Capítulo 5

Matrizes, vectores e outros agregados

É muitas vezes conveniente para a resolução de problemas ter uma forma de guardar agregados de valores de um determinado tipo. Neste capítulo apresentam-se duas alternativas para a representação de agregados em C++: as matrizes e os vectores. As primeiras fazem parte da linguagem propriamente dita e são bastante primitivas e pouco flexíveis. No entanto, há muitos problemas para os quais são a solução mais indicada, além de que existe muito código C++ escrito que as utiliza, pelo que a sua aprendizagem é importante. Os segundos não fazem parte da linguagem. São fornecidos pela biblioteca padrão do C++, que fornece um conjunto vasto de ferramentas que, em conjunto com a linguagem propriamente dita, resultam num potente ambiente de programação. Os vectores são consideravelmente mais flexíveis e fáceis de usar que as matrizes, pelo que estas serão apresentadas primeiro. No entanto, as secções sobre matrizes de vectores têm a mesma estrutura, pelo que o leitor pode optar por ler as secções pela ordem inversa à da apresentação.

Considere-se o problema de calcular a média de três valores de vírgula flutuante dados e mostrar cada um desses valores dividido pela média calculada. Um possível programa para resolver este problema é:

```
#include <iostream>

using namespace std;

int main()
{
    // Leitura:
    cout << "Introduza três valores: ";
    double a, b, c;
    cin >> a >> b >> c;

    // Cálculo da média:
    double const média = (a + b + c) / 3.0;

    // Divisão pela média:
```

```
    a /= média;
    b /= média;
    c /= média;

    // Escrita do resultado:
    cout << a << ' ' << b << ' ' << c << endl;
}
```

O programa é simples e resolve bem o problema em causa. Mas será facilmente generalizável? E se se pretendesse dividir, já não três, mas 100 ou mesmo 1000 valores pela sua média? A abordagem seguida no programa acima seria no mínimo pouco elegante, já que seriam necessárias tantas variáveis quantos os valores a ler do teclado, e estas variáveis teriam de ser todas definidas explicitamente. Convém usar um *agregado* de variáveis.

5.1 Matrizes clássicas do C++

A generalização de este tipo de problemas faz-se recorrendo a uma das possíveis formas de agregado que são as matrizes clássicas do C++[1]. Uma matriz é um agregado de elementos do mesmo tipo que podem ser indexados (identificados) por números inteiros e que podem ser interpretados e usados como outra variável qualquer.

A conversão do programa acima para ler 1000 em vez de três valores pode ser feita facilmente recorrendo a este novo conceito:

```
#include <iostream>

using namespace std;

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";

    // Definição da matriz com número_de_valores elementos do tipo double:
    double valores[número_de_valores];

    for(int i = 0; i != número_de_valores; ++i)
        cin >> valores[i]; // lê o i-ésimo valor do teclado.

    // Cálculo da média:
```

[1]Traduziu-se o inglês *array* por matriz, à falta de melhor alternativa. Isso pode criar algumas confusões se se usar uma biblioteca que defina o conceito matemático de matriz. Por isso a estas matrizes básicas se chama “matrizes clássicas do C++”.

```
    double soma = 0.0;
    for(int i = 0; i != número_de_valores; ++i)
        soma += valores[i]; // acrescenta o i-ésimo valor à soma.

    double const média = soma / número_de_valores;

    // Divisão pela média:
    for(int i = 0; i != número_de_valores; ++i)
        valores[i] /= média; // divide o i-ésimo valor pela média.

    // Escrita do resultado:
    for(int i = 0; i != número_de_valores; ++i)
        cout << valores[i] << ' '; // escreve o i-ésimo valor.
    cout << endl;
}
```

Utilizou-se uma constante para representar o número de valores a processar. Dessa forma, alterar o programa para processar não 1000 mas 10000 valores é trivial: basta alterar o valor da constante. A alternativa à utilização da constante seria usar o valor literal 1000 explicitamente onde quer que fosse necessário. Isso implicaria que, ao adaptar o programa para fazer a leitura de 1000 para 10000 valores, o programador recorreria provavelmente a uma substituição de 1000 por 10000 ao longo de todo o texto do programa, usando as funcionalidade do editor de texto. Essa substituição poderia ter consequências desastrosas se o valor literal 1000 fosse utilizado em algum local do programa num contexto diferente.

Recorda-se que o nome de uma constante atribui um significado ao valor por ela representado. Por exemplo, definir num programa de gestão de uma disciplina as constantes:

```
int const número_máximo_de_alunos_por_turma = 50;
int const peso_do_trabalho_na_nota_final = 50;
```

permite escrever o código sem qualquer utilização explícita do valor 50. A utilização explícita do valor 50 obrigaria a inferir o seu significado exacto (número máximo de alunos ou peso do trabalho na nota final) a partir do contexto, tarefa que nem sempre é fácil e que se torna mais difícil à medida que os programas se vão tornando mais extensos.

5.1.1 Definição de matrizes

Para utilizar uma matriz de variáveis é necessário defini-la[2]. A sintaxe da definição de matrizes é simples:

```
tipo nome[número_de_elementos];
```

[2]Na realidade, para se usar uma variável, é necessário que ela esteja declarada, podendo por vezes a sua definição estar noutro local, um pouco como no caso das funções e procedimentos. No Capítulo 9 se verá como e quando se podem declarar variáveis ou constantes sem as definir.

em que `tipo` é o tipo de cada elemento da matriz, `nome` é o nome da matriz, e *`número_de_elementos`* é o número de elementos ou dimensão da nova matriz. Por exemplo:

```
int mi[10];    // matriz com 10 int.
char mc[80];   // matriz com 80 char.
float mf[20]; // matriz com 20 float.
double md[3]; // matriz com 3 double.
```

O número de elementos de uma matriz tem de ser um valor conhecido durante a compilação do programa (i.e., um valor literal ou uma constante). Não é possível criar uma matriz usando um número de elementos variável[3]. Mas é possível, e em geral aconselhável, usar constantes em vez de valores literais para indicar a sua dimensão:

```
int n = 50;
int const m = 100;
int matriz_de_inteiros[m];    // ok, m é uma constante.
int matriz_de_inteiros[300]; // ok, 300 é um valor literal.
int matriz_de_inteiros[n];    // errado, n não é uma constante.
```

Nem todas as constantes têm um valor conhecido durante a compilação: a palavra-chave `const` limita-se a indicar ao compilador que o objecto a que diz respeito não poderá ser alterado. Por exemplo:

```
// Valor conhecido durante a compilação:
int const número_máximo_de_alunos = 10000;
int alunos[número_máximo_de_alunos]; // ok.

int número_de_alunos_lido;
cin >> número_de_alunos_lido;

// Valor desconhecido durante a compilação:
int const número_de_alunos = número_de_alunos_lido;
int alunos[número_de_alunos]; // erro!
```

5.1.2 Indexação de matrizes

Seja a definição

```
int m[10];
```

Para atribuir o valor 5 ao sétimo elemento da matriz pode-se escrever:

[3]Na realidade podem-se definir matrizes com um dimensão variável, desde que seja uma matriz *dinâmica*. Alternativamente pode-se usar o tipo genérico `vector` da biblioteca padrão do C++. Ambos os assuntos serão vistos mais tarde.

```
m[6] = 5;
```

Ao valor 6, colocado entre `[]`, chama-se *índice*. Ao escrever numa expressão o nome de uma matriz seguido de `[]` com um determinado índice está-se a efectuar uma *operação de indexação*, i.e., a aceder a um dado elemento da matriz indicando o seu índice.

Os índices das matrizes são sempre números inteiros e começam sempre em zero. Assim, o primeiro elemento da matriz `m` é `m[0]`, o segundo `m[1]`, e assim sucessivamente. Embora esta convenção pareça algo estranha a início, ao fim de algum tempo torna-se muito natural. Recorde-se os problemas de contagem de anos que decorreram de chamar ano 1 ao primeiro ano de vida de Cristo e a consequente polémica acerca de quando ocorre de facto a mudança de milénio[4]...

Como é óbvio, os índices usados para aceder às matrizes não necessitam de ser constantes: podem-se usar variáveis, como aliás se pode verificar no exemplo da leitura de 1000 valores apresentado atrás. Assim, o exemplo anterior pode-se escrever:

```
int a = 6;
m[a] = 5; // atribui o inteiro 5 ao 7º elemento da matriz.
```

Usando de novo a analogia das folhas de papel, uma matriz é como um bloco de notas em que as folhas são numeradas a partir de 0 (zero). Ao se escrever `m[6] = 5`, está-se a dizer algo como "substitua-se o valor que está escrito na folha 6 (a sétima) do bloco `m` pelo valor 5". Na Figura 5.1 mostra-se um diagrama com a representação gráfica da matriz `m` depois desta atribuição (por '?' representa-se um valor arbitrário, também conhecido por "lixo").

m: int[10]									
m[0]:	m[1]:	m[2]:	m[3]:	m[4]:	m[5]:	m[6]:	m[7]:	m[8]:	m[9]:
?	?	?	?	?	?	5	?	?	?

Figura 5.1: Matriz `m` definida por `int m[10];` depois das instruções `int a = 6;` e `m[a] = 5;`.

Porventura uma das maiores deficiências da linguagem C++ está no tratamento das matrizes, particularmente na indexação. Por exemplo, o código seguinte não resulta em qualquer erro de compilação nem tão pouco na terminação do programa com uma mensagem de erro, isto apesar de tentar atribuir valores a posições inexistentes da matriz (as posições com índices -1 e 4):

[4]Note-se que o primeiro ano antes do nascimento de Cristo é o ano -1: não há ano zero! O problema é mais frequente do que parece. Em Portugal, por exemplo, a numeração dos andares começa em R/C, ou zero, enquanto nos EUA começa em um (e que número terá o andar imediatamente abaixo?). Ainda no que diz respeito a datas e horas, o primeiro dia de cada mês é 1, mas a primeira hora do dia (e o primeiro minuto de cada hora) é 0, apesar de nos relógios analógicos a numeração começar em 1! E, já agora, porque se usam em português as expressões absurdas "de hoje a oito dias" e "de hoje a quinze dias" em vez de "daqui a sete dias" ou "daqui a 14 dias"? Será que "amanhã" é sinónimo de "de hoje a dois dias" e "hoje" o mesmo que "de hoje a um dia"?

```
int a = 0;
int m[4];
int b = 0;

m[-1] = 1; // erro! só se pode indexar de 0 a 3!
m[4] = 2;  // erro! só se pode indexar de 0 a 3!
cout << a << ' ' << b << endl;
```

O que aparece provavelmente escrito no ecrã é, dependendo do ambiente em que o programa foi compilado e executado,

```
2 1
```

ou

```
1 2
```

dependendo da arrumação que é dada às variáveis `a`, `m` e `b` na memória. O que acontece é que as variáveis são arrumadas na memória (neste caso na pilha ou *stack*) do computador por ordem de definição, pelo que as "folhas de papel" correspondentes a `a` e `b` seguem ou precedem (conforme a direcção de crescimento da pilha) as folhas do "bloco de notas" correspondente a `m`. Assim, a atribuição `m[-1] = 1;` coloca o valor 1 na folha que precede a folha 0 de `m` na memória, que é normalmente a folha de `b` (a pilha normalmente cresce "para baixo", como se verá na disciplina de Arquitectura de Computadores).

Esta é uma fonte muito frequente de erros no C++, pelo que se recomenda extremo cuidado na utilização de matrizes. Uma vez que muitos ciclos usam matrizes, este é mais um argumento a favor do desenvolvimento disciplinado de ciclos (Secção 4.7) e da utilização de asserções (Secção 3.2.19).

5.1.3 Inicialização de matrizes

Tal como para as variáveis simples, também as matrizes só são inicializadas implicitamente quando são estáticas[5] . Matrizes automáticas, i.e., locais a alguma rotina (e sem o qualificador `static`) não são inicializadas: os seus elementos contêm inicialmente valores arbitrários ("lixo"). Mas é possível inicializar explicitamente as matrizes. Para isso, colocam-se os valores com que se pretende inicializar os elementos da matriz entre `{}`, por ordem e separados por vírgulas. Por exemplo:

```
int m[4] = {10, 20, 0, 0};
```

[5]Na realidade as variáveis de tipos definidos pelo programador ou os elementos de matrizes com elementos de um tipo definido pelo programador (com excepção dos tipos enumerados) são sempre inicializadas, ainda que implicitamente (nesse caso são inicializadas com o construtor por omissão, ver Secção 7.17.5). Só variáveis automáticas ou elementos de matrizes automáticas de tipos básicos do C++ ou de tipos enumerados não são inicializadas implicitamente, contendo por isso lixo se não forem inicializadas explicitamente.

Podem-se especificar menos valores de inicialização do que o número de elementos da matriz. Nesse caso, os elementos por inicializar explicitamente são inicializados implicitamente com 0 (zero), ou melhor, com o valor por omissão correspondente ao seu tipo (o valor por omissão dos tipos aritméticos é zero, o dos booleanos é falso, o dos enumerados, ver Capítulo 6, é zero, mesmo que este valor não seja tomado por nenhum dos seus valores literais enumerados, e o das classes, ver Secção 7.17.5, é o valor construído pelo construtor por omissão). Ou seja,

```
int m[4] = {10, 20};
```

tem o mesmo efeito que a primeira inicialização. Pode-se aproveitar este comportamento algo obscuro para forçar a inicialização de uma matriz inteira com zeros:

```
int grande[100] = {}; // inicializa toda a matriz com 0 (zero).
```

Mas, como este comportamento é tudo menos óbvio, recomenda-se que se comentem bem tais utilizações, tal como foi feito aqui.

Quando a inicialização é explícita, pode-se omitir a dimensão da matriz, sendo esta inferida a partir do número de inicializadores. Por exemplo,

```
int m[] = {10, 20, 0, 0};
```

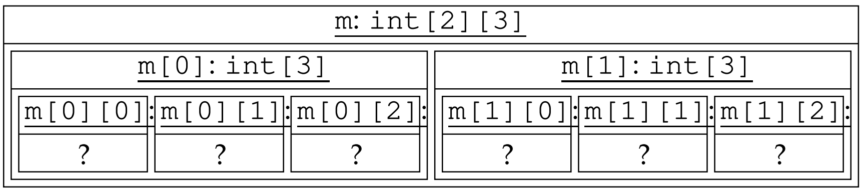
tem o mesmo efeito que as definições anteriores.

5.1.4 Matrizes multidimensionais

Em C++ não existe o conceito de matrizes multidimensionais. Mas a sintaxe de definição de matrizes permite a definição de matrizes cujos elementos são outras matrizes, o que acaba por ter quase o mesmo efeito prático. Assim, a definição:

```
int m[2][3];
```

é interpretada como significando `int (m[2])[3]`, o que significa "`m` é uma matriz com dois elementos, cada um dos quais é uma matriz com três elementos inteiros". Ou seja, graficamente:



Embora sejam na realidade matrizes de matrizes, é usual interpretarem-se como matrizes multidimensionais (de resto, será esse o termo usado daqui em diante). Por exemplo, a matriz m acima é interpretada como:

A indexação destas matrizes faz-se usando tantos índices quantas as "matrizes dentro de matrizes" (incluindo a exterior), ou seja, tantos índices quantas as dimensões da matriz. Para `m` conforme definida acima:

```
m[1][2] = 4;          // atribui 4 ao elemento na linha 1, coluna 2 da matriz.
int i = 0, j = 0;
m[i][j] = m[1][2];  // atribui 4 à posição (0,0) da matriz.
```

A inicialização destas matrizes pode ser feita como indicado, tomando em conta, no entanto, que cada elemento da matriz é por sua vez uma matriz e por isso necessita de ser inicializado da mesma forma. Por exemplo, a inicialização:

```
int m[2][3] = { {1, 2, 3}, {4} };
```

leva a que o troço de programa[6]

```
for(int i = 0; i != 2; ++i) {
    for(int j = 0; j != 3; ++j)
        cout << setw(2) << m[i][j];
   cout << endl; }
```

escreva no ecrã

```
1 2 3
4 0 0
```

5.1.5 Matrizes constantes

Não existe em C++ a noção de matriz constante. Um efeito equivalente pode no entanto ser obtido indicando que os elementos da matriz são constantes. Por exemplo:

```
int const dias_no_mês_em_ano_normal[] = {
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

int const dias_no_mês_em_ano_bissexto[] = {
    31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

5.1.6 Matrizes como parâmetros de rotinas

É possível usar matrizes como parâmetros de rotinas. O programa original pode ser modularizado do seguinte modo:

[6] O manipulador `setw()` serve para indicar quantos caracteres devem ser usados para escrever o próximo valor inserido no canal. Por omissão são acrescentados espaços à esquerda para perfazer o número de caracteres indicado. Para usar este manipulador é necessário fazer `#include <iomanip>`.

```
#include <iostream>

using namespace std;

int const número_de_valores = 1000;

/** Preenche a matriz com valores lidos do teclado.
    @pre PC ≡ o canal de entrada (cin) contém número_de_valores
              números decimais.
    @post CO ≡ a matriz m contém os valores decimais que estavam em cin. */
void lê(double m[número_de_valores])
{
    for(int i = 0; i != número_de_valores; ++i)
        cin >> m[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos valores na matriz.
    @pre PC ≡ 𝒱.
    @post CO ≡ média = (S j : 0 ≤ j < número_de_valores : m[j]) / número_de_valores. */
double média(double const m[número_de_valores])
{
    double soma = 0.0;
    for(int i = 0; i != número_de_valores; ++i)
        soma += m[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / número_de_valores;
}

/** Divide todos os elementos da matriz por divisor.
    @pre PC ≡ divisor ≠ 0 ∧ m = m.
    @post CO ≡ (Q j : 0 ≤ j < número_de_valores : m[j] = m[j] / divisor). */
void normaliza(double m[número_de_valores], double const divisor)
{
    assert(divisor != 0); // ver nota7 abaixo.

    for(int i = 0; i != número_de_valores; ++i)
        m[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores no ecrã.
    @pre PC ≡ 𝒱.
    @post CO ≡ o canal cout contém representações dos valores em m,
```

[7] Na pré-condição há um termo, $\mathtt{m} = m$, em que ocorre uma variável matemática: m. Como esta variável não faz parte do programa C++, não é possível usar esse termo na instrução de asserção. De resto, essa variável é usada simplesmente para na condição objectivo significar o valor original da variável de programa m. Como se disse mais atrás, o mecanismo de instruções de asserção do C++ é algo primitivo, não havendo nenhuma forma de incluir referências ao valor original de uma variável.

```
               por ordem. */
void escreve(double const m[número_de_valores])
{
    for(int i = 0; i != número_de_valores; ++i)
        cout << m[i] << ' '; // escreve o i-ésimo elemento.
}

int main()
{
    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    double valores[número_de_valores];
    lê(valores);

    // Divisão pela média:
    normaliza(valores, média(valores));

    // Escrita do resultado:
    escreve(valores);
    cout << endl;
}
```

Passagens por referência

Para o leitor mais atento o programa acima tem, aparentemente, um erro. É que, nos procedimentos onde se alteram os valores da matriz (nomeadamente `lê()` e `normaliza()`), a matriz parece ser passada por valor e não por referência, visto que não existe na declaração dos parâmetros qualquer `&` (ver Secção 3.2.11). Acontece que as matrizes são sempre passadas por referência[8] e o `&` é, portanto, redundante[9]. Assim, os dois procedimentos referidos alteram de facto a matriz que lhes é passada como argumento.

Esta é uma característica desagradável das matrizes, pois introduz uma excepção à semântica das chamadas de rotinas. Esta característica mantém-se na linguagem apenas por razões de compatibilidade com código escrito na linguagem C.

Dimensão das matrizes parâmetro

Há uma outra característica das matrizes que pode causar alguma perplexidade. Se um parâmetro de uma rotina for uma matriz, então a sua dimensão é simplesmente ignorada pelo compilador. Assim, o compilador não verifica a dimensão das matrizes passadas como argumento, limitando-se a verificar o tipo dos seus elementos. Esta característica está relacionada com o

[8] A verdadeira explicação não é esta. A verdadeira explicação será apresentada quando se introduzir a noção de ponteiro no Capítulo 11.

[9] Poder-se-ia ter explicitado a referência escrevendo `void lê(double (&m)[número_de_valores])` (os `()` são fundamentais), muito embora isso não seja recomendado, pois sugere que na ausência do `&` a passagem se faz por valor, o que não é verdade.

facto de não se verificar a validade dos índices em operações de indexação[10], e com o facto de estarem proibidas a maior parte das operações sobre matrizes tratadas como um todo (ver Secção 5.1.7). Assim, `lêMatriz(double m[número_de_valores])` e `lêMatriz(double m[])` têm exactamente o mesmo significado[11].

Este facto pode ser usado a favor do programador (se ele tiver cuidado), pois permite-lhe escrever rotinas que operam sobre matrizes de dimensão arbitrária, desde que a dimensão das matrizes seja também passada como argumento[12]. Assim, na versão do programa que se segue todas as rotinas são razoavelmente genéricas, pois podem trabalhar com matrizes de dimensão arbitrária:

```
#include <iostream>

using namespace std;

/** Preenche os primeiros n elementos da matriz com n valores lidos do teclado.
    @pre  PC ≡ 0 ≤ n ∧ n ≤ dim(m)∧o canal de entrada (cin)
              contém n números decimais13.
    @post CO ≡a matriz m contém os n valores decimais que estavam em cin. */
void lê(double m[], int const n)
{
    assert(0 <= n); // ver nota14 abaixo.

    for(int i = 0; i != n; ++i)
```

[10]Na realidade as matrizes são sempre passadas na forma de um ponteiro para o primeiro elemento, como se verá no !!.

[11]Esta equivalência deve-se ao compilador ignorar a dimensão de uma matriz indicada como parâmetro de uma função ou procedimento. Isto não acontece se a passagem por referência for explicitada conforme indicado na nota Nota 9 na página 244: nesse caso a dimensão não é ignorada e o compilador verifica se o argumento respectivo tem tipo e dimensão compatíveis.

[12]Na realidade o valor passado como argumento não precisa de ser a dimensão real da matriz: basta que seja menor ou igual ao dimensão real da matriz. Isto permite processar apenas parte de uma matriz.

[13]Em expressões matemáticas, $\dim(m)$ significa a dimensão de uma matriz m. A mesma notação também se pode usar no caso dos vectores, que se estudarão em secções posteriores.

[14]Nesta instrução de asserção não é possível fazer melhor do que isto. Acontece que é impossível verificar se existem `n` valores decimais disponíveis para leitura no canal de entrada antes de os tentar extrair e, por outro lado, é característica desagradável da linguagem C++ não permitir saber a dimensão de matrizes usadas como parâmetro de uma função ou procedimento.

O primeiro problema poderia ser resolvido de duas formas. A primeira, mais correcta, passava por exigir ao utilizador a introdução dos números decimais pretendidos, insistindo com ele até a inserção ter sucesso. Neste caso a pré-condição seria simplificada. Alternativamente poder-se-ia usar uma asserção para saber se cada extracção teve sucesso. Como um canal, interpretado como um booleano, tem valor verdadeiro se e só se não houve qualquer erro de extracção, o ciclo poderia ser reescrito como:

```
for(int i = 0; i != n; ++i) {
    cin >> m[i]; // lê o i-ésimo elemento do teclado.
    assert(cin);
}
```

Um canal de entrada, depois de um erro de leitura, fica em estado de erro, falhando todas as extracções que se tentarem realizar. Um canal em estado de erro tem sempre o valor falso quando interpretado como um booleano.

```
        cin >> m[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos n primeiros elementos da matriz.
    @pre PC ≡ 1 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ média = (S j : 0 ≤ j < n : m[j]) / n. */
double média(double const m[], int const n)
{
    assert(1 <= n);

    double soma = 0.0;
    for(int i = 0; i != n; ++i)
        soma += m[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / n;
}

/** Divide os primeiros n elementos da matriz por divisor.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m) ∧ divisor ≠ 0 ∧ m = m.
    @post CO ≡ (Q j : 0 ≤ j < n : m[j] = m[j] / divisor). */
void normaliza(double m[], int const n, double const divisor)
{
    assert(0 <= n and divisor != 0);

    for(int i = 0; i != n; ++i)
        m[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores no ecrã.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ o canal cout contém representações dos primeiros n
                elementos de m, por ordem. */
void escreve(double const m[], int const n)
{
    assert(0 <= n);

    for(int i = 0; i != n; ++i)
        cout << m[i] << ' '; // escreve o i-ésimo elemento.
}

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    double valores[número_de_valores];
```

```
    lê(valores, número_de_valores);

    // Divisão pela média:
    normaliza(valores, número_de_valores,
              média(valores, número_de_valores));

    // Escrita do resultado:
    escreve(valores, número_de_valores);
    cout << endl;
}
```

O caso das matrizes multidimensionais é mais complicado: apenas é ignorada a primeira dimensão das matrizes multidimensionais definidas como parâmetros de rotinas. Assim, é impossível escrever

```
double determinante(double const m[][],
                    int const linhas, int const colunas)
{
    ...
}
```

com a esperança de definir uma função para calcular o determinante de uma matriz bidimensional de dimensões arbitrárias... É obrigatório indicar as dimensões de todas as matrizes excepto a mais "exterior":

```
double determinante(double const m[][10], int const linhas)
{
    ...
}
```

Ou seja, a flexibilidade neste caso resume-se a que a função pode trabalhar com um número arbitrário de linhas[15]...

5.1.7 Restrições na utilização de matrizes

Para além das particularidades já referidas relativamente à utilização de matrizes em C++, existem duas restrições que é importante conhecer:

Devolução Não é possível devolver matrizes (directamente) em funções. Esta restrição desagradável obriga à utilização de procedimentos para processamento de matrizes.

Operações Não são permitidas as atribuições ou comparações entre matrizes. Estas operações têm de ser realizadas através da aplicação sucessiva da operação em causa a cada um dos elementos da matriz. Pouco prático, em suma.

[15] Na realidade nem isso, porque o determinante de uma matriz só está definido para matrizes quadradas...

5.2 Vectores

Viu-se que as matrizes têm um conjunto de particularidades que as tornam pouco simpáticas de utilizar na prática. No entanto, a linguagem C++ traz associada a chamada biblioteca padrão (por estar disponível em qualquer ambiente de desenvolvimento que se preze) que disponibiliza um conjunto muito vasto de ferramentas (rotinas, variáveis e constantes, tipos, etc.), entre os quais um tipo genérico chamado `vector`. Este tipo genérico é uma excelente alternativa às matrizes e é apresentado brevemente nesta secção.

Pode-se resolver o problema de generalizar o programa apresentado no início deste capítulo recorrendo a outra das possíveis formas de agregado: os vectores. Ao contrário das matrizes, os vectores não fazem parte da linguagem C++. Os vectores são um tipo genérico, ou melhor uma classe C++ genérica, definido na biblioteca padrão do C++ e a que se acede colocando

```
#include <vector>
```

no início do programa. As noções de classe C++ e classe C++ genérica serão abordadas no Capítulo 7 e no Capítulo 13, respectivamente.

Um vector é um contentor de itens do mesmo tipo que podem ser indexados (identificados) por números inteiros e que podem ser interpretados e usados como outra variável qualquer. Ao contrário das matrizes, os vectores podem ser redimensionados sempre que necessário.

A conversão do programa inicial para ler 1000 em vez de três valores pode ser feita facilmente recorrendo ao tipo genérico `vector`:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";

    // Definição de um vector com número_de_valores itens do tipo double:
    vector<double> valores(número_de_valores);

    for(int i = 0; i != número_de_valores; ++i)
        cin >> valores[i]; // lê o i-ésimo valor do teclado.

    // Cálculo da média:
    double soma = 0.0;
    for(int i = 0; i != número_de_valores; ++i)
```

```
        soma += valores[i]; // acrescenta o i-ésimo valor à soma.

    double const média = soma / número_de_valores;

    // Divisão pela média:
    for(int i = 0; i != número_de_valores; ++i)
        valores[i] /= média; // divide o i-ésimo valor pela média.

    // Escrita do resultado:
    for(int i = 0; i != número_de_valores; ++i)
        cout << valores[i] << ' '; // escreve o i-ésimo valor.
    cout << endl;
}
```

5.2.1 Definição de vectores

Para utilizar um vector é necessário defini-lo. A sintaxe da definição de vectores é simples:

```
vector<tipo> nome(número_de_itens);
```

em que *tipo* é o tipo de cada item do vector, *nome* é o nome do vector, e *número_de_itens* é o número de itens ou dimensão inicial do novo vector. É possível omitir a dimensão inicial do vector: nesse caso o vector inicialmente não terá qualquer elemento:

```
vector<tipo> nome;
```

Por exemplo:

```
vector<int>    vi(10); // vector com 10 int.
vector<char>   vc(80); // vector com 80 char.
vector<float>  vf(20); // vector com 20 float.
vector<double> vd;     // vector com zero double.
```

Um vector é uma variável como outra qualquer. Depois das definições acima pode-se dizer que `vd` é uma variável do tipo `vector<double>` (vector de `double`) que contém zero itens.

5.2.2 Indexação de vectores

Seja a definição

```
vector<int> v(10);
```

Para atribuir o valor 5 ao sétimo item do vector pode-se escrever:

```
v[6] = 5;
```

Ao valor 6, colocado entre `[]`, chama-se *índice*. Ao escrever numa expressão o nome de um vector seguido de `[]` com um determinado índice está-se a efectuar uma *operação de indexação*, i.e., a aceder a um dado item do vector indicando o seu índice.

Os índices dos vectores, como os das matrizes, são sempre números inteiros e começam sempre em zero. Assim, o primeiro item do vector `v` é `v[0]`, o segundo `v[1]`, e assim sucessivamente.

Os índices usados para aceder aos vectores não necessitam de ser constantes: podem-se usar variáveis, como aliás se pode verificar no exemplo da leitura de 1000 valores apresentado atrás. Assim, o exemplo anterior pode-se escrever:

```
int a = 6;
v[a] = 5; // atribui o inteiro 5 ao 7º item do vector.
```

Da mesma forma que acontece com as matrizes, também com os vectores as indexações não são verificadas. Por exemplo, o código seguinte não resulta em qualquer erro de compilação e provavelmente também não resulta na terminação do programa com uma mensagem de erro, isto apesar de tentar atribuir valores a posições inexistentes do vector:

```
vector<int> v(4);

v[-1] = 1; // erro! só se pode indexar de 0 a 3!
v[4] = 2;  // erro! só se pode indexar de 0 a 3!
```

Esta é uma fonte muito frequente de erros no C++, pelo que se recomenda extremo cuidado na indexação de vectores.

5.2.3 Inicialização de vectores

Os itens de um vector, ao contrário do que acontece com as matrizes, são sempre inicializados. A inicialização usada é a chamada inicialização por omissão, que no caso dos itens serem dos tipos básicos (`int`, `char`, `float`, etc.), é a inicialização com o valor zero. Por exemplo, depois da definição

```
vector<int> v(4);
```

todos os itens do vector `v` têm o valor zero.

Não é possível inicializar os itens de um vector como se inicializam os elementos de uma matriz. Mas pode-se especificar um valor com o qual se pretendem inicializar todos os itens do vector. Por exemplo, depois da definição

```
vector<int> v(4, 13);
```

todos os itens do vector `v` têm o valor 13.

5.2.4 Operações

Os vectores são variáveis como qualquer outras. A diferença principal está em suportarem a invocação das chamadas *operações*, um conceito que será visto em pormenor no Capítulo 7. Simplificando grosseiramente, as operações são rotinas associadas a um tipo (classe C++) que se invocam para uma determinada variável desse tipo. A forma de invocação é um pouco diferente da usada para as rotinas "normais". As operações invocam-se usando o operador `.` de selecção de membro (no Capítulo 7 se verá o que é exactamente um *membro*). Este operador tem dois operandos. O primeiro é a variável à qual a operação é aplicado. O segundo é a operação a aplicar, seguida dos respectivos argumentos, se existirem.

Uma operação extremamente útil do tipo genérico `vector` chama-se `size()` e permite saber a dimensão actual de um vector. Por exemplo:

```
vector<double> distâncias(10);

cout << "A dimensão é " << distâncias.size() << '.' << endl;
```

O valor devolvido pela operação `size()` pertence a um dos tipos inteiros do C++, mas a norma da linguagem não especifica qual... No entanto, é fornecido um sinónimo desse tipo para cada tipo de vector. Esse sinónimo chama-se `size_type` e pode ser acedido usando o operador `::` de resolução de âmbito em que o operando esquerdo é o tipo do vector:

```
vector<double>::size_type
```

Um ciclo para percorrer e mostrar todos os itens de um vector pode ser escrito fazendo uso deste tipo e da operação `size()`:

```
for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    cout << distâncias[i] << endl;
```

Uma outra operação pode ser usada se se pretender apenas saber se um vector está ou não vazio, i.e., se se pretender saber se um vector tem dimensão zero. A operação chama-se `empty()` e devolve verdadeiro se o vector estiver vazio e falso no caso contrário.

5.2.5 Acesso aos itens de vectores

Viu-se que a indexação de vectores é insegura. O tipo genérico vector fornece uma operação chamada `at()` que permite aceder a um item dado o seu índice, tal como a operação de indexação, mas que garantidamente resulta num erro[16] no caso de o índice ser inválido. Esta operação tem como argumento o índice do item ao qual se pretende aceder e devolve esse mesmo item. Voltando ao exemplo original,

[16]Ou melhor, resulta no lançamento de uma *excepção*, ver !!.

```
vector<int> v(4);

v.at(-1) = 1; // erro! só se pode indexar de 0 a 3!
v.at(4) = 2;  // erro! só se pode indexar de 0 a 3!
```

este troço de código levaria à terminação abrupta[17] do programa em que ocorresse logo ao ser executada a primeira indexação errada.

Existem ainda duas operações que permitem aceder aos dois itens nas posições extremas do vector. A operação `front()` devolve o primeiro item do vector. A operação `back()` devolve o último item do vector. Por exemplo, o seguinte troço de código

```
vector<double> distâncias(10);

for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    distâncias[i] = double(i); // converte i num double.

cout << "Primeiro: " << distâncias.front() << '.' << endl;
cout << "Último: " << distâncias.back() << '.' << endl;
```

escreve no ecrã

```
Primeiro: 0.
Último: 9.
```

5.2.6 Alteração da dimensão de um vector

Ao contrário do que acontece com as matrizes, os vectores podem ser redimensionados sempre que necessário. Para isso recorre-se à operação `resize()`. Esta operação recebe como primeiro argumento a nova dimensão pretendida para o vector e, opcionalmente, recebe como segundo argumento o valor com o qual são inicializados os possíveis novos itens criados pelo redimensionamento. Caso o segundo argumento não seja especificado, os possíveis novos itens são inicializados usando a inicialização por omissão. Por exemplo, o resultado de

```
vector<double> distâncias(3, 5.0);
distâncias.resize(6, 10.0);
distâncias.resize(9, 15.0);
distâncias.resize(8);
for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    cout << distâncias[i] << endl;
```

é surgir no ecrã

[17] Isto, claro está, se ninguém capturar a excepção lançada, como se verá no !!.

```
5
5
5
10
10
10
15
15
```

Se a intenção for eliminar todos os itens do vector, reduzindo a sua dimensão a zero, pode-se usar a operação `clear()` como se segue:

```
distâncias.clear();
```

Os vectores não podem ter uma dimensão arbitrária. Para saber a dimensão máxima possível para um vector usar a operação `max_size()`:

```
cout << "Dimensão máxima do vector de distâncias é "
     << distâncias.max_size() << " itens." << endl;
```

Finalmente, como as operações que alteram a dimensão de um vector podem ser lentas, já que envolvem pedidos de memória ao sistema operativo, o tipo genérico fornece uma operação chamada `reserve()` para pré-reservar espaço para itens que se prevê venham a ser necessários no futuro. Esta operação recebe como argumento o número de itens que se prevê virem a ser necessários na pior das hipóteses e reserva espaço para eles. Enquanto a dimensão do vector não ultrapassar a reserva feita todas as operações têm eficiência garantida. A operação `capacity()` permite saber qual o número de itens para os quais há espaço reservado em cada momento. A secção seguinte demonstra a utilização desta operação.

5.2.7 Inserção e remoção de itens

As operações mais simples para inserção e remoção de itens referem-se ao extremo final dos vectores. Se se pretender acrescentar um item no final de um vector pode-se usar a operação `push_back()`, que recebe como argumento o valor com o qual inicializar o novo item. Se se pretender remover o último item de um vector pode-se usar a operação `pop_back()`. Por exemplo, em vez de escrever

```
vector<double> distâncias(10);

for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    distâncias[i] = double(i); // converte i num double.
```

é possível escrever

```
vector<double> distâncias;

for(vector<double>::size_type i = 0; i != 10; ++i)
    distâncias.push_back(double(i));
```

A segunda versão, com `push_back()`, é particularmente vantajosa quando o número de itens a colocar no vector não é conhecido à partida.

Se a dimensão máxima do vector for conhecida à partida, pode ser útil começar por reservar espaço suficiente para os itens a colocar:

```
vector<double> distâncias;
distâncias.reserve(10);
/* Neste local o vector está vazio, mas tem espaço para crescer sem precisar de recor-
rer ao sistema operativo para requerer memória. */
for(vector<double>::size_type i = 0; i != 10; ++i)
    distâncias.push_back(double(i));
```

A operação `pop_back()` pode ser conjugada com a operação `empty()` para mostrar os itens de um vector pela ordem inversa e, simultaneamente, esvaziá-lo:

```
while(not distâncias.empty()) {
    cout << distâncias.back() << endl;
    distâncias.pop_back();
}
```

5.2.8 Vectores multidimensionais?

O tipo genérico vector não foi pensado para representar matrizes multidimensionais. No entanto, é possível definir vectores de vectores usando a seguinte sintaxe, apresentada sem mais explicações[18]:

```
vector<vector<tipo> > nome(linhas, vector<tipo>(colunas));
```

onde *`linhas`* é o número de linhas pretendido para a matriz e *`colunas`* o número de colunas. O processo não é elegante, mas funciona e pode ser estendido para mais dimensões:

```
vector<vector<vector<tipo> > >
    nome(planos, vector<vector<tipo>>(linhas,
                                      vector<tipo>(colunas)));
```

[18]Tem de se deixar um espaço entre símbolos > sucessivos para que não se confundam com o operador > >.

No entanto, estas variáveis não são verdadeiramente representações de matrizes, pois, por exemplo no primeiro caso, é possível alterar a dimensão das linhas da "matriz" independentemente umas das outras.

É possível simplificar as definições acima se o objectivo não for emular as matrizes mas simplesmente definir um vector de vectores. Por exemplo, o seguinte troço de programa

```
/* Definição de um vector de vectores com quatro itens inicialmente, cada um tendo inicialmente dimensão nula. */
vector<vector<char> > v(4);

v[0].resize(1, 'a');
v[1].resize(2, 'b');
v[2].resize(3, 'c');
v[3].resize(4, 'd');

for(vector<vector<char> >::size_type i = 0; i != v.size(); ++i) {
    for(vector<char>::size_type j = 0; j != v[i].size(); ++j)
        cout << v[i][j];
    cout << endl;
}
```

quando executado resulta em

```
a
bb
ccc
dddd
```

5.2.9 Vectores constantes

Como para qualquer outro tipo, pode-se definir vectores constantes. A grande dificuldade está em inicializar um vector constante, visto que a inicialização ao estilo das matrizes não é permitida:

```
vector<char> const muitos_aa(10, 'a');
```

5.2.10 Vectores como parâmetros de rotinas

A passagem de vectores como argumento não difere em nada de outros tipos. Podem-se passar vectores por valor ou por referência. O programa original pode ser modularizado de modo a usar rotinas do seguinte modo:

```
#include <iostream>
#include <vector>

using namespace std;

/** Preenche o vector com valores lidos do teclado.
    @pre  PC ≡ o canal de entrada (cin) contém v.size() números decimais.
    @post CO ≡ os itens do vector v contêm os v.size() valores decimais
               que estavam em cin. */
void lê(vector<double>& v)
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cin >> v[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos itens do vector.
    @pre  PC ≡ 1 ≤ v.size().
    @post CO ≡ média = (S j : 0 ≤ j < v.size() : v[j]) / v.size(). */
double média(vector<double> const v)
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}

/** Divide os itens do vector por divisor.
    @pre  PC ≡ divisor ≠ 0 ∧ v = v.
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] = v[j] / divisor). */
void normaliza(vector<double>& v, double const divisor)
{
    assert(divisor != 0);

    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        v[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores do vector no ecrã.
    @pre  PC ≡ V.
    @post CO ≡ o canal cout contém representações dos itens de v, por ordem. */
void escreve(vector<double> const v)
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cout << v[i] << ' '; // escreve o i-ésimo elemento.
```

```
}

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    vector<double> valores(número_de_valores);
    lê(valores);

    // Divisão pela média:
    normaliza(valores, média(valores));

    // Escrita do resultado:
    escreve(valores);
    cout << endl;
}
```

No entanto, é importante perceber que *a passagem de um vector por valor implica a cópia do vector inteiro*, o que pode ser extremamente ineficiente. O mesmo não acontece se a passagem se fizer por referência, mas seria desagradável passar a mensagem errada ao compilador e ao leitor do código, pois uma passagem por referência implica que a rotina em causa pode alterar o argumento. Para resolver o problema usa-se uma passagem de argumentos por referência constante.

5.2.11 Passagem de argumentos por referência constante

Considere-se de novo função para somar a média de todos os itens de um vector de inteiros:

```
double média(vector<double> const v)
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}
```

De acordo com o mecanismo de chamada de funções (descrito na Secção 3.2.11), e uma vez que o vector é passado por valor, é evidente que a chamada da função `média()` implica a construção de uma nova variável do tipo `vector<int>`, o parâmetro `v`, que é inicializada a partir do vector passado como argumento e que é, portanto, uma cópia desse mesmo argumento. Se o vector passado como argumento tiver muitos itens, como é o caso no programa acima, é evidente que esta cópia pode ser muito demorada, podendo mesmo em alguns casos tornar a utilização da função impossível na prática. Como resolver o problema? Se a passagem do vector fosse feita por referência, e não por valor, essa cópia não seria necessária. Assim, poder-se-ia aumentar a eficiência da chamada da função definindo-a como

```
double média(vector<double>& v) // má ideia!
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}
```

Esta nova versão da função tem uma desvantagem: na primeira versão, o consumidor da função e o compilador sabiam que o vector passado como argumento não poderia ser alterado pela função, já que esta trabalhava com uma cópia. Na nova versão essa garantia não é feita. O problema pode ser resolvido se se disser de algum modo que, apesar de o vector ser passado por referência, a função não está autorizada a alterá-lo. Isso consegue-se recorrendo de novo ao qualificador `const`:

```
double média(vector<double> const& v) // boa ideia!
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}
```

Pode-se agora converter o programa de modo a usar passagens por referência constante em todas as rotinas onde é útil fazê-lo:

```
#include <iostream>
#include <vector>

using namespace std;

/** Preenche o vector com valores lidos do teclado.
    @pre  PC ≡ o canal de entrada (cin) contém v.size() números decimais.
    @post CO ≡ os itens do vector v contêm os v.size() valores decimais
               que estavam em cin. */
void lê(vector<double>& v)
```

```
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cin >> v[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos itens do vector.
    @pre PC ≡ 1 ≤ v.size().
    @post CO ≡ média = (S j : 0 ≤ j < v.size() : v[j]) / v.size(). */
double média(vector<double> const& v)
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}

/** Divide os itens do vector por divisor.
    @pre PC ≡ divisor ≠ 0 ∧ v = v.
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] = v[j] / divisor). */
void normaliza(vector<double>& v, double const divisor)
{
    assert(divisor != 0);

    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        v[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores do vector no ecrã.
    @pre PC ≡ V.
    @post CO ≡ o canal cout contém representações dos itens de v, por ordem. */
void escreve(vector<double> const& v)
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cout << v[i] << ' '; // escreve o i-ésimo elemento.
}

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    vector<double> valores(número_de_valores);
    lê(valores);
```

```
    // Divisão pela média:
    normaliza(valores, média(valores));

    // Escrita do resultado:
    escreve(valores);
    cout << endl;
}
```

Compare-se esta versão do programa com a versão usando matrizes apresentada mais atrás.

As passagens por referência constante têm uma característica adicional que as distingue das passagens por referência simples: permitem passar qualquer constante (e.g., um valor literal) como argumento, o que não era possível no caso das passagens por referência simples. Por exemplo:

```
// Mau código. Bom para exemplos apenas...
int soma1(int& a, int& b)
{
    return a + b;
}

/* Não é grande ideia usar referências constantes para os tipos bási-
cos do C++ (não se poupa nada): */
int soma2(int const& a, int const& b)
{
    return a + b;
}

int main()
{
    int i = 1, j = 2, res;
    res = soma1(i, j); // válido.
    res = soma2(i, j); // válido.
    res = soma1(10, 20); // erro!
    res = soma2(10, 20); // válido! os parâmetros a e b
                         // tornam-se sinónimos de duas
                         // constantes temporárias inicializadas
                         // com 10 e 20.
}
```

A devolução de vectores em funções é possível, ao contrário do que acontece com as matrizes, embora não seja recomendável. A devolução de um vector implica também uma cópia: o valor devolvido é uma cópia do vector colocado na expressão de retorno. O problema pode, por vezes, ser resolvido também por intermédio de referências, como se verá na Secção 7.7.1.

5.2.12 Outras operações com vectores

Ao contrário do que acontece com as matrizes, é possível atribuir vectores e mesmo compará-los usando os operadores de igualdade e os operadores relacionais. A única particularidade merecedora de referência é o que se entende por um vector ser menor do que outro.

A comparação entre vectores com os operadores relacionais é feita usando a chamada *ordenação lexicográfica*. Esta ordenação é a mesma que é usada para colocar as palavras nos vulgares dicionários[19], e usa os seguintes critérios:

1. A comparação é feita da esquerda para a direita, começando portanto nos primeiros itens dos dois vectores em comparação, e prosseguindo sempre a par ao longo dos vectores.
2. A comparação termina assim que ocorrer uma de três condições:
   (a) os itens em comparação são diferentes: nesse caso o vector com o item mais pequeno é considerado menor que o outro.
   (b) um dos vectores não tem mais itens, sendo por isso mais curto que o outro: nesse caso o vector mais curto é considerado menor que o outro.
   (c) nenhum dos vectores tem mais itens: nesse caso os dois vectores têm o mesmo comprimento e os mesmos valores dos itens e são por isso considerados iguais.

Representando os vectores por tuplos:

- $() = ()$.
- $() < (10)$.
- $(1, 10, 20) < (2)$.
- $(1, 2, 3, 4) < (1, 3, 2, 4)$.
- $(1, 2) < (1, 2, 3)$.

O troço de código abaixo

```
vector<int> v1;

v1.push_back(1);
v1.push_back(2);

vector<int> v2;
```

[19] De acordo com [4]:
lexicográfico (cs). [De *lexicografia* + *-ico*^{2}.] ***Adj.*** Respeitante à lexicografia.
lexicografia (cs). [De *lexico-* + *-grafia*.] ***S.f.*** A ciência do lexicógrafo.
lexicógrafo (cs). [Do gr. *lexikográphos*.] ***S.m.*** Autor de dicionário ou de trabalho a respeito de palavras duma língua; dicionarista; lexicólogo. [Cf. *lexicografo*, do v. *lexicografar*.]

```
v2.push_back(1);
v2.push_back(2);
v2.push_back(3);

if(v1 < v2)
    cout << "v1 é mesmo menor que v2." << endl;
```

produz no ecrã

```
v1 é mesmo menor que v2.
```

como se pode verificar observando o último exemplo acima.

5.3 Algoritmos com matrizes e vectores

Esta secção apresenta o desenvolvimento pormenorizado de quatro funções usando ciclos e operando com matrizes e vectores e serve, por isso, de complemento aos exemplos de desenvolvimento de ciclos apresentados no capítulo anterior.

5.3.1 Soma dos elementos de uma matriz

O objectivo da função é calcular a soma dos valores dos primeiros `n` elementos de uma matriz `m`.

O primeiro passo da resolução do problema é a sua especificação, ou seja, a escrita da estrutura da função, incluindo a pré-condição e a condição objectivo:

```
/** Devolve a soma dos primeiros n elementos da matriz m.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ soma = (S j : 0 ≤ j < n : m[j]). */
int soma(int const m[], int const n)
{
    assert(0 <= n);

    int soma = ...;
    ...
    return soma;
}
```

Como é necessário no final devolver a soma dos elementos, define-se imediatamente uma variável `soma` para guardar esse valor. Usa-se uma variável com o mesmo nome da função para que a condição objectivo do ciclo e a condição objectivo da função sejam idênticas[20].

[20] O nome de uma variável local pode ser igual ao da função em que está definida. Nesse caso ocorre uma ocultação do nome da função (ver Secção 3.2.14). A única consequência desta ocultação é que para invocar recursivamente a função, se isso for necessário, é necessário usar o operador de resolução de âmbito `::`.

O passo seguinte consiste em, tendo-se percebido que a solução pode passar pela utilização de um ciclo, determinar uma condição invariante apropriada, o que se consegue usualmente por enfraquecimento da condição objectivo. Neste caso pode-se simplesmente substituir o limite superior do somatório (a constante `n`) por uma variável `i` inteira, limitada a um intervalo apropriado de valores. Assim, a estrutura do ciclo é

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = ...;
int i = ...;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(G) {
    passo
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

Identificada a condição invariante, é necessário escolher uma guarda tal que seja possível demonstrar que $CI \wedge \neg G \Rightarrow CO$. Ou seja, escolher uma guarda que, quando o ciclo terminar, conduza naturalmente à condição objectivo[21]. Neste caso é evidente que a escolha correcta para $\neg G$ é $\mathtt{i} = \mathtt{n}$, ou seja, $G \equiv \mathtt{i} \neq \mathtt{n}$:

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = ...;
int i = ...;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(i != n) {
    passo
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

De seguida deve-se garantir, por escolha apropriada das instruções das inicializações, que a condição invariante é verdadeira no início do ciclo. Como sempre, devem-se escolher as instruções mais simples que conduzem à veracidade da condição invariante. Neste caso é fácil verificar que se deve inicializar `i` com zero e `soma` também com zero (recorde-se de que a soma de zero elementos é, por definição, zero):

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = 0;
int i = 0;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(i != n) {
    passo
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

[21] A condição invariante é verdadeira, por construção, no início, durante, e no final do ciclo: por isso se chama condição invariante. Quando o ciclo termina, tem de se ter forçosamente que a guarda é falsa, ou seja, que $\neg G$ é verdadeira.

Como se pretende que o algoritmo termine, deve-se agora escolher um progresso que o garanta (o passo é normalmente dividido em duas partes, o progresso *prog* e a acção *acção*). Sendo `i` inicialmente zero, e sendo $0 \leq \mathtt{n}$ (pela pré-condição), é evidente que uma simples incrementação da variável `i` conduzirá à falsidade da guarda, e portanto à terminação do ciclo, ao fim de exactamente `n` iterações do ciclo:

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = 0;
int i = 0;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(i != n) {
    acção
    ++i;
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

Finalmente, é necessário construir uma acção que garanta a veracidade da condição invariante depois do passo e *apesar* do progresso entretanto realizado. Sabe-se que a condição invariante e a guarda são verdadeiras antes do passo, logo, é necessário encontrar uma acção tal que:

```
// CI ∧ G ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n ∧ i ≠ n, ou seja,
// soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i < n.
acção
++i;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
```

Pode-se começar por verificar qual a pré-condição mais fraca do progresso que garante que a condição invariante é recuperada:

```
// soma = (S j : 0 ≤ j < i + 1 : m[j]) ∧ 0 ≤ i + 1 ≤ n, ou seja,
// soma = (S j : 0 ≤ j < i + 1 : m[j]) ∧ −1 ≤ i < n.
++i;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
```

Se se admitir que $0 \leq \mathtt{i}$, então o último termo do somatório pode ser extraído:

```
// soma = (S j : 0 ≤ j < i : m[j]) + m[i] ∧ 0 ≤ i < n.
// soma = (S j : 0 ≤ j < i + 1 : m[j]) ∧ −1 ≤ i < n.
```

Conclui-se que a acção deverá ser escolhida de modo a que:

```
// soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i < n.
acção
// soma = (S j : 0 ≤ j < i : m[j]) + m[i] ∧ 0 ≤ i < n.
```

o que se consegue facilmente com a acção:

```
soma += m[i];
```

A função completa é então

```
/** Devolve a soma dos primeiros n elementos da matriz m.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ soma = (S j : 0 ≤ j < n : m[j]). */
int soma(int const m[], int const n)
{
    assert(0 <= n);

    int soma = 0;
    int i = 0;
    // CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        soma += m[i];
        ++i;
    }
    return soma;
}
```

Pode-se ainda converter o ciclo de modo a usar a instrução `for`:

```
/** Devolve a soma dos primeiros n elementos da matriz m.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ soma = (S j : 0 ≤ j < n : m[j]). */
int soma(int const m[], int const n)
{
    assert(0 <= n);

    int soma = 0;
    // CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        soma += m[i];
    return soma;
}
```

Manteve-se a condição invariante como um comentário antes do ciclo, pois é muito importante para a sua compreensão. A condição invariante no fundo reflecte como o ciclo (e a função) funcionam, ao contrário da pré-condição e da condição objectivo, que se referem àquilo que o ciclo (ou a função) faz. A pré-condição e a condição objectivo são úteis para o programador consumidor da função, enquanto a condição invariante é útil para o programador produtor e para o programador "assistência técnica", que pode precisar de verificar a correcta implementação do ciclo.

O ciclo desenvolvido corresponde a parte da função `média()` usada em exemplos anteriores.

5.3.2 Soma dos itens de um vector

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. A função resultante desse desenvolvimento é

```
/** Devolve a soma dos itens do vector v.
    @pre PC ≡ V.
    @post CO ≡ soma = (S j : 0 ≤ j < v.size() : v[j]). */
int soma(vector<int> const& v)
{
    int soma = 0;
    // CI ≡ soma = (S j : 0 ≤ j < i : v[j]) ∧ 0 ≤ i ≤ v.size().
    for(vector<int>::size_type i = 0; i != v.size(); ++i)
        soma += v[i];
    return soma;
}
```

5.3.3 Índice do maior elemento de uma matriz

O objectivo é construir uma função que devolva um dos índices do máximo valor contido nos primeiros `n` elementos de uma matriz `m`. Como não se especifica qual dos índices devolver caso existam vários elementos com o valor máximo, arbitra-se que a função deve devolver o primeiro desses índices. Assim, a estrutura da função é:

```
/** Devolve o índice do primeiro elemento com o máximo valor entre os primeiros
    n elementos da matriz m.
    @pre PC ≡ 1 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < n∧
              (Q j : 0 ≤ j < n : m[j] ≤ m[índiceDoPrimeiroMáximoDe]) ∧
              (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : m[j] < m[índiceDoPrimeiroMáximoDe]).
int índiceDoPrimeiroMáximoDe(int const m[], int const n)
{
    assert(1 <= n);

    int i = ...;
    ...

    // Sem ciclos não se pode fazer muito melhor:
    assert(0 <= i < n and m[0] <= m[i]);

    return i;
}
```

A condição objectivo indica que o índice devolvido tem de pertencer à gama de índices válidos para a matriz, que o valor do elemento de `m` no índice devolvido tem de ser maior ou igual aos valores de todos os elementos da matriz (estas condições garantem que o índice devolvido é um dos índices do valor máximo na matriz) e que os valores dos elementos com índice menor do que o índice devolvido têm de ser estritamente menores que o valor da matriz no índice devolvido (ou seja, o índice devolvido é o primeiro dos índices dos elementos com valor máximo na matriz). A pré-condição, neste caso, impõe que `n` não pode ser zero, pois não tem sentido falar do máximo de um conjunto vazio, além de obrigar `n` a ser inferior ou igual à dimensão da matriz.

É evidente que a procura do primeiro máximo de uma matriz pode recorrer a um ciclo. A estrutura do ciclo é pois:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = ...;
while(G) {
    passo
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

onde a condição objectivo do ciclo se refere à variável `i`, a ser devolvida pela função no seu final.

Os dois primeiros passos da construção de um ciclo, obtenção da condição invariante e da guarda, são, neste caso, idênticos aos do exemplo anterior: substituição da constante `n` por uma nova variável `k`:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = ...;
int k = ...;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
while(k != n) {
    passo
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

A condição invariante indica que a variável `i` contém sempre o índice do primeiro elemento cujo valor é o máximo dos valores contidos nos `k` primeiros elementos da matriz.

A inicialização a usar também é simples, embora desta vez não se possa inicializar `k` com 0, pois não existe máximo de um conjunto vazio (não se poderia atribuir qualquer valor a `i`)! Assim, a solução é inicializar `k` com 1 e `i` com 0:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
int k = 1;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
```

```
while(k != n) {
    passo
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

Analisando os termos da condição invariante um a um, verifica-se que:

1. $0 \leq \texttt{i} < \texttt{k}$ fica $0 \leq 0 < 1$ que é verdadeiro;
2. $(\mathbf{Q}\, j : 0 \leq j < \texttt{k} : \texttt{m}[j] \leq \texttt{m[i]})$ fica $(\mathbf{Q}\, j : 0 \leq j < 1 : \texttt{m}[j] \leq \texttt{m}[0])$, que é o mesmo que $\texttt{m}[0] \leq \texttt{m}[0]$, que é verdadeiro;
3. $(\mathbf{Q}\, j : 0 \leq j < \texttt{i} : \texttt{m}[j] < \texttt{m[i]})$ fica $(\mathbf{Q}\, j : 0 \leq j < 0 : \texttt{m}[j] < \texttt{m}[0])$ que, como existem zero termos no quantificador, tem valor verdadeiro por definição; e
4. $0 \leq \texttt{k} \leq \texttt{n}$ fica $0 \leq 1 \leq \texttt{n}$, que é verdadeira desde que a pré-condição o seja, o que se admite acontecer;

isto é, a inicialização leva à veracidade da condição invariante, como se pretendia.

O passo seguinte é a determinação do progresso. Mais uma vez usa-se a simples incrementação de `k` em cada passo, que conduz forçosamente à terminação do ciclo:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
int k = 1;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
while(k != n) {
    acção
    ++k;
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

A parte mais interessante deste exemplo é a determinação da acção a utilizar para manter a condição invariante verdadeira apesar do progresso. A acção tem de ser tal que

```
// CI∧G ≡ 0 ≤ i < k∧(Q j : 0 ≤ j < k : m[j] ≤ m[i])∧(Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//        0 ≤ k ≤ n ∧ k ≠ n, ou seja,
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//        0 ≤ k < n.
acção
++k;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
```

Mais uma vez começa-se por encontrar a pré-condição mais fraca do progresso:

// $0 \leq \mathtt{i} < \mathtt{k}+1 \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}+1\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge$
// $\quad 0 \leq \mathtt{k}+1 \leq \mathtt{n}$, ou seja,
// $0 \leq \mathtt{i} \leq \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}+1\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge$
// $\quad -1 \leq \mathtt{k} < \mathtt{n}$.
`++k;`
// $CI \equiv 0 \leq \mathtt{i} < \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge$
// $\quad 0 \leq \mathtt{k} \leq \mathtt{n}$.

Se se admitir que $0 \leq \mathtt{k}$, então o último termo do primeiro quantificador universal pode ser extraído:

// $0 \leq \mathtt{i} \leq \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge \mathtt{m}[\mathtt{k}] \leq \mathtt{m}[\mathtt{i}] \wedge$
// $\quad (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge 0 \leq \mathtt{k} < \mathtt{n}$.
// $0 \leq \mathtt{i} \leq \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}+1\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge$
// $\quad -1 \leq \mathtt{k} < \mathtt{n}$.

Conclui-se que a acção deverá ser escolhida de modo a que:

// $0 \leq \mathtt{i} < \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge$
// $\quad 0 \leq \mathtt{k} < \mathtt{n}$.
acção
// $0 \leq \mathtt{i} \leq \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge \mathtt{m}[\mathtt{k}] \leq \mathtt{m}[\mathtt{i}] \wedge$
// $\quad (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge 0 \leq \mathtt{k} < \mathtt{n}$.

É claro que a acção deverá afectar apenas a variável `i`, pois a variável `k` é afectada pelo progresso. Mas como? Haverá alguma circunstância em que não seja necessária qualquer alteração da variável `i`, ou seja, em que a acção possa ser a instrução nula? Comparando termo a termo as asserções antes e depois da acção, conclui-se que isso só acontece se $\mathtt{m}[\mathtt{k}] \leq \mathtt{m}[\mathtt{i}]$. Então a acção deve consistir numa instrução de selecção:

// $0 \leq \mathtt{i} < \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge$
// $\quad 0 \leq \mathtt{k} < \mathtt{n}$.
`if(m[k] <= m[i])`
  // $G_1 \equiv \mathtt{m}[\mathtt{k}] \leq \mathtt{m}[\mathtt{i}]$.
  `;` // instrução nula!
`else`
  // $G_2 \equiv \mathtt{m}[\mathtt{i}] < \mathtt{m}[\mathtt{k}]$.
  $\mathtt{instrução}_2$

// $0 \leq \mathtt{i} \leq \mathtt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{k}\, :\, \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}]) \wedge \mathtt{m}[\mathtt{k}] \leq \mathtt{m}[\mathtt{i}] \wedge$
// $\quad (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \wedge 0 \leq \mathtt{k} < \mathtt{n}$.

Resta saber que instrução deve ser usada para resolver o problema no caso em que $\mathtt{m}[\mathtt{i}] < \mathtt{m}[\mathtt{k}]$. Falta, pois, falta determinar uma $\mathtt{instrução}_2$ tal que:

```
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k < n ∧ m[i] < m[k].
instrução₂
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
```

Antes da instrução, `i` contém o índice do primeiro elemento contendo o maior dos valores dos elementos com índices entre 0 e `k` *exclusive*. Por outro lado, o elemento de índice `k` contém um valor superior ao valor do elemento de índice `i`. Logo, há um elemento entre 0 e `k` *inclusive* com um valor superior a todos os outros: o elemento de índice `k`. Assim, a variável `i` deverá tomar o valor `k`, de modo a continuar a ser o índice do elemento com maior valor entre os valores inspeccionados. A instrução a usar é portanto:

```
i = k;
```

A ideia é que, quando se atinge um elemento com valor maior do que aquele que se julgava até então ser o máximo, deve-se actualizar o índice do máximo. Para verificar que assim é, calcule-se a pré-condição mais fraca que conduz à asserção final pretendida:

```
// 0 ≤ k ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[k]) ∧ m[k] ≤ m[k]∧
//       (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n, ou seja,
// (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n.
i = k;
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
```

Falta pois verificar se

```
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k < n ∧ m[i] < m[k]
// ⇒ (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n.
```

Eliminando os termos que não são necessários para verificar a implicação,

```
// (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ 0 ≤ k < n ∧ m[i] < m[k]
// ⇒ (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n.
```

é evidente que a implicação é verdadeira e, portanto, a atribuição `i = k;` resolve o problema. Assim, a acção do ciclo é a instrução de selecção

```
if(m[k] <= m[i])
    // G₁ ≡ m[k] ≤ m[i].
    ; // instrução nula!
else
    // G₂ ≡ m[i] < m[k].
    i = k;
```

que pode ser simplificada para uma instrução condicional mais simples

```
if(m[i] < m[k])
    i = k;
```

O ciclo completo fica

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
int k = 1;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
while(k != n) {
    if(m[i] < m[k])
        i = k;
    ++k;
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

que pode ser convertido para um ciclo `for`:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
for(int k = 1; k != n; ++k)
    if(m[i] < m[k])
        i = k;
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

A função completa é:

```
/** Devolve o índice do primeiro elemento com o máximo valor entre os primeiros
    n elementos da matriz m.
    @pre PC ≡ 1 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < n∧
              (Q j : 0 ≤ j < n : m[j] ≤ m[índiceDoPrimeiroMáximoDe]) ∧
              (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : m[j] < m[índiceDoPrimeiroMáximoDe]).
int índiceDoPrimeiroMáximoDe(int const m[], int const n)
{
    assert(1 <= n);

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧
    //      (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k ≤ n.
```

```
    for(int k = 1; k != n; ++k)
        if(m[i] < m[k])
            i = k;

    // Sem ciclos não se pode fazer muito melhor (amostragem em três locais):
    assert(0 <= i < n and
           m[0] <= m[i] and m[n / 2] <= m[i] and
           m[n - 1] <= m[i]);

    return i;
}
```

5.3.4 Índice do maior item de um vector

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. A função resultante desse desenvolvimento é

```
/** Devolve o índice do primeiro item com o máximo valor do vector v.
    @pre PC ≡ 1 ≤ v.size().
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < v.size()∧
              (Q j : 0 ≤ j < v.size() : v[j] ≤ v[índiceDoPrimeiroMáximoDe]) ∧
              (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : v[j] < v[índiceDoPrimeiroMáximoDe]).
int índiceDoPrimeiroMáximo(vector<int> const& v)
{
    assert(1 <= v.size());

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : v[j] ≤ v[i]) ∧
    //      (Q j : 0 ≤ j < i : v[j] < v[i]) ∧ 0 ≤ k ≤ v.size().
    for(vector<int>::size_type k = 1; k != v.size(); ++k)
        if(v[i] < v[k])
            i = k;

    // Sem ciclos não se pode fazer muito melhor (amostragem em três locais):
    assert(0 <= i < v.size() and
           m[0] <= m[i] and m[v.size() / 2] <= m[i] and
           m[v.size() - 1] <= m[i]);

    return i;
}
```

5.3.5 Elementos de uma matriz num intervalo

Pretende-se escrever uma função que devolva o valor lógico verdadeiro se e só se os valores dos `n` primeiros elementos de uma matriz `m` estiverem entre `mínimo` e `máximo` (*inclusive*).

Neste caso a estrutura da função e a sua especificação (i.e., a sua pré-condição e a sua condição objectivo) são mais fáceis de escrever:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre  PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    ...
}
```

Uma vez que esta função devolve um valor booleano, que apenas pode ser V ou F, vale a pena verificar em que circunstâncias cada uma das instruções de retorno

```
return false;
return true;
```

resolve o problema. Começa por se verificar a pré-condição mais fraca da primeira destas instruções:

```
// CO ≡ F = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo), ou seja,
// (E j : 0 ≤ j < n : m[j] < mínimo ∨ máximo < m[j]).
return false;
// CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo).
```

Consequentemente, deve-se devolver falso se existir um elemento da matriz fora da gama pretendida.

Depois verifica-se a pré-condição mais fraca da segunda das instruções de retorno:

```
// CO ≡ V = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo), ou seja,
// (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo).
return true;
// CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo).
```

Logo, deve-se devolver verdadeiro se todos os elementos da matriz estiverem na gama pretendida.

Que ciclo resolve o problema? Onde colocar, se é que é possível, estas instruções de retorno? Seja a condição invariante do ciclo:

$$CI \equiv (\mathbf{Q}\, j\ :\ 0 \leq j < \mathtt{i}\ :\ \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n},$$

onde `i` é uma variável introduzida para o efeito. Esta condição invariante afirma que todos os elementos inspeccionados até ao momento (com índices inferiores a `i`) estão na gama pretendida. Esta condição invariante foi obtida intuitivamente, e não através da metodologia de Dijkstra. Em particular, esta condição invariante obriga o ciclo a terminar de uma forma pouco usual se se encontrar um elemento fora da gama pretendida, como se verá mais abaixo.

Se a guarda for

$$G \equiv \mathtt{i} \neq \mathtt{n},$$

então no final do ciclo tem-se $CI \wedge \neg G$, ou seja,

$$CI \wedge \neg G \equiv \left(\mathbf{Q}\, j\,:\, 0 \leq j < \mathtt{i}\,:\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}\right) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{n},$$

que implica

$$\left(\mathbf{Q}\, j\,:\, 0 \leq j < \mathtt{n}\,:\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}\right).$$

Logo, a instrução

```
return true;
```

deve terminar a função.

Que inicialização usar? A forma mais simples de tornar verdadeira a condição invariante é inicializar `i` com 0, pois o quantificador "qualquer que seja" sem qualquer termo tem valor lógico verdadeiro. Pode-se agora acrescentar à função o ciclo parcialmente desenvolvido:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    int i = 0;
    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ m[j] ≤ máximo) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        passo
    }

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
           (mínimo <= m[0] <= máximo and
           (mínimo <= m[n / 2] <= máximo and
           (mínimo <= m[n - 1] <= máximo));

    return true;
}
```

Que progresso usar? Pelas razões habituais, o progresso mais simples a usar é:

```
++i;
```

Resta determinar a acção de modo a que a condição invariante seja de facto invariante. Ou seja, é necessário garantir que

// $CI \wedge G \equiv (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n}$, ou seja,
// $(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n}$.
acção
++i;
// $CI \equiv (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Verificando qual a pré-condição mais fraca que, depois do progresso, conduz à veracidade da condição invariante, conclui-se:

// $(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i}+1 \leq \mathtt{n}$, ou seja,
// $(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.
++i;
// $CI \equiv (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Se se admitir que $0 \leq \mathtt{i}$, então o último termo do quantificador universal pode ser extraído:

// $(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge \mathtt{mínimo} \leq \mathtt{m}[\mathtt{i}] \leq \mathtt{máximo} \wedge 0 \leq \mathtt{i} < \mathtt{n}$.
// $(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.

Conclui-se facilmente que, se $\mathtt{mínimo} \leq \mathtt{m}[\mathtt{i}] \leq \mathtt{máximo}$, então não é necessária qualquer acção para que a condição invariante se verifique depois do progresso. Isso significa que a acção consiste numa instrução de selecção:

// $(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n}$.
if(mínimo <= m[i] and m[i] <= máximo)
    // $G_1 \equiv \mathtt{mínimo} \leq \mathtt{m}[\mathtt{i}] \leq \mathtt{máximo}$.
    ; // instrução nula!
else
    // $G_2 \equiv \mathtt{m}[\mathtt{i}] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[\mathtt{i}]$.
$\mathtt{instrução}_2$
++i;
// $CI \equiv (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Resta pois verificar que instrução deve ser usada na alternativa da instrução de selecção. Para isso começa por se verificar que, antes dessa instrução, se verificam simultaneamente a condição invariante a guarda e a segunda guarda da instrução de selecção, ou seja,

$$\begin{aligned} CI \wedge G \wedge G_2 \quad \equiv \quad & (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n} \wedge \\ & (\mathtt{m}[\mathtt{i}] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[\mathtt{i}]), \end{aligned}$$

Traduzindo para português vernáculo: "os primeiros i elementos da matriz estão dentro da gama pretendida mas o $\mathtt{i}+1$-ésimo (de índice i) não está". É claro portanto que

$$\begin{aligned} CI \wedge G \wedge G_2 &\Rightarrow (\mathbf{E}\, j\, :\, 0 \le j < \mathtt{i}+1 : \mathtt{m}[j] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[j]) \wedge 0 \le \mathtt{i} < \mathtt{n} \\ &\Rightarrow (\mathbf{E}\, j\, :\, 0 \le j < \mathtt{n} : \mathtt{m}[j] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[j]) \end{aligned}$$

Ou seja, existe pelo menos um elemento com índice entre 0 e i *inclusive* que não está na gama pretendida e portanto o mesmo se passa para os elementos com índices entre 0 e n *exclusive*.

Conclui-se que a instrução alternativa da instrução de selecção deve ser

```
return false;
```

pois termina imediatamente a função (e portanto o ciclo), devolvendo o valor apropriado (ver pré-condição mais fraca desta instrução mais atrás).

A acção foi escolhida de tal forma que, ou termina o ciclo devolvendo o valor apropriado (falso), ou garante a validade da condição invariante apesar do progresso.

A função completa é:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    int i = 0;
    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ m[j] ≤ máximo) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        if(mínimo <= m[i] and m[i] <= máximo)
            ; // instrução nula!
        else
            return false;
        ++i;
    }

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
           (mínimo <= m[0] <= máximo and
           (mínimo <= m[n / 2] <= máximo and
           (mínimo <= m[n - 1] <= máximo));

    return true;
}
```

Trocando as instruções alternativas da instrução de selecção (e convertendo-a numa instrução condicional) e convertendo o ciclo `while` num ciclo `for` obtém-se a versão final da função:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre  PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ m[j] ≤ máximo) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        if(m[i] < mínimo or máximo < m[i])
            return false;

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
           (mínimo <= m[0] <= máximo and
           (mínimo <= m[n / 2] <= máximo and
           (mínimo <= m[n - 1] <= máximo));

    return true;
}
```

5.3.6 Itens de um vector num intervalo

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. A função resultante desse desenvolvimento é

```
/** Devolve verdadeiro se os itens do vector v têm valores entre mínimo e máximo.
    @pre  PC ≡ V.
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < v.size() : mínimo ≤ v[j] ≤ máximo).
*/
bool estáEntre(vector<int> const& v,
               int const mínimo, int const máximo)
{
    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ v[j] ≤ máximo) ∧ 0 ≤ i ≤ v.size().
    for(vector<int>::size_type i = 0; i != v.size(); ++i)
        if(v[i] < mínimo or máximo < v[i])
            return false;

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
```

```
                (mínimo <= m[0] <= máximo and
                (mínimo <= m[n / 2] <= máximo and
                (mínimo <= m[n - 1] <= máximo));

        return true;
    }
```

5.3.7 Segundo elemento de uma matriz com um dado valor

O objectivo agora é encontrar o índice do segundo elemento com valor `k` nos primeiros `n` elementos de uma matriz `m`.

Neste caso a pré-condição é um pouco mais complicada do que para os exemplos anteriores, pois tem de se garantir que existem pelo menos dois elementos com o valor pretendido, o que, por si só, implica que a matriz tem de ter pelo menos dois elementos.

A condição objectivo é mais simples. Afirma que o índice a devolver deve corresponder a um elemento com valor `k` e que, no conjunto dos elementos com índice menor, existe apenas um elemento com valor `k` (diz ainda que o índice deve ser válido, neste caso maior do que 0, porque têm de existir pelo menos dois elementos com valores iguais até ao índice). Assim, a estrutura da função é:

/** Devolve o índice do segundo elemento com valor `k` nos primeiros `n` elementos da matriz `m`.
@pre $PC \equiv 2 \leq \mathtt{n} \wedge \mathtt{n} \leq \dim(\mathtt{m}) \wedge 2 \leq (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{n}\, :\, \mathtt{m}[j] = \mathtt{k})$.
@post $CO \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{índiceDoSegundo}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge$
$1 \leq \mathtt{índiceDoSegundo} < \mathtt{n} \wedge \mathtt{m}[\mathtt{índiceDoSegundo}] = \mathtt{k}$. */

```
int índiceDoSegundo(int const m[], int const n, int const k)
{
    int i = ...;
    ...
    return i;
}
```

Para resolver este problema é necessário um ciclo, que pode ter a seguinte estrutura:

// $PC \equiv 2 \leq \mathtt{n} \wedge \mathtt{n} \leq \dim(\mathtt{m}) \wedge 2 \leq (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{n}\, :\, \mathtt{m}[j] = \mathtt{k})$.
```
int i = ...;
while(G) {
    passo
}
```
// $CO \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{m}[\mathtt{i}] = \mathtt{k}$.

Neste caso não existe na condição objectivo do ciclo um quantificador onde se possa substituir (com facilidade) uma constante por uma variável. Assim, a determinação da condição objectivo pode ser tentada factorizando a condição objectivo, que é uma conjunção, em CI e $\neg G$.

Uma observação atenta das condições revela que a escolha apropriada é

$$CO \equiv \overbrace{(\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{i}\ :\ \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}}^{CI} \wedge \overbrace{\mathtt{m[i]} = \mathtt{k}}^{\neg G}$$

Que significa esta condição invariante? Simplesmente que, durante todo o ciclo, tem de se garantir que há um único elementos da matriz com valor `k` e com índice entre 0 e `i` *exclusive*.

O ciclo neste momento é

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
while(m[i] != k) {
    passo
}
// CO ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n ∧ m[i] = k.
```

Um problema com a escolha que se fez para a condição invariante é que, aparentemente, não é fácil fazer a inicialização: como escolher um valor para `i` tal que existe um elemento de valor `k` com índice inferior a `i`? Em vez de atacar imediatamente esse problema, adia-se o problema e assume-se que a inicialização está feita. O passo seguinte, portanto, é determinar o passo do ciclo. Antes do passo sabe-se que $CI \wedge G$, ou seja:

$$CI \wedge G \equiv (\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{i}\ :\ \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{m[i]} \neq \mathtt{k}.$$

Mas

$$(\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{i}\ :\ \mathtt{m}[j] = \mathtt{k}) = 1 \wedge \mathtt{m[i]} \neq \mathtt{k}$$

é o mesmo que

$$(\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{i} + 1\ :\ \mathtt{m}[j] = \mathtt{k}) = 1,$$

pois, sendo $\mathtt{m[i]} \neq \mathtt{k}$, pode-se estender a gama de valores tomados por j sem afectar a contagem de afirmações verdadeiras:

$$CI \wedge G \equiv (\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{i} + 1\ :\ \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}.$$

Atente-se bem na expressão acima. Será que pode ser verdadeira quando `i` atinge o seu maior valor possível de acordo com o segundo termo da conjunção, i.e., quando $\mathtt{i} = \mathtt{n} - 1$? Sendo $\mathtt{i} = \mathtt{n} - 1$, o primeiro termo da conjunção fica

$$(\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{n}\ :\ \mathtt{m}[j] = \mathtt{k}) = 1,$$

o que não pode acontecer, dada a pré-condição! Logo $\mathtt{i} \neq \mathtt{n} - 1$, e portanto

$$CI \wedge G \equiv (\mathbf{N}\, j\ :\ 0 \leq j < \mathtt{i} + 1\ :\ \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} - 1.$$

O passo tem de ser escolhido de modo a garantir a invariância da condição invariante, ou seja, de modo a garantir que

// $CI \wedge G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}-1$.
passo
// $CI \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.

Começa por se escolher um progresso apropriado. Qual a forma mais simples de garantir que a guarda se torna falsa ao fim de um número finito de passos? Simplesmente incrementando `i`. Se `i` atingisse alguma vez o valor `n` (índice para além do fim da matriz) sem que a guarda se tivesse alguma vez tornado falsa, isso significaria que a matriz não possuía pelo menos dois elementos com o valor `k`, o que violaria a pré-condição. Logo, o ciclo tem de terminar antes de `i` atingir `n`, ao fim de um número finito de passos, portanto. O passo do ciclo pode então ser escrito como:

// $CI \wedge G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}-1$.
acção
`++i;`
// $CI \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.

Determinando a pré-condição mais fraca do progresso que conduz à verificação da condição invariante no seu final,

// $(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i}+1 < \mathtt{n}$, ou seja,
// $(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 0 \leq \mathtt{i} < \mathtt{n}-1$.
`++i;`
// $CI \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.

Assim sendo, a acção terá de ser escolhida de modo a garantir que

// $CI \wedge G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}-1$.
acção
// $(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}+1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 0 \leq \mathtt{i} < \mathtt{n}-1$.

Mas isso consegue-se sem necessidade de qualquer acção, pois

$$1 \leq \mathtt{i} < \mathtt{n}-1 \Rightarrow 0 \leq \mathtt{i} < \mathtt{n}-1$$

O ciclo completo é

// $PC \equiv 2 \leq \mathrm{n} \wedge \mathrm{n} \leq \dim(\mathrm{m}) \wedge 2 \leq (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{n}\, :\, \mathtt{m}[j] = \mathtt{k})$.
```
int i = ...;
```
// $CI \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.
```
while(m[i] != k)
    ++i
```
// $CO \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{m}[\mathtt{i}] = \mathtt{k}$.

E a inicialização?

A inicialização do ciclo anterior é um problema por si só, com as mesmas pré-condições, mas com uma outra condição objectivo, igual à condição invariante do ciclo já desenvolvido. Isto é, o problema a resolver é:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
```

Pretende-se que `i` seja maior do que o índice da primeira ocorrência de `k` na matriz. A solução para este problema é mais simples se se reforçar a sua condição objectivo (que é a condição invariante do ciclo anterior) um pouco mais. Pode-se impor que `i` seja o índice imediatamente após a primeira ocorrência de `k`:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ m[i − 1] = k ∧ 1 ≤ i < n.
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
```

Pode-se simplificar ainda mais o problema se se terminar a inicialização com uma incrementação de `i` e se calcular a pré-condição mais fraca dessa incrementação:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// (N j : 0 ≤ j < i + 1 : m[j] = k) = 1 ∧ m[i] = k ∧ 1 ≤ i + 1 < n, ou seja,
// (N j : 0 ≤ j < i + 1 : m[j] = k) = 1 ∧ m[i] = k ∧ 0 ≤ i < n − 1.
++i;
// (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ m[i − 1] = k ∧ 1 ≤ i < n.
```

Sendo $0 \leq \texttt{i} < \texttt{n} - 1$, então

$$(\mathbf{N}\, j\ :\ 0 \leq j < \texttt{i} + 1\ :\ \texttt{m}[j] = \texttt{k}) = 1 \wedge \texttt{m}[\texttt{i}] = \texttt{k} \wedge 0 \leq \texttt{i} < \texttt{n} - 1$$

é o mesmo que

$$(\mathbf{N}\, j\ :\ 0 \leq j < \texttt{i}\ :\ \texttt{m}[j] = \texttt{k}) = 0 \wedge \texttt{m}[\texttt{i}] = \texttt{k} \wedge 0 \leq \texttt{i} < \texttt{n} - 1$$

ou ainda (ver Apêndice A)

$$(\mathbf{Q}\, j\ :\ 0 \leq j < \texttt{i}\ :\ \texttt{m}[j] \neq \texttt{k}) \wedge \texttt{m}[\texttt{i}] = \texttt{k} \wedge 0 \leq \texttt{i} < \texttt{n} - 1$$

pelo que o código de inicialização se pode escrever:

```
// $PC \equiv 2 \leq n \wedge n \leq \dim(m) \wedge 2 \leq (\mathbf{N}\, j : 0 \leq j < n : m[j] = k)$.
int i = ...;
inic
// $CO' \equiv (\mathbf{Q}\, j : 0 \leq j < i : m[j] \neq k) \wedge m[i] = k \wedge 0 \leq i < n - 1$.
++i;
// $(\mathbf{N}\, j : 0 \leq j < i : m[j] = k) = 1 \wedge m[i-1] = k \wedge 1 \leq i < n$.
```

onde CO' representa a condição objectivo do código de inicialização e não a condição objectivo do ciclo já desenvolvido.

A inicialização reduz-se portanto ao problema de encontrar o índice do primeiro elemento com valor `k`. Este índice é forçosamente inferior a $n - 1$, pois a matriz, pela pré-condição, possui dois elementos com valor `k`. A solução deste problema passa pela construção de um outro ciclo e é relativamente simples, pelo que se apresenta a solução sem mais comentários (dica: factorize-se a condição objectivo):

```
// $PC \equiv 2 \leq n \wedge n \leq \dim(m) \wedge 2 \leq (\mathbf{N}\, j : 0 \leq j < n : m[j] = k)$.
int i = 0;
// $CI' \equiv (\mathbf{Q}\, j : 0 \leq j < i : m[j] \neq k) \wedge 0 \leq i < n - 1$.
while(m[i] != k)
    ++i;
// $CO' \equiv (\mathbf{Q}\, j : 0 \leq j < i : m[j] \neq k) \wedge m[i] = k \wedge 0 \leq i < n - 1$.
++i;
// $(\mathbf{N}\, j : 0 \leq j < i : m[j] = k) = 1 \wedge m[i-1] = k \wedge 1 \leq i < n$.
// $CI \equiv (\mathbf{N}\, j : 0 \leq j < i : m[j] = k) = 1 \wedge 1 \leq i < n$.
```

A função completa é:

```
/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre $PC \equiv 2 \leq n \wedge n \leq \dim(m) \wedge 2 \leq (\mathbf{N}\, j : 0 \leq j < n : m[j] = k)$.
    @post $CO \equiv (\mathbf{N}\, j : 0 \leq j < \text{índiceDoSegundo} : m[j] = k) = 1 \wedge$
              $1 \leq \text{índiceDoSegundo} < n \wedge m[\text{índiceDoSegundo}] = k$. */
int índiceDoSegundo(int const m[], int const n, int const k)
{
    int i = 0;

    // $CI' \equiv (\mathbf{Q}\, j : 0 \leq j < i : m[j] \neq k) \wedge 0 \leq i < n - 1$.
    while(m[i] != k)
        ++i;
    // $CO' \equiv (\mathbf{Q}\, j : 0 \leq j < i : m[j] \neq k) \wedge m[i] = k \wedge 0 \leq i < n - 1$.

    ++i;

    // $CI \equiv (\mathbf{N}\, j : 0 \leq j < i : m[j] = k) = 1 \wedge 1 \leq i < n$.
    while(m[i] != k)
```

```
        ++i

    return i;
}
```

Pode-se obter código mais fácil de perceber se se começar por desenvolver uma função que devolva o primeiro elemento com valor `k` da matriz. Essa função usa um ciclo que é, na realidade o ciclo de inicialização usado acima:

```
/** Devolve o índice do primeiro elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre  PC ≡ 1 ≤ n ∧ n ≤ dim(m) ∧ 1 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (Q j : 0 ≤ j < índiceDoPrimeiro : m[j] ≠ k) ∧
               0 ≤ índiceDoPrimeiro < n ∧ m[índiceDoPrimeiro] = k. */
int índiceDoPrimeiro(int const m[], int const n, int const k)
{

    int i = 0;

    // CI ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i < n.
    while(m[i] != k)
        ++i;

    return i;
}

/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre  PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1∧
               1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k. */
int índiceDoSegundo(int const m[], int const n, int const k)
{

    int i = índiceDoPrimeiro(m, n, k) + 1;

    // CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
    while(m[i] != k)
        ++i

    return i;
}
```

Deixou-se propositadamente para o fim a escrita das instruções de asserção para verificação da pré-condição e da condição objectivo. No caso destas funções, quer a pré-condição quer a condição objectivo envolvem quantificadores. Será que, por isso, as instruções de asserção têm de ser mais fracas do que deveriam, verificando apenas parte do que deveriam? Na realidade não. É possível escrever-se uma função para contar o número de ocorrências de um valor nos primeiros elementos de uma matriz, e usá-la para substituir o quantificador de contagem:

```
/** Devolve o número de ocorrências do valor k nos primeiros n elementos
    da matriz m.
    @pre  PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ ocorrênciasDe = (N j : 0 ≤ j < n : m[j] = k). */
int ocorrênciasDe(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int ocorrências = 0;

    // CI ≡ ocorrênciasDe = (N j : 0 ≤ j < i : m[j] = k) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        if(m[i] == k)
            ++ocorrências;

    return ocorrências;
}

/** Devolve o índice do primeiro elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre  PC ≡ 1 ≤ n ∧ n ≤ dim(m) ∧ 1 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (Q j : 0 ≤ j < índiceDoPrimeiro : m[j] ≠ k) ∧
               0 ≤ índiceDoPrimeiro < n ∧ m[índiceDoPrimeiro] = k. */
int índiceDoPrimeiro(int const m[], int const n, int const k)
{
    assert(1 <= n and 1 <= ocorrênciasDe(m, n, k));

    int i = 0;

    // CI ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i < n.
    while(m[i] != k)
        ++i;

    assert(ocorrênciasDe(m, i, k) == 0 and
           0 <= i < n and m[i] = k);

    return i;
}

/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre  PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
```

@post $CO \equiv (\mathbf{N}\, j : 0 \leq j < \texttt{índiceDoSegundo} : \texttt{m}[j] = \texttt{k}) = 1 \wedge$
$1 \leq \texttt{índiceDoSegundo} < \texttt{n} \wedge \texttt{m}[\texttt{índiceDoSegundo}] = \texttt{k}$. */

```
int índiceDoSegundo(int const m[], int const n, int const k)
{
    assert(2 <= n and 2 <= ocorrênciasDe(m, n, k));

    int i = índiceDoPrimeiro(m, n, k) + 1;
```

// $CI \equiv (\mathbf{N}\, j : 0 \leq j < \texttt{i} : \texttt{m}[j] = \texttt{k}) = 1 \wedge 1 \leq \texttt{i} < \texttt{n}$.

```
    while(m[i] != k)
        ++i

    assert(ocorrênciasDe(m, i, k) == 1 and
           1 <= i < n and m[i] = k);

    return i;
}
```

Ao se especificar as funções acima, poder-se-ia ter decidido que, caso o número de ocorrências do valor `k` na matriz fosse inferior ao desejado, estas deveriam devolver o valor `n`, pois é sempre um índice inválido (os índices dos primeiros `n` elementos da matriz variam entre 0 e $\texttt{n} - 1$) sendo por isso um valor apropriado para indicar uma condição de erro. Nesse caso as funções poderiam ser escritas como[22]:

/** Devolve o índice do primeiro elemento com valor `k` nos primeiros `n` elementos da matriz `m` ou `n` se não existir.
@pre $PC \equiv 0 \leq \texttt{n} \wedge \texttt{n} \leq \dim(\texttt{m})$.
@post $CO \equiv ((\mathbf{Q}\, j : 0 \leq j < \texttt{índiceDoPrimeiro} : \texttt{m}[j] \neq \texttt{k}) \wedge$
$0 \leq \texttt{índiceDoPrimeiro} < \texttt{n} \wedge \texttt{m}[\texttt{índiceDoPrimeiro}] = \texttt{k}) \vee$
$((\mathbf{Q}\, j : 0 \leq j < \texttt{n} : \texttt{m}[j] \neq \texttt{k}) \wedge \texttt{índiceDoPrimeiro} = \texttt{n})$. */

```
int índiceDoPrimeiro(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int i = 0;
```

[22] As condições objectivo das duas funções não são, em rigor, correctas. O problema é que em

$$0 \leq \texttt{índiceDoPrimeiro} < \texttt{n} \wedge \texttt{m}[\texttt{índiceDoPrimeiro}] = \texttt{k}$$

o segundo termo tem valor indefinido para $\texttt{índiceDoPrimeiro} = \texttt{n}$. Na realidade dever-se-ia usar uma conjunção especial, que tivesse valor falso desde que o primeiro termo tivesse valor falso *independentemente do segundo termo estar ou não definido*. Pode-se usar um símbolo especial para uma conjunção com estas características, por exemplo $\curlywedge$. De igual forma pode-se definir uma disjunção especial com valor verdadeiro se o primeiro termo for verdadeiro independentemente de o segundo termo estar ou não definido, por exemplo $\curlyvee$. Em [8] usam-se os nomes **cand** e **cor** com o mesmo objectivo. Em [11] chama-se-lhes "and if" e "or else". Estes operadores binários *não são comutativos*, ao contrário do que acontece com a disjunção e a conjunção usuais. Na linguagem C++, curiosamente, só existem as versões não-comutativas destes operadores.

```
    // CI ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i ≤ n.
    while(i != n and m[i] != k)
        ++i;

    assert((ocorrênciasDe(m, i, k) == 0 and
            0 <= i < n and m[i] = k) or
           (ocorrênciasDe(m, n, k) == 0 and i == n));

    return i;
}

/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m ou n se não existir.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ ((N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1∧
             1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k) ∨
             ((N j : 0 ≤ j < n : m[j] = k) < 2 ∧ índiceDoSegundo = n). */
int índiceDoSegundo(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int i = índiceDoPrimeiro(m, n, k);

    if(i == n)
        return n;

    ++i;

    // CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i ≤ n.
    while(i != n and m[i] != k)
        ++i

    assert((ocorrênciasDe(m, i, k) == 1 and
            1 <= i < n and m[i] = k) or
           (ocorrênciasDe(m, n, k) < 2 and i == n));

    return i;
}
```

5.3.8 Segundo item de um vector com um dado valor

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. As funções resultantes desse desenvolvimento são

```
/** Devolve o número de ocorrências do valor k nos primeiros n elementos
```

```
    do vector v.
    @pre PC ≡ 0 ≤ n ∧ n ≤ v.size().
    @post CO ≡ ocorrênciasDe = (N j : 0 ≤ j < v.size() : v[j] = k). */
vector<int>::size_type ocorrênciasDe(vector<int> const& v,
                                     int const n, int const k)
{
    assert(0 <= n and n <= v.size());

    vector<int>::size_type ocorrências = 0;

    // CI ≡ ocorrênciasDe = (N j : 0 ≤ j < i : v[j] = k) ∧ 0 ≤ i ≤ v.size().
    for(vector<int>::size_type i = 0; i != v.size(); ++i)
        if(v[i] == k)
            ++ocorrências;

    return ocorrências;
}

/** Devolve o índice do primeiro item com valor k do vector v ou
    v.size() se não existir.
    @pre PC ≡ 𝒱.
    @post CO ≡ ((Q j : 0 ≤ j < índiceDoPrimeiro : v[j] ≠ k) ∧
               0 ≤ índiceDoPrimeiro < v.size() ∧ v[índiceDoPrimeiro] = k) ∨
               ((Q j : 0 ≤ j < v.size() : v[j] ≠ k) ∧ índiceDoPrimeiro = v.size()). */
vector<int>::size_type índiceDoPrimeiro(vector<int> const& v,
                                        int const k)
{
    vector<int>::size_type i = 0;

    // CI ≡ (Q j : 0 ≤ j < i : v[j] ≠ k) ∧ 0 ≤ i ≤ v.size().
    while(i != v.size() and v[i] != k)
        ++i;

    assert((ocorrênciasDe(v, i, k) == 0 and
            0 <= i < v.size() and v[i] = k) or
           (ocorrênciasDe(v, v.size(), k) == 0 and
            i == v.size()));

    return i;
}

/** Devolve o índice do segundo elemento com valor k do vector v ou
    v.size() se não existir.
    @pre PC ≡ 𝒱.
    @post CO ≡ ((N j : 0 ≤ j < índiceDoSegundo : v[j] = k) = 1∧
               1 ≤ índiceDoSegundo < v.size() ∧ v[índiceDoSegundo] = k) ∨
```

```
              ((N j : 0 ≤ j < v.size() : m[j] = k) < 2 ∧ índiceDoSegundo = v.size()). */
vector<int>::size_type índiceDoSegundo(vector<int> const& v,
                                       int const k)
{
    int i = índiceDoPrimeiro(m, k);

    if(i == v.size()
        return v.size();

    ++i;

    // CI ≡ (N j : 0 ≤ j < i : v[j] = k) = 1 ∧ 1 ≤ i ≤ v.size().
    while(i != v.size() and v[i] != k)
        ++i

    assert((ocorrênciasDe(v, i, k) == 1 and
            1 <= i < v.size() and v[i] = k) or
           (ocorrênciasDe(v, v.size(), k) < 2 and
            i == v.size()));

    return i;
}
```

5.4 Cadeias de caracteres

A maior parte da comunicação entre humanos faz-se usando a palavra, falada ou escrita. É natural, portanto, que o processamento de palavras escritas seja também parte importante da maior parte dos programas: a comunicação entre computador e humano suporta-se ainda fortemente na palavra escrita. Dos tipos básicos do C++ faz parte o tipo `char`, que é a base para todo este processamento. Falta, no entanto, forma de representar *cadeias ou sequências de caracteres*. O C++ herdou da linguagem C uma forma de representação de cadeias de caracteres peculiar. São as chamadas cadeias de caracteres clássicas, representadas por matrizes de caracteres. As cadeias de caracteres *clássicas* são matrizes de caracteres em que o fim da cadeia é assinalado usando um caractere especial: o chamado caractere nulo, de código 0, que não representa nenhum símbolo em nenhuma das possíveis tabelas de codificação de caracteres (ver exemplo no Apêndice G).

As cadeias de caracteres clássicas são talvez eficientes, mas são também seguramente inflexíveis e desagradáveis de utilizar, uma vez que sofrem de todos os inconvenientes das matrizes clássicas. É claro que uma possibilidade de representação alternativa para as cadeias de caracteres seria recorrendo ao tipo genérico `vector`: a classe `vector<char>` permite de facto representar sequências de caracteres arbitrárias. No entanto, a biblioteca padrão do C++ fornece uma classe, de nome `string`, que tem todas as capacidades de `vector<char>` mas acrescenta uma quantidade considerável de operações especializadas para lidar com cadeias de caracteres.

Como se fez mais atrás neste capítulo acerca das matrizes e dos vectores, começar-se-á por apresentar brevemente a representação mais primitiva de cadeias de caracteres, passando-se depois a uma descrição mais ou menos exaustiva da classe `string`.

5.4.1 Cadeias de caracteres clássicas

A representação mais primitiva de cadeias de caracteres em C++ usa matrizes de caracteres em que o final da cadeia é marcado através de um caractere especial, de código zero, e que pode ser explicitado através da sequência de escape `\0`. Por exemplo,

```
char nome[] = {'Z', 'a', 'c', 'a', 'r', 'i', 'a', 's', '\0'};
```

define uma cadeia de caracteres representando o nome "Zacarias". É importante perceber que uma matriz de caracteres só é uma cadeia de caracteres clássica se possuir o terminador `'\0'`. Assim,

```
char nome_matriz[] = {'Z', 'a', 'c', 'a', 'r', 'i', 'a', 's'};
```

é uma matriz de caracteres mas *não é* uma cadeia de caracteres clássica.

Para simplificar a inicialização de cadeias de caracteres, a linguagem permite colocar a sequência de caracteres do inicializador entre aspas e omitir o terminador, que é colocado automaticamente. Por exemplo:

```
char dia[] = "Sábado";
```

define uma cadeia com seis caracteres contendo "Sábado" e representada por uma matriz de dimensão sete: os seis caracteres da palavra "Sábado" e o caractere terminador.

Uma cadeia de caracteres não necessita de ocupar toda a matriz que lhe serve de suporte. Por exemplo,

```
char const janeiro[12] = "Janeiro";
```

define uma matriz de 12 caracteres constantes contendo uma cadeia de caracteres de comprimento sete, que ocupa exactamente oito elementos da matriz: os primeiros sete com os caracteres da cadeia e o oitavo com o terminador.

As cadeias de caracteres podem ser inseridas em canais, o que provoca a inserção de cada um dos seus caracteres (com excepção do terminador). Assim, dadas as definições acima, o código

```
cout << "O nome é " << nome << '.' << endl;
cout << "Hoje é " << dia << '.' << endl;
```

faz surgir no ecrã

```
O nome é Zacarias.
Hoje é Sábado.
```

Se se olhar atentamente o código acima, verificar-se-á que existe uma forma alternativa escrever cadeias de caracteres num programa: as chamadas cadeias de caracteres literais, que correspondem a uma sequência de caracteres envolvida em aspas, por exemplo `"O nome é"`. As cadeias de caracteres literais são uma peculiaridade da linguagem. Uma cadeia de caracteres literal:

1. É uma matriz de caracteres, como qualquer cadeia de caracteres clássica.
2. Os seus caracteres são constantes. O tipo de uma cadeia de caracteres literal é, por isso, `char const [`*`dimensão`*`]`, onde *`dimensão`* é o número de caracteres somado de um (para haver espaço para o terminador).
3. Não tem nome, ao contrário das variáveis usuais.
4. O seu âmbito está limitado à expressão em que é usada.
5. Tem permanência estática, existindo durante todo o programa, como se fosse global.

Para o demonstrar comece-se por um exemplo simples. É possível percorrer uma cadeia de caracteres usando um ciclo. Por exemplo:

```
char nome[] = "Zacarias";

for(int i = 0; nome[i] != '\0'; ++i)
    cout << nome[i];
cout << endl;
```

Este troço de código tem exactamente o mesmo resultado que

```
char nome[] = "Zacarias";

cout << nome << endl;
```

A sua particularidade é a guarda usada para o ciclo. É que, como a dimensão da cadeia não é conhecida à partida, uma vez que pode ocupar apenas uma parte da matriz, a forma mais segura de a percorrer é usar o terminador para verificar quando termina.

As mesmas ideias podem ser usadas para percorrer uma cadeia de caracteres literal, o que demonstra claramente que estas são também matrizes:

```
int comprimento = 0;
while("Isto é um teste."[comprimento] != '\0')
    ++comprimento;

cout << "O comprimento é " << comprimento << '.' << endl;
```

Este troço de programa escreve no ecrã o comprimento da cadeia "`Isto é um teste.`", i.e., escreve 16.

Uma cadeia de caracteres literal não pode ser dividida em várias linhas. Por exemplo, o código seguinte é ilegal:

```
cout << "Isto é uma frase comprida que levou à necessidade de a
dividir em três linhas.  Só que, infelizmente, de uma
forma ilegal." << endl;
```

No entanto, como cadeias de caracteres adjacentes no código são consideradas como uma e uma só cadeia de caracteres literal, pode-se resolver o problema acima de uma forma perfeitamente legal e legível:

```
cout << "Isto é uma frase comprida que levou à necessidade de a "
        "dividir em três linhas.  Desta vez de uma "
        "forma legal." << endl;
```

5.4.2 A classe `string`

As cadeias de caracteres clássicas devem ser utilizadas apenas onde indispensável: para especificar cadeias de caracteres literais, e.g., para compor frases a inserir no canal de saída `cout`, como tem vindo a ser feito até aqui. Para representar cadeias de caracteres deve-se usar a classe `string`, definida no ficheiro de interface com o mesmo nome. Ou seja, para usar esta classe deve-se colocar no topo dos programas a directiva

```
#include <string>
```

De este ponto em diante usar-se-ão as expressões "cadeia de caracteres" e "cadeia" para classificar qualquer variável ou constante do tipo `string`.

Como todas as descrições de ferramentas da biblioteca padrão feitas neste texto, a descrição da classe `string` que se segue não pretende de modo algum ser exaustiva. Para tirar partido de todas as possibilidades das ferramentas da biblioteca padrão é indispensável recorrer, por exemplo, a [12].

As cadeias de caracteres suportam todas as operações dos vectores descritas na Secção 5.2, com excepção das operações `push_back()`, `pop_back()`, `front()` e `back()`, pelo que serão apresentadas apenas as operações específicas das cadeias de caracteres.

Definição (construção) e inicialização

Para definir e inicializar uma cadeia de caracteres pode-se usar qualquer das formas seguintes:

```
string vazia;                                 // cadeia vazia, sem caracteres.
string nome = "Zacarias Zebedeu Zagalo";      // a partir de cadeia clássica
                                              // (literal, neste caso).
string mesmo_nome = nome;                     // cópia a partir de cadeia.
string apelidos(nome, 9);                     // cópia a partir da posição 9.
string nome_do_meio(nome, 9, 7);              // cópia de 7 caracteres a
                                              // partir da posição 9.
string vinte_aa(20, 'a');                     // dimensão inicial 20,
                                              // tudo com 'a'.
```

Atribuição

Podem-se fazer atribuições entre cadeias de caracteres. Também se pode atribuir uma cadeia de caracteres clássica ou mesmo um simples caractere a uma cadeia de caracteres:

```
string nome1 = "Xisto Ximenes";
string nome2;
string nome3;

nome2 = nome1;        // atribuição entre cadeias.
nome1 = "Ana Anes";   // atribuição de cadeia clássica.
nome3 = 'X';          // atribuição de um só caractere (literal, neste caso).
```

Podem-se fazer atribuições mais complexas usando a operação `assign()`:

```
string nome = "Zacarias Zebedeu Zagalo";
string apelidos;
string nome_do_meio;
string vinte_aa;

apelidos.assign(nome, 9, string::npos); // só final de cadeia.
nome_do_meio.assign(nome, 9, 7);        // só parte de cadeia.
vinte_aa.assign(20, 'a');               // caractere 'a' repetido 20
                                        // vezes.
```

A constante `string::npos` é maior do que o comprimento de qualquer cadeia possível. Quando usada num local onde se deveria indicar um número de caracteres significa "todos os restantes". Daí que a variável apelidos passe a conter "Zebedeu Zagalo".

Dimensão e capacidade

As cadeias de caracteres suportam todas as operações relativas a dimensões e capacidade dos vectores, adicionadas das operações `length()` e `erase()`:

`size()` Devolve a dimensão actual da cadeia.

`length()` Sinónimo de `size()`, porque é mais usual falar-se em comprimento que em dimensão de uma cadeia.

`max_size()` Devolve a maior dimensão possível de uma cadeia.

`resize(n, v)` Altera a dimensão da cadeia para *n*. Tal como no caso dos vectores, o segundo argumento, o valor dos possíveis novos caracteres, é opcional (se não for especificado os novos caracteres serão o caractere nulo).

`reserve(n)` Reserva espaço para *n* caracteres na cadeia, de modo a evitar a ineficiência associada a aumentos sucessivos.

`capacity()` Devolve a capacidade actual da cadeia, que é a dimensão até à qual a cadeia pode crescer sem ter de requerer memória ao sistema operativo.

`clear()` Esvazia a cadeia.

`erase()` Sinónimo de `clear()`, porque é mais usual dizer-se apagar do que limpar uma cadeia.

`empty()` Indica se a cadeia está vazia.

Indexação

Os modos de indexação são equivalentes aos dos vectores. A indexação usando o operador de indexação `[]` é insegura, no sentido em que a validade dos índices não é verificada. Para realizar uma indexação segura utilizar a operação `at()`.

Acrescento

Existem várias formas elementares de acrescentar cadeias de caracteres:

```
string nome = "Zacarias";
string nome_do_meio = "Zebedeu";

nome += ' ';           // um só caractere.
nome += nome_do_meio;  // uma cadeia.
nome += " Zagalo";     // uma cadeia clássica (literal, neste caso).
```

Para acrescentos mais complexos existe a operação `append()`:

```
string vinte_aa(10, 'a'); // só 10...
string contagem = "um dois ";
string dito = "não há duas sem três";

vinte_aa.append(10, 'a');        // caracteres repetidos (10 novos 'a').
contagem.append(dito, 17, 4);    // quatro caracteres da posição 17 de dito.
```

Inserção

De igual forma existem várias versões da operação `insert()` para inserir material em lugares arbitrários de uma cadeia:

```
string contagem = "E vão  ,  e !";
string dito = "não há uma sem duas";
string duas = "duas";

contagem.insert(12, "três");     // cadeia clássica na posição 12.
contagem.insert(9, duas);        // cadeia na posição 9.
contagem.insert(7, dito, 7, 3); // parte da cadeia dito na posição 7.
contagem.insert(6, 3, '.');      // caracteres repetidos na posição 6.
// contagem fica com "E vão ... uma, duas e três!".
```

O material é inserido *antes* da posição indicada.

Substituição

É possível, recorrendo às várias operações `replace()`, substituir pedaços de uma cadeia por outras sequências de caractere. Os argumentos são os mesmos que para as operações `insert()`, mas, para além de se indicar onde começa a zona a substituir, indica-se também quantos caracteres substituir. Por exemplo:

```
string texto = "O nome é $nome.";

texto.replace(9, 5, "Zacarias"); // substitui "$nome" por "Zacarias".
```

Procuras

Há muitas formas de procurar texto dentro de uma cadeia de caracteres. A primeira forma procura uma sequência de caracteres do início para o fim (operação `find()`) ou do fim para o início (operação `rfind()`) da cadeia. Estas operações têm como argumentos, em primeiro lugar, a sequência a procurar e opcionalmente, em segundo lugar, a posição a partir da qual iniciar a procura (se não se especificar este argumento, a procura começa no início ou no fim da cadeia, consoante a direcção de procura). A sequência a procurar pode ser uma cadeia de caracteres, uma cadeia de caracteres clássica ou um simples caractere. O valor devolvido é a posição onde a sequência foi encontrada ou, caso não o tenha sido, o valor `string::npos`. Por exemplo:

```
string texto = "O nome é $nome.";

// Podia também ser texto.find('$'):
string::size_type posição = texto.find("$nome");
```

```
if(posição != string::npos)
    // Substitui "$nome" por "Zacarias":
    texto.replace(posição, 5, "Zacarias");
```

A segunda forma serve para procurar caracteres de uma dado conjunto dentro de uma cadeia de caracteres. A procura é feita do início para o fim (operação `find_first_of()`) ou do fim para o início (operação `find_last_of()`) da cadeia. O conjunto de caracteres pode ser dado na forma de uma cadeia de caracteres, de uma cadeia de caracteres clássica ou de um simples caractere. Os argumentos são mais uma vez o conjunto de caracteres a procurar e a posição inicial de procura, que é opcional. O valor devolvido é mais uma vez a posição encontrada ou `string::npos` se nada foi encontrado. Por exemplo:

```
string texto = "O nome é $nome.";

// Podia também ser texto.find_first_of('$'):
string::size_type posição = texto.find_first_of("$%#", 3);
// Começa-se em 3 só para clarificar onde se especifica a posição inicial de procura.

if(posição != string::npos)
    // Substitui "$nome" por "Zacarias":
    texto.replace(posição, 5, "Zacarias");
```

Existem ainda as operações `find_first_not_of()` e `find_last_not_of()` se se quiser procurar caracteres que *não pertençam* a um dado conjunto de caracteres.

Concatenação

É possível concatenar cadeias usando o operador +. Por exemplo:

```
string um = "um";
string contagem = um + " dois" + ' ' + 't' + "rês";
```

Comparação

A comparação entre cadeias é feita usando os operadores de igualdade e relacionais. A ordem considerada para as cadeias é lexicográfica (ver Secção 5.2.12) e depende da ordenação dos caracteres na tabela de codificação usada, que por sua vez é baseada no respectivo código. Assim, consultando o Apêndice G, que contém a tabela de codificação Latin-1, usada por enquanto em Portugal, conclui-se que

$$
\begin{aligned}
\texttt{'A'} &< \texttt{'a'} \\
\texttt{'a'} &< \texttt{'á'} \\
\texttt{'Z'} &< \texttt{'a'} \\
\texttt{'z'} &< \texttt{'a'}
\end{aligned}
$$

onde infelizmente a ordenação relativa de maiúsculas e minúsculas e, sobretudo, dos caracteres acentuados, não respeita as regras habituais do português. Aqui entra-se pela terceira vez no problema das variações regionais de critérios[23]. Estes problemas resolvem-se recorrendo ao conceito de `locales`, que saem fora do âmbito deste texto. Assim, representando as cadeias por sequências de caracteres entre aspas:

- "leva" < "levada"
- "levada" < "levadiça"
- "órfão" > "orfeão" (infelizmente...)
- "fama" < "fome"
- etc.

Formas mais complexas de comparação recorrem à operação `compare()`, que devolve um valor positivo se a cadeia for maior que a passada como argumento, zero se for igual, e um valor negativo se for menor. Por exemplo:

```
string cadeia1 = "fama";
string cadeia2 = "fome";

int resultado = cadeia1.compare(cadeia2);

if(resultado == 0)
    cout << "São iguais!" << endl;
else if(resultado < 0)
    cout << "A primeira é a menor!" << endl;
else
    cout << "A segunda é a menor!" << endl;
```

Este código, para as cadeias indicadas, escreve:

```
A primeira é a menor!
```

[23]As duas primeiras não foram referidas explicitamente e dizem respeito ao formato dos valores decimais e de valores booleanos em operações de inserção e extracção. Seria desejável, por exemplo, que os números decimais aparecessem ou fossem lidos com vírgula em vez de ponto, e os valores booleanos por extenso na forma `verdadeiro` e `falso`, em vez de `true` e `false`.