á. Mas é sempre conveniente adiar a implementação de um módulo até depois de definida a sua interface e de implementada a respectiva função de teste, pelo que esta é a altura certa para atacar o problema.

Nesta secção optar-se-á por uma implementação simplista, e consequentemente ineficiente, para as listas e respectivos iteradores. Discutir-se-ão apenas os aspectos fundamentais dessa implementação, pelo que se recomenda que o leitor complemente esta leitura com o estudo da Secção H.1, onde se encontra listada a implementação completa do módulo.

10.4.1 Implementação de `ListaDeInt`

É natural começar a implementação pelas listas, uma vez que a implementação dos iteradores dependerá da forma como os itens das listas forem organizados.

A questão mais importante é onde guardar os itens. Esta questão pode ser respondida de uma forma simples se se determinar que as listas são limitadas e se impuser que o número máximo de itens é comum a qualquer lista e é exactamente igual a, por exemplo, 100 itens. Com esta restrição, é evidente que os itens podem ser guardados numa matriz clássica[15]. Para isso útil começar por definir uma constante com o número máximo de itens das listas:

```
class ListaDeInt {
  public:
    ...
  private:
    static int const número_máximo_de_itens = 100;
```

Esta constante é partilhada por todas as instâncias da classe lista. Por isso se utilizou a palavra-chave `static`, que aplicada a um atributo torna-o um atributo de classe e não um atributo de instância (ver Secção 7.17.2). O facto de ser um atributo constante de classe e de um tipo aritmético inteiro básico autoriza a sua definição dentro da própria classe. Como o compilador conhece o valor da constante, esta pode ser usada para definir a matriz que guardará os itens:

```
    Item itens[número_máximo_de_itens];
```

Como organizar os itens na matriz? A solução simplista aqui adoptada passa por guardá-los nos primeiros elementos da matriz pela mesma ordem que têm na lista. Dessa forma os

[15] Poderiam ser guardados alternativamente num vector, mas as matrizes têm duas vantagens. A primeira é que demonstram que é possível implementar as classes em causa sem recorrer a outras classes do C++, e portanto de raiz. A segunda é que permitirá uma evolução clara e elegante para as versões mais eficientes da implementação e, finalmente, para o uso de ponteiros e variáveis dinâmicas.

elementos da matriz estarão divididos em duas zonas: os de menor índice estão ocupados com itens da lista, enquanto os de maior índice estão disponíveis para futura utilização. Esta organização torna necessária uma variável que guarde o número de itens da lista em cada instante, i.e., que permita saber a dimensão relativa dessas duas zonas da matriz:

```
    int número_de_itens;
```

A Figura 10.3 ilustra o estado interno de uma lista contendo

`lista` = (! 10 5 3 20 !).

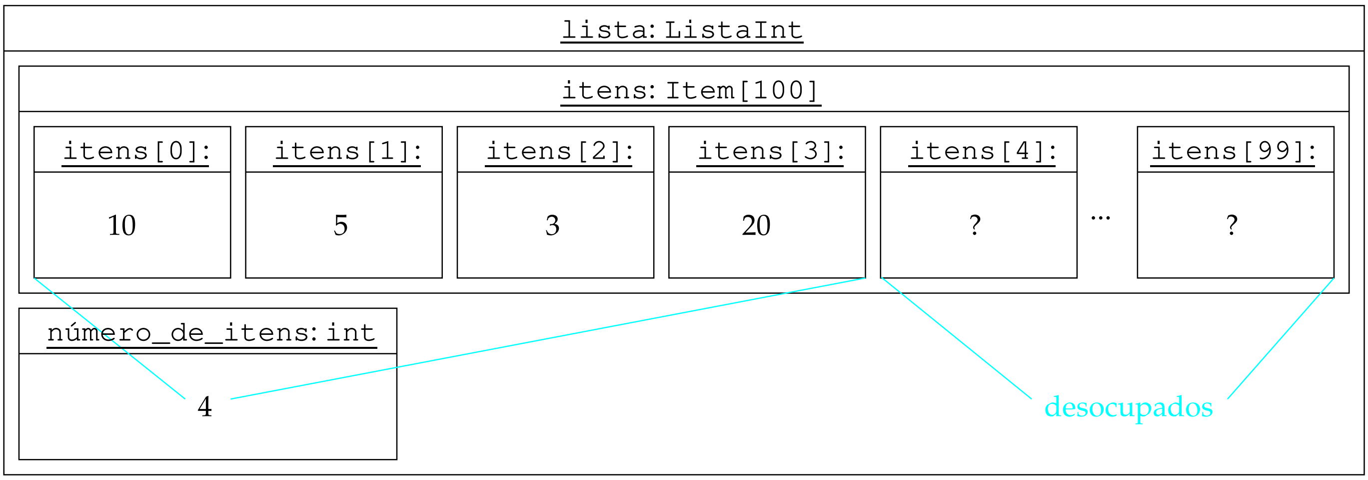

Figura 10.3: Estado interno da lista `lista` = (! 10 5 3 20 !).

Com a implementação escolhida os itens fictícios não têm existência física. Pode-se arbitrar, no entanto, que o item fictício antes da frente é o elemento (inexistente) da matriz com índice -1 e que o item fictício depois da traseira é o elemento (inexistente) com índice `número_de_itens`. Escolheu-se como item fictício final o elemento com índice `número_de_itens` e não `número_máximo_de_i` de modo a que a passagem entre itens se possa fazer sempre por simples incrementação ou decrementação de índices.

A classe `ListaDeInt::Iterador`, que se implentará na próxima secção, necessita de acesso aos atributos privados da classe `ListaDeInt`. De outra forma não se poderia escrever o método `ListaDeInt::Iterador::item()`, como se verá. Assim, é necessário permitir acesso irrestrito dos iteradores às listas, o que se consegue introduzindo uma amizade:

```
    friend class Iterador;
};
```

A utilização de amizades é, em geral, má ideia. Neste caso, porém, existe um casamento perfeito[16] entre as duas classes: o conceito de lista só se completa com o conceito de iterador e o conceito de iterador só se pode concretizar para um tipo específico de contentor, neste caso as listas. É natural que, havendo um casamento perfeito entre duas classes, estas tenham acesso aos membros privados (às partes íntimas, digamos) mutuamente. Sendo o casamento legítimo, esta amizade não é promíscua...

Condição invariante de instância

Qualquer classe que mereça esse nome tem uma condição invariante de instância (CIC). Esta condição indica as relações entre os atributos de instância da classe que se devem verificar em cada instante. No caso presente, a condição invariante de instância é simplesmente

$$CIC \equiv 0 \leq \texttt{número\_ de\_ itens} \leq \texttt{número\_ máximo\_ de\_ itens}.$$

Isto é, para que uma instância da classe ListaDeInt represente de facto uma lista, é necessário que a variável `número_de_itens` contenha um valor não-negativo e inferior ou igual ao máximo estipulado. Não é necessário impor qualquer condição adicional.

A condição invariante de instância tem de se verificar para todas as instâncias em jogo quer no início dos métodos públicos da classe quer no seu final[17]. Para segurança do programador produtor, é conveniente colocar asserções no início e no fim de todos esses métodos que verifiquem explicitamente esta condição. Para simplificar a escrita dessas asserções, pode-se acrescentar à classe um método privado `cumpreInvariante()` para verificar se o invariante é ou não cumprido:

```
class ListaDeInt {
  public:
    ...
  private:
    static int const número_máximo_de_itens = 100;

    Item itens[número_máximo_de_itens];
    int número_de_itens;

    bool cumpreInvariante() const;
};
```

Este método é privado porque o programador consumidor não precisa de verificar nunca se uma lista cumpre o invariante. Limita-se a assumir que sim. De resto, o invariante de uma classe é irrelevante para o programador consumidor, pois reflecte uma implementação particular e nada diz acerca da interface. A definição do método é simplesmente

[16] I.e., por amor.

[17] O caso dos construtores e dos destrutores é especial. A condição invariante de instância deve ser válida apenas no final do construtor e no início do destrutor.

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

10.4.2 Implementação de `ListaDeInt::Iterador`

Dada a implementação da classe `ListaDeInt`, a implementação da classe `ListaDeInt::Iterador` é quase imediata. Em primeiro lugar, cada iterador está associado a uma determinada lista. Assim, é necessário guardar uma referência para a lista associada dentro de cada iterador, pois podem existir variadas lista num programa, cada uma com vários iteradores associados. Em segundo lugar um iterador precisa de saber onde se encontra o item referenciado. Como os itens são guardados na matriz `itens` da classe `ListaDeInt`, basta guardar o índice nessa matriz do item referenciado:

```
class ListaDeInt::Iterador {
  public:
    ...
  private:
    ListaDeInt& lista_associada;
    int índice_do_item_referenciado;
};
```

A Figura 10.5 contém a representação interna da situação retratada na Figura 10.4, em que um iterador `i` referencia o terceiro item de uma lista

$$\texttt{lista} = (10\ 5\ 3\ 20).$$

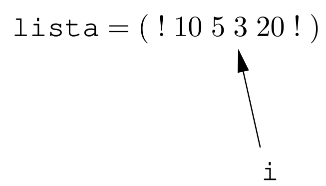

Figura 10.4: Representação simplificada de uma lista $\texttt{lista} = (\,!\ 10\ 5\ 3\ 20\ !\,)$ e de um iterador `i` referenciando o seu terceiro item.

Condição invariante de classe

Também os iteradores têm uma condição invariante de classe. Para esta implementação essa condição é muito simples:

$$\begin{aligned} CIC \equiv -1 \ \leq\ & \texttt{índice\_do\_item\_referenciado} \leq \\ & \texttt{lista\_associada.número\_de\_itens}. \end{aligned}$$

Figura 10.5: Estado interno da lista `lista` = (! 10 5 3 20 !) e de um iterador `i` referenciando o seu terceiro item.

e indica simplesmente que o índice do item referenciado tem de estar entre -1 (se o iterador for o início da lista) e `lista_associada.número_de_itens` (se o iterador for o início da lista).

O problema é que os iteradores, tal como pensados até aqui, têm uma característica infeliz: podem estar em estados inválidos. Por exemplo, se uma lista for esvaziada, todos os iteradores a ela associada ficam inválidos. Essa condição de iterador inválido deveria ser prevista pelo próprio código de tal forma que a condição invariante de instância fosse verdadeira mesmo para iteradores inválidos:

$$\begin{aligned} CIC \equiv \neg \text{ é válido} \vee -1 \leq & \texttt{ índice\_ do\_ item\_ referenciado} \leq \\ & \texttt{lista\_ associada.número\_ de\_ itens}. \end{aligned}$$

Na prática poder-se-ia guardar informação acerca do estado de validade de um iterador num novo atributo, booleano. Mas este pequeno acrescento exigiria muitas outras alterações, nomeadamente na implementação das listas, o que levaria um aumento considerável da complexidade deste par de classes. Em particular seria necessário prever o estado de iterador inválido em todas as operações que os envolvessem e, na implementação das listas, garantir que os iteradores invalidados fossem assinalados como tal. Fazê-lo é um exercício interessante e útil, mas fora do âmbito deste capítulo e do próximo[18]. Assim, com as implementações que serão feitas neste texto, deverá ser o programador consumidor a garantir que não faz uso de um iterador depois de este ter sido invalidado por alguma operação realizada sobre a lista associada: o código, através do método privado `cumpreInvariante()` que também será definido nesta classe, não o verificará:

```
class ListaDeInt::Iterador {
  public:
    ...
  private:
    ListaDeInt& lista_associada;
    int índice_do_item_referenciado;

    bool cumpreInvariante() const;
};

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

10.4.3 Implementação dos métodos públicos de `ListaDeInt`

Neste secção definir-se-ão alguns dos métodos da classe `ListaDeInt`. Os restantes ficam como exercício para o leitor, que poderá depois conferir com a implementação total do módulo `lista_int` que se encontra na Secção H.1.

[18] O Apêndice I conterá no futuro uma versão mais segura e comentada das classes `ListaInt` e `Iterador` que verifica e mantém informação acerca da validade dos iteradores. !!!referir STLport

O primeiro método a definir é o construtor da lista, que se pode limitar a inicializar o número de itens com o valor zero, para que a lista fique vazia:

```
inline ListaDeInt::ListaDeInt()
    : número_de_itens(0) {
```

termina-se verificando se a nova lista cumpre o seu invariante:

```
    assert(cumpreInvariante());
}
```

O método `ListaDeInt::põeAtrás()` é também muito simples. Basta colocar o novo item após todos os itens existentes (ver Figura 10.3) e incrementar o número de itens:

```
inline void ListaDeInt::põeAtrás(Item const& novo_item) {
```

Começa-se por verificar se a lista cumpre a sua condição invariante de instância,

```
    assert(cumpreInvariante());
```

depois verifica-se a pré-condição do método,

```
    assert(not estáCheia());
```

faz-se a inserção do novo item,

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

e termina-se verificando se no final do método a lista continua a cumprir a sua condição invariante de instância.

```
    assert(cumpreInvariante());
}
```

O método `ListaDeInt::põeNaFrente()` é um pouco mais complicado. O primeiro item está na posição 0 da matriz, pelo que é necessário deslocar todos os itens existentes de modo a deixar espaço para o novo item. Esta é a fonte principal de ineficiência desta implementação e a justificação para a necessidade de se mudar a implementação das listas, como se fará mais tarde:

```
void ListaDeInt::põeNaFrente(Item const& novo_item)
{
```

Começa-se por verificar a condição invariante de instância e a pré-condição do método,

```
        assert(cumpreInvariante());
        assert(not estáCheia());
```

Rearranjam-se os elementos da matriz de modo a deixar um espaço vago na posição 0,

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

depois insere-se o novo item na posição livre,

```
        itens[0] = novo_item;
```

incrementa-se o contador de itens,

```
        ++número_de_itens;
```

e termina-se verificando se no final do método a lista continua a cumprir a sua condição invariante de instância.

```
        assert(cumpreInvariante());
    }
```

O método `ListaDeInt::insereAntes()` é muito semelhante, embora o elemento da matriz a vagar seja o que contém o item referenciado pelo iterador, pelo que já não é necessário, em geral, rearranjar todos os elementos da matriz:

```
    void ListaDeInt::insereAntes(Iterador& iterador,
                                 Item const& novo_item)
    {
        assert(cumpreInvariante());
        assert(not estáCheia());
        assert(iterador é válido);
        assert(iterador != lista_associada.início());

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

Tem de se incrementar o índice do item referenciado pelo iterador, para que ele passe a referenciar o item imediatamente depois do que se pretende remover.

```
        itens[iterador.índice_do_item_referenciado++] = novo_item;

        assert(iterador.cumpreInvariante());

        ++número_de_itens;

        assert(cumpreInvariante());
    }
```

Os métodos `ListaDeInt::remove()` e `ListaDeInt::tiraDaFrente()` têm de fazer a operação inversa. Define-se aqui o mais complicado dos dois:

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

        --número_de_itens;
```

A decrementação do número de itens é fundamental para o bom funcionamento do ciclo!

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

O iterador fica automaticamente no local apropriado.

```
        assert(cumpreInvariante());
    }
```

Para terminar definem-se dois dos métodos construtores de iteradores:

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

Constrói-se um novo iterador para esta lista, que referencia inicialmente o item na frente da lista (ver construtor da classe `ListaDeInt::Iterador`), e devolve-se imediatamente o iterador criado.

```
        return Iterador(*this);
    }

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

        Iterador iterador(*this);
```

Aqui, depois de construído um novo iterador, altera-se o seu índice, de modo a que referencie o item desejado (o da traseira).

```
        iterador.índice_do_item_referenciado = número_de_itens - 1;
```

Não é boa ideia que seja a classe `ListaDeInt` a invocar directamente o método de verificação do invariante da classe `ListaDeInt::Iterador`: Neste caso está-se mesmo a introduzir um excesso de intimidade... Mais tarde se verá uma melhor solução para este problema.

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

        return iterador;
    }
```

10.4.4 Implementação dos métodos públicos de `ListaDeInt::Iterador`

Neste secção definir-se-ão alguns dos métodos da classe `ListaDeInt::Iterador`. Os restantes ficam como exercício para o leitor, que poderá depois conferir com a implementação total do módulo `lista_int` que se encontra na Secção H.1.

O primeiro método a definir é o construtor da classe. Este método inicializa um novo ponteiro e modo a que referencie o primeiro item de uma lista dada:

```
    inline ListaDeInt::Iterador::Iterador(ListaDeInt& lista_a_associar)
```

O atributo `lista_associada` é uma referência, pelo que neste caso não é apenas conveniente ou recomendável usar uma lista de inicializadores: é obrigatório.

```
        : lista_associada(lista_a_associar),
          índice_do_item_referenciado(0) {
```

Mais uma vez se termina verificando a condição invariante de instância.

```
        assert(cumpreInvariante());
    }
```

O operador de decrementação serve para recuar o item referenciado na lista:

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

Basta decrementar o índice do item referenciado.

```
        --índice_do_item_referenciado;

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

O operador de igualdade entre iteradores é muito simples. Dois uteradores válidos e associados à mesma lista são iguais se referenciarem o mesmo item, ou seja, se tiverem o mesmo valor no atributo `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...);
```

É importante garantir que os iteradores estão associados à mesma lista. O problema resolver-se-á no próximo capítulo, quando se introduzirem os conceitos e ponteiro e endereço.

```
        return índice_do_item_referenciado ==
               outro_iterador.índice_do_item_referenciado;
    }
```

Finalmente, define-se o método inspector que devolve uma referência para o item referenciado pelo iterador:

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

Esta linha de código, em que a classe `ListaDeInt::Iterador` acede a um atributo da classe `ListaDeInt` é uma das razões pelas quais se declarou a primeira classe amiga da segunda.

```
        return lista_associada.itens[índice_do_item_referenciado];
    }
```