Capítulo 4

Controlo do fluxo dos programas

Se temos...! Diz ela
mas o problema não é só de aprender
é saber a partir daí que fazer

Sérgio Godinho, 2º andar direito, *Pano Cru*.

Quase todas as resoluções de problemas envolvem tomadas de decisões e repetições. Dificilmente se consegue encontrar um algoritmo interessante que não envolva, quando lido em português, as palavras "se" e "enquanto", correspondendo aos conceitos de selecção e de iteração. Neste capítulo estudar-se-ão em pormenor os mecanismos da programação imperativa que suportam esses conceitos e discutir-se-ão metodologias de resolução de problemas usando esses mecanismos.

4.1 Instruções de selecção

A resolução de um problema implica quase sempre a tomada de decisões ou pelo menos a selecção de alternativas.

Suponha-se que se pretendia desenvolver uma função para calcular o valor absoluto de um inteiro. O "esqueleto" da função pode ser o que se segue

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0 ≤ absoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    ...
}
```

Este esqueleto inclui, como habitualmente, documentação clarificando o que a função faz, o cabeçalho que indica como a função se utiliza, e um corpo, onde se colocarão as instruções que resolvem o problema, i.e., que explicitam como a função funciona.

A resolução deste problema requer que sejam tomadas acções diferentes consoante o valor de `x` seja positivo ou negativo (ou nulo). São portanto fundamentais as chamadas instruções de selecção ou alternativa.

4.1.1 As instruções `if` e `if else`

As instruções `if` e `if else` são das mais importantes instruções de controlo do fluxo de um programa, i.e., instruções que alteram a sequência normal de execução das instruções de um programa. Estas instruções permitem executar uma instrução caso uma condição seja verdadeira e, no caso da instrução `if else`, uma outra instrução caso a condição seja falsa. Por exemplo, no troço de programa

```
if(x < 0)  // 1
    x = 0; // 2
else       // 3
    x = 1; // 4
...        // 5
```

as linhas 1 a 4 correspondem a uma única *instrução de selecção*. Esta instrução de selecção é composta de uma condição (na linha 1) e duas instruções alternativas (linhas 2 e 4). Se `x` for menor do que zero quando a instrução `if` é executada, então a próxima instrução a executar é a instrução na linha 2, passando o valor de `x` a ser zero. No caso contrário, se `x` for inicialmente maior ou igual a zero, a próxima instrução a executar é a instrução na linha 4, passando a variável `x` a ter o valor 1. Em qualquer dos casos, a execução continua na linha 5[1].

Como a instrução na linha 2 só é executada se $\mathtt{x} < 0$, diz-se que $\mathtt{x} < 0$ é a sua guarda, normalmente representada por G. De igual modo, a guarda da instrução na linha 4 é $0 \leq \mathtt{x}$. A guarda da instrução alternativa após o `else` está implícita, podendo ser obtida por negação da guarda do `if`: $\neg(\mathtt{x} < 0)$ é equivalente a $0 \leq \mathtt{x}$. Assim, numa instrução de selecção pode-se sempre inverter a ordem das instruções alternativas desde que se negue a condição. Assim,

```
if(x < 0)
    // G₁ ≡ x < 0
    x = 0;
else
    // G₂ ≡ 0 ≤ x
    x = 1;
```

é equivalente a

[1]Se existirem instruções `return`, `break`, `continue` ou `goto` nas instruções controladas por um `if`, o fluxo normal de execução pode ser alterado de tal modo que a execução não continua na instrução imediatamente após esse `if`.

```
if(0 <= x)
    // G₂ ≡ 0 ≤ x
    x = 1;
else
    // G₁ ≡ x < 0
    x = 0;
```

O fluxo de execução de uma instrução de selecção genérica

```
if(C)
    // G₁ ≡ C
    instrução₁
else
    // G₂ ≡ ¬C
    instrução₂
```

é representado no diagrama de actividade da Figura 4.1.

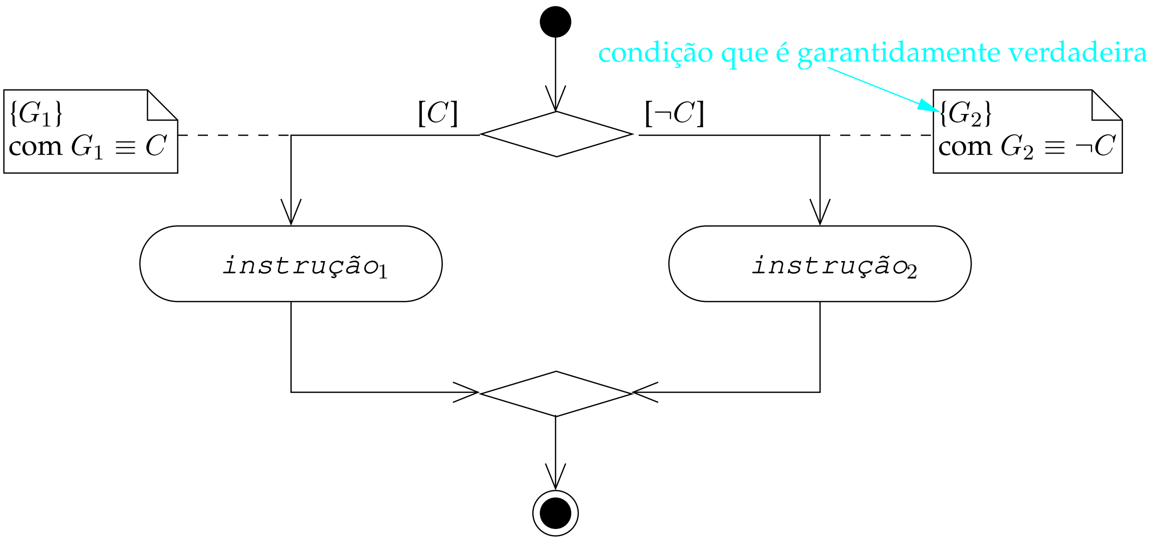

Figura 4.1: Diagrama de actividade de uma instrução de selecção genérica. Uma condição que tem de ser sempre verdadeira coloca-se num comentário entre chavetas.

Em certos casos pretende-se apenas executar uma dada instrução se determinada condição se verificar, não se desejando fazer nada caso a condição seja falsa. Nestes casos pode-se omitir o `else` e a instrução que se lhe segue. Por exemplo, no troço de programa

```
if(x < 0)   // 1
    x = 0;  // 2
...         // 3
```

se x for inicialmente menor do que zero, então é executada a instrução condicionada na linha 2, passando o valor de x a ser zero, sendo em seguida executada a instrução na linha 3. No caso contrário a execução passa directamente para a linha 3.

A este tipo restrito de instrução de selecção também se chama *instrução condicional*. Uma instrução condicional é sempre equivalente a uma instrução de selecção em que a instrução após o `else` é uma instrução nula (a instrução nula corresponde em C++ a um simples terminador `;`). Ou seja, o exemplo anterior é equivalente a:

```
if(x < 0)  // 1
    x = 0; // 2a
else       // 2b
    ;      // 2c
...        // 3
```

O fluxo de execução de uma instrução de condicional genérica

```
if(C)
    instrução
```

é representado no diagrama de actividade da Figura 4.2.

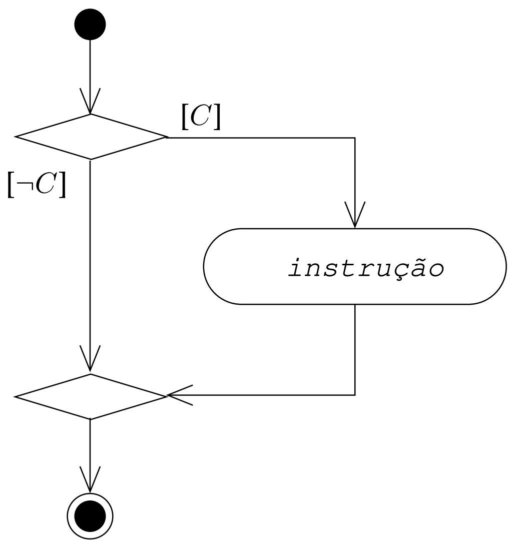

Figura 4.2: Diagrama de actividade de uma instrução condicional genérica.

Em muitos casos é necessário executar condicional ou alternativamente não uma instrução simples mas sim uma sequência de instruções. Para o conseguir, agrupam-se essas instruções num bloco de instruções ou instrução composta, colocando-as entre chavetas `{}`. Por exemplo, o código que se segue ordena os valores guardados nas variáveis x e y de tal modo que x termine com um valor menor ou igual ao de y:

```
int x, y;
...
if(y < x) {
    int const auxiliar = x;
    x = y;
    y = auxiliar;
}
```

4.1.2 Instruções de selecção encadeadas

Caso seja necessário, podem-se encadear instruções de selecção umas nas outras. Isso acontece quando se pretende seleccionar uma entre mais do que duas instruções alternativas. Por exemplo, para verificar qual a posição do valor de uma variável `a` relativamente a um intervalo [`mínimo` `máximo`], pode-se usar

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    cout << a << " menor que mínimo." << endl;
else {
    if(a <= máximo)
        cout << a << " entre mínimo e máximo." << endl;
    else
        cout << a << " maior que máximo." << endl;
}
```

Sendo as instruções de selecção instruções por si só, o código acima pode-se escrever sem recurso às chavetas, ou seja,

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    cout << a << " menor que mínimo." << endl;
else
    if(a <= máximo)
        cout << a << " entre mínimo e máximo." << endl;
    else
        cout << a << " maior que máximo." << endl;
```

Em casos como este, em que se encadeiam `if else` sucessivos, é comum usar uma indentação que deixa mais claro que existem mais do que duas alternativas,

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    cout << a << " menor que mínimo." << endl;
else if(a <= máximo)
    cout << a << " entre mínimo e máximo." << endl;
else
    cout << a << " maior que máximo." << endl;
```

Guardas em instruções alternativas encadeadas

Podem-se colocar como comentários no exemplo anterior as guardas de cada instrução alternativa. Estas guardas reflectem as circunstâncias nas quais as instruções alternativas respectivas são executadas

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    // G₁ ≡ a < mínimo.
    cout << a << " menor que mínimo." << endl;
else if(a <= máximo)
    // G₂ ≡ mínimo ≤ a ≤ máximo.
    cout << a << " entre mínimo e máximo." << endl;
else
    // G₃ ≡ mínimo ≤ a ∧ máximo < a.
    cout << a << " maior que máximo." << endl;
```

Fica claro que as guardas não correspondem directamente à condição indicada no `if` respectivo, com excepção da primeira.

As n guardas de uma instrução de selecção, construída com $n-1$ instruções `if` encadeadas, podem ser obtidas a partir das condições de cada um dos `if` como se segue:

```
if(C₁)
    // G₁ ≡ C₁.
    ...
else if(C₂)
    // G₂ ≡ ¬G₁ ∧ C₂  (ou, ¬C₁ ∧ C₂).
    ...
else if(C₃)
    // G₃ ≡ ¬G₁ ∧ ¬G₂ ∧ C₃  (ou, ¬C₁ ∧ ¬C₂ ∧ C₃).
    ...
...
```

```
else if(C_{n-1})
    // G_{n-1} ≡ ¬G_1 ∧ ··· ∧ ¬G_{n-2} ∧ C_{n-1} (ou, ¬C_1 ∧ ··· ∧ ¬C_{n-2} ∧ C_{n-1}).
    ...
else
    // G_n ≡ ¬G_1 ∧ ··· ∧ ¬G_{n-1} (ou, ¬C_1 ∧ ··· ∧ ¬C_{n-1}).
    ...
```

Ou seja, as guardas das instruções alternativas são obtidas por conjunção da condição do `if` respectivo com a negação das guardas das instruções alternativas (ou das condições de todos os `if`) anteriores na sequência, como seria de esperar. Uma instrução de selecção encadeada é pois equivalente a uma instrução de selecção com mais do que duas instruções alternativas, como se mostra na Figura 4.3.

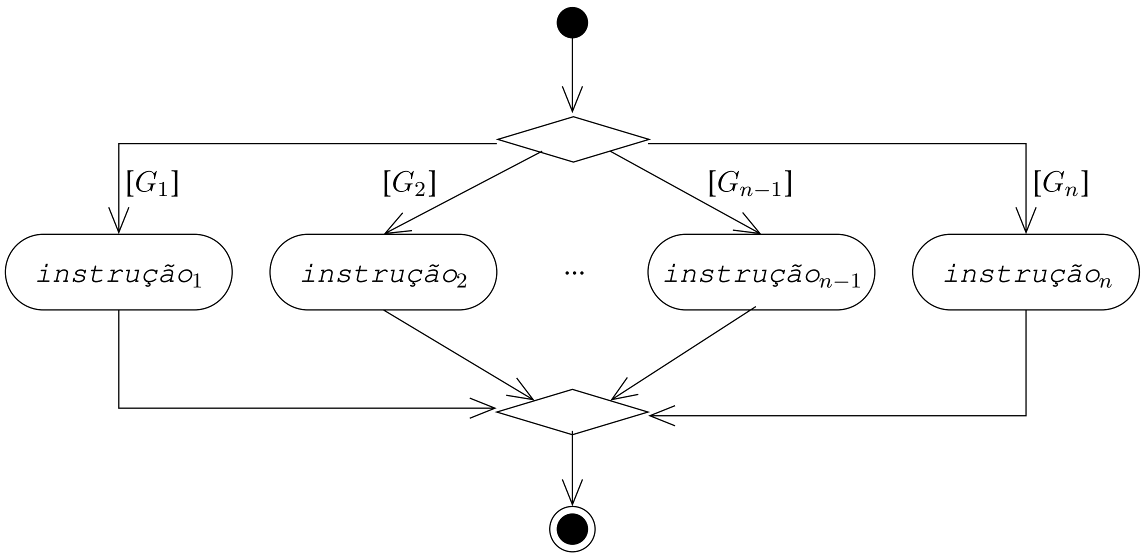

Figura 4.3: Diagrama de actividade de uma instrução de múltipla (sem correspondente directo na linguagem C++) equivalente a uma instrução de selecção encadeada. As guardas presumem-se mutuamente exclusivas.

Influência de pré-condições

A última guarda do exemplo dado anteriormente parece ser redundante: se `a` é maior que `máximo` não é forçosamente maior que `mínimo`? Então porque não é a guarda simplesmente `máximo` $<$ `a`? Acontece que nada garante que `mínimo` $\leq$ `máximo`! Se se introduzir essa condição como pré-condição das instruções de selecção encadeadas, então essa condição verifica-se imediatamente antes de cada uma das instruções alternativas, pelo que as guardas podem ser simplificadas e tomar a sua forma mais intuitiva

```
// PC ≡ mínimo ≤ máximo
if(a < mínimo)
```

```
    // G₁ ≡ a < mínimo.
    cout << a << " menor que mínimo." << endl;
else if(a <= máximo)
    // G₂ ≡ mínimo ≤ a ≤ máximo.
    cout << a << " entre mínimo e máximo." << endl;
else
    // G₃ ≡ máximo < a.
    cout << a << " maior que máximo." << endl;
```

4.1.3 Problemas comuns

A sintaxe das instruções `if` e `if else` do C++ presta-se a ambiguidades. No código

```
if(m == 0)
    if(n == 0)
        cout << "m e n são zero." << endl;
else
    cout << "m não é zero." << endl;
```

o `else` não diz respeito ao primeiro `if`. Ao contrário do que a indentação do código sugere, o `else` diz respeito ao segundo `if`. Em caso de dúvida, um `else` pertence ao `if` mais próximo (e acima...) dentro mesmo bloco de instruções e que não tenha já o respectivo `else`. Para corrigir o exemplo anterior é necessário construir uma instrução composta, que neste caso consiste de uma única instrução de selecção

```
if(m == 0) {
    if(n == 0)
        cout << "m e n são zero." << endl;
} else
    cout << "m não é zero." << endl;
```

É conveniente usar blocos de instruções de uma forma liberal, pois construções como a que se apresentou podem dar origem a erros muito difíceis de detectar e corrigir. Os compiladores de boa qualidade, no entanto, avisam o programador da presença de semelhantes (aparentes) ambiguidades.

Um outro erro frequente corresponde a colocar um terminador `;` logo após a condição do `if` ou logo após o `else`. Por exemplo, a intenção do programador do troço de código

```
if(x < 0);
    x = 0;
```

era provavelmente que se mantivesse o valor de `x` excepto quando este fosse negativo. Mas a interpretação feita pelo compilador (e a correcta dada a sintaxe da linguagem C++) é

```
if(x < 0)
    ; // instrução nula: não faz nada.
x = 0;
```

ou seja, x terminará sempre com o valor zero! Este tipo de erro, não sendo muito comum, é ainda mais difícil de detectar do que o da (suposta) ambiguidade da pertença de um `else`: os olhos do programador, habituados que estão à presença de `;` no final de cada linha, recusam-se a detectar o erro.

4.2 Asserções

Antes de se passar ao desenvolvimento de instruções de selecção, é importante fazer uma pequena digressão para introduzir um pouco mais formalmente o conceito de asserção.

Chama-se asserção a um predicado (ver Secção A.1) escrito normalmente na forma de comentário antes ou depois de uma instrução de um programa. As asserções correspondem a afirmações acerca das variáveis do programa que se sabe serem verdadeiras antes da instrução seguinte à asserção e depois da instrução anterior à asserção. Uma asserção pode sempre ser vista como pré-condição PC da instrução seguinte e condição objectivo CO da instrução anterior. As asserções podem também incluir afirmações acerca de variáveis matemáticas, que não pertencem ao programa.

Nas asserções cada variável pertence a um determinado conjunto. Para as variáveis C++, esse conjunto é determinado pelo tipo da variável indicado na sua definição (e.g., `int x;` significa que `x` pertence ao conjunto dos inteiros entre -2^{n-1} e $2^{n-1} - 1$, se os `int` tiverem n *bits*). Para as variáveis matemáticas, esse conjunto deveria, em rigor, ser indicado explicitamente. Neste texto, no entanto, admite-se que as variáveis matemáticas pertencem ao conjunto dos inteiros, salvo onde for explicitamente indicado outro conjunto ou onde o conjunto seja fácil de inferir pelo contexto da asserção. Nas asserções é também normal assumir que as variável C++ não têm limitações (e.g., admite-se que uma variável `int` pode guardar qualquer inteiro). Embora isso não seja rigoroso, permite resolver com maior facilidade um grande número de problemas sem que seja necessário introduzir demasiados pormenores nas demonstrações.

Em cada ponto de um programa existe um determinado conjunto de variáveis, cada uma das quais pode tomar determinados valores, consoante o seu tipo. Ao conjunto de todos os possíveis valores de todas as variáveis existentes num dado ponto de um programa chama-se o *espaço de estados* do programa nesse ponto. Ao conjunto dos valores das variáveis existentes num determinado ponto do programa num determinado instante de tempo chama-se o *estado* de um programa. Assim, o estado de um programa é um elemento do espaço de estados.

As asserções fazem afirmações acerca do estado do programa num determinado ponto. Podem ser mais fortes, por exemplo se afirmarem que uma dada variável toma o valor 1, ou mais fracas, por exemplo se afirmarem que uma dada variável toma um valor positivo.

4.2.1 Dedução de asserções

Suponha-se o seguinte troço de programa

```
++n;
```

onde se admite que a variável `n` tem inicialmente um valor não-negativo. Como adicionar asserções a este troço de programa? Em primeiro lugar escreve-se a asserção que corresponde à assunção de que `n` guarda inicialmente um valor não-negativo:

```
// 0 ≤ n
++n;
```

Como se pode verificar, as asserções são colocadas no código na forma de comentários.

A asserção que segue uma instrução, a sua condição objectivo, é tipicamente obtida pela semântica da instrução e pela respectiva pré-condição. Ou seja, dada a PC, a instrução implica a veracidade da respectiva CO. No caso acima é óbvio que

```
// 0 ≤ n.
++n;
// 1 ≤ n.
```

ou seja, se `n` era maior ou igual a zero antes da incrementação, depois da incrementação será forçosamente maior ou igual a um.

A demonstração informal da correcção de um pedaço de código pode, portanto, ser feita recorrendo a asserções. Suponha-se que se pretende demonstrar que o código

```
int const t = x;
x = y;
y = t;
```

troca os valores de duas variáveis `x` e `y` do tipo `int`. Começa-se por escrever as duas principais asserções: a pré-condição e a condição objectivo da sequência completa de instruções. Neste caso a escrita destas asserções é complicada pelo facto de a CO ter de ser referir aos valores das variáveis `x` e `y` *antes das instruções*. Para resolver este problema, considere-se que as variáveis matemáticas x e y representam os valores iniciais das variáveis C++ `x` e `y` (repare-se bem na diferença de tipo de letra usado para variáveis matemáticas e para variáveis do programa C++). Então a CO pode ser escrita simplesmente como

$$\mathtt{x} = y \wedge \mathtt{y} = x$$

e a PC como

$$\mathtt{x} = x \wedge \mathtt{y} = y$$

donde o código com as asserções iniciais e finais é

```
// x = x ∧ y = y.
int const t = x;
x = y;
y = t;
// x = y ∧ y = x.
```

A demonstração de correcção pode ser feita deduzindo as asserções intermédias:

```
// x = x ∧ y = y.
int const t = x;
// x = x ∧ y = y ∧ t = x.
x = y;
// x = y ∧ y = y ∧ t = x.
y = t;
// x = y ∧ y = x ∧ t = x ⇒ x = y ∧ y = x.
```

A demonstração de correcção pode ser feita também partindo dos objectivos. Para cada instrução, começando na última, determina-se quais as condições mínimas a impor às variáveis antes da instrução, i.e., determina-se qual a PC mais fraca a impor à instrução em causa, de modo a que, depois dessa instrução, a sua CO seja verdadeira. Antes do o fazer, porém, é necessário introduzir mais alguns conceitos.

4.2.2 Predicados mais fortes e mais fracos

Diz-se que um predicado P é mais fraco do que outro Q se Q implicar P, ou seja, se o conjunto dos valores que tornam o predicado P verdadeiro contém o conjunto dos valores que tornam o predicado Q verdadeiro. Por exemplo, se $P \equiv 0 < x$ e $Q \equiv x = 1$, então P é mais fraco do que Q, pois o conjunto $\{1\}$ está contido no conjunto dos positivos $\{x : 0 < x\}$. O mais fraco de todos os possíveis predicados é aquele que é sempre verdadeiro, pois o conjunto dos valores que o verificam é o conjunto universal. Logo, o mais fraco de todos os possíveis predicados é $\mathcal{V}$. Por razões óbvias, o mais forte de todos os possíveis predicados é $\mathcal{F}$, sendo vazio o conjunto dos valores que o verificam.

4.2.3 Dedução da pré-condição mais fraca de uma atribuição

É possível estabelecer uma relação entre as asserções que é possível escrever antes e depois de uma instrução de atribuição. Ou seja, é possível relacionar duma forma algébrica PC e CO em

```
// PC
x = expressão;
// CO
```

A relação é mais de estabelecer partindo da condição objectivo CO. É que a PC mais fraca que, depois da atribuição, conduz à CO, pode ser obtida substituindo por `expressão` todas as ocorrências da variável x em CO. Esta dedução da PC mais fraca só pode ser feita se a expressão cujo valor se atribui a x não tiver efeitos laterais, i.e., se não implicar a alteração de nenhuma variável do programa (ver Secção 2.7.8). Se a expressão tiver efeitos laterais mas apesar de tudo for bem comportada, é possível decompô-la numa sequência de instruções e aplicar a dedução a cada uma delas.

Por exemplo, suponha-se que se pretendia saber qual a PC mais fraca para a qual a instrução de atribuição `x = -x;` conduz à CO $0 \leq x$. Aplicando o método descrito conclui-se que

```
// 0 ≤ −x, ou seja, x ≤ 0.
x = -x;
// 0 ≤ x.
```

Ou seja, para que a inversão do sinal de x conduza a um valor não-negativo, o menos que tem de se exigir é que o valor de x seja inicialmente não-positivo.

Voltando ao exemplo da troca de valores de duas variáveis

```
int const t = x;
x = y;
y = t;
// x = y ∧ y = x.
```

e aplicando a técnica proposta obtém-se sucessivamente

```
int const t = x;
x = y;
// x = y ∧ t = x.
y = t;
// x = y ∧ y = x.

int const t = x;
// y = y ∧ t = x.
x = y;
// x = y ∧ t = x.
y = t;
// x = y ∧ y = x.

// y = y ∧ x = x.
int const t = x;
// y = y ∧ t = x.
x = y;
// x = y ∧ t = x.
y = t;
// x = y ∧ y = x.
```

tendo-se recuperado a PC inicial.

Em geral este método não conduz exactamente à PC escrita inicialmente. Suponha-se que se pretendia demonstrar que

```
// x < 0.
x = -x;
// 0 ≤ x.
```

Usando o método anterior conclui-se que:

```
// x < 0.
// 0 ≤ −x, ou seja, x ≤ 0.
x = -x;
// 0 ≤ x.
```

Mas como $x < 0$ implica que $x \leq 0$, conclui-se que o código está correcto.

Em geral, portanto, quando se escrevem duas asserções em sequência, a primeira inserção implica a segunda, ou seja,

```
// A1.
// A2.
```

só pode acontecer se $A_1 \Rightarrow A_2$. Se as duas asserções surgirem separadas por uma instrução, então se a primeira asserção se verificar antes da instrução a segunda asserção verificar-se-á depois da instrução ser executada.

4.2.4 Asserções em instruções de selecção

Suponha-se de novo o troço de código que pretende ordenar os valores das variáveis `x` e `y` (que se presume serem do tipo `int`) de modo que `x` $\leq$ `y`,

```
if(y < x) {
    int const t = x;
    x = y;
    y = t;
}
```

Qual a CO e qual a PC? Considerem-se x e y os valores das variáveis `x` e `y` antes da instrução de selecção. Então a PC e a CO podem ser escritas

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    int const t = x;
    x = y;
    y = t;
}
// CO ≡ x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

Ou seja, o problema fica resolvido quando as variáveis `x` e `y` mantêm ou trocam os seus valores iniciais e `x` termina com um valor não-superior ao de `y`.

Determinar uma CO não é fácil. A PC e a CO, em conjunto, constituem a especificação formal do problema. A sua escrita obriga à compreensão profunda do problema a resolver, e daí a dificuldade.

Será que a instrução condicional conduz forçosamente da PC à CO? É necessário demonstrá-lo.

Partindo da pré-condição

É conveniente começar por converter a instrução condicional na instrução de selecção equivalente e, simultaneamente, explicitar as guardas das instruções alternativas:

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    // G1 ≡ y < x.
    int const t = x;
    x = y;
    y = t;
} else
    // G2 ≡ x ≤ y.
    ; // instrução nula!
```

A PC, sendo verdadeira antes da instrução de selecção, também o será imediatamente antes de qualquer das instruções alternativas, pelo que se pode escrever

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    // y < x ∧ x = x ∧ y = y.
    int const t = x;
    x = y;
    y = t;
} else
    // x ≤ y ∧ x = x ∧ y = y.
    ; // instrução nula!
```

Pode-se agora deduzir as asserções válidas após cada instrução alternativa:

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    // y < x ∧ x = x ∧ y = y.
    int const t = x;
    // y < x ∧ x = x ∧ y = y ∧ y < t ∧ t = x.
    x = y;
    // y = y ∧ y < t ∧ t = x ∧ x = y ∧ x < t.
    y = t;
    // t = x ∧ x = y ∧ x < t ∧ y = x ∧ x < y, que implica
    // x ≤ y ∧ x = y ∧ y = x.
} else
    // x ≤ y ∧ x = x ∧ y = y.
    ; // instrução nula!
    // x ≤ y ∧ x = x ∧ y = y, já que a instrução nula não afecta asserções.
```

Conclui-se que, depois da troca de valores entre `x` e `y` na primeira das instruções alternativas, $\texttt{x} < \texttt{y}$, o que implica que $\texttt{x} \leq \texttt{y}$. Eliminando as asserções intermédias, úteis apenas durante a demonstração,

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    int const t = x;
    x = y;
    y = t;
    // x ≤ y ∧ x = y ∧ y = x.
} else
    ; // instrução nula!
    // x ≤ y ∧ x = x ∧ y = y, já que a instrução nula não afecta asserções.
// Que asserção é válida aqui?
```

Falta agora deduzir a asserção válida depois da instrução de selecção completa. Esse ponto pode ser atingido depois de se ter passado por qualquer uma das instruções alternativas, pelo que uma asserção que é válida certamente é a disjunção das asserções deduzidas para cada uma das instruções alternativas:

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    int const t = x;
    x = y;
    y = t;
    // x ≤ y ∧ x = y ∧ y = x.
} else
    ; // instrução nula!
```

```
        // $\texttt{x} \leq \texttt{y} \land \texttt{x} = x \land \texttt{y} = y$, já que a instrução nula não afecta asserções.
// $(\texttt{x} \leq \texttt{y} \land \texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} \leq \texttt{y} \land \texttt{x} = y \land \texttt{y} = x)$, ou seja
// $\texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
```

que é exactamente a CO que se pretendia demonstrar válida.

Partindo da condição objectivo

Neste caso começa por se observar que a CO tem de ser válida no final de qualquer das instruções alternativas para que o possa ser no final da instrução de selecção:

```
if(y < x) {
    int const t = x;
    x = y;
    y = t;
    // $\texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
} else
    ; // instrução nula!
    // $\texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
// $CO \equiv \texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
```

Depois vão-se determinando sucessivamente as pré-condições mais fracas de cada instrução (do fim para o início) usando as regras descritas acima para as atribuições:

```
if(y < x) {
    // $\texttt{y} \leq \texttt{x} \land ((\texttt{y} = x \land \texttt{x} = y) \lor (\texttt{y} = y \land \texttt{x} = x))$.
    int const t = x;
    // $\texttt{y} \leq \texttt{t} \land ((\texttt{y} = x \land \texttt{t} = y) \lor (\texttt{y} = y \land \texttt{t} = x))$.
    x = y;
    // $\texttt{x} \leq \texttt{t} \land ((\texttt{x} = x \land \texttt{t} = y) \lor (\texttt{x} = y \land \texttt{t} = x))$.
    y = t;
    // $\texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
} else
    // $\texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
    ; // instrução nula!
    // $\texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
// $CO \equiv \texttt{x} \leq \texttt{y} \land ((\texttt{x} = x \land \texttt{y} = y) \lor (\texttt{x} = y \land \texttt{y} = x))$.
```

Eliminando as asserções intermédias obtém-se:

```
if(y < x) {
    // $\texttt{y} \leq \texttt{x} \land ((\texttt{y} = x \land \texttt{x} = y) \lor (\texttt{y} = y \land \texttt{x} = x))$.
    int const t = x;
    x = y;
```

```
        y = t;
    } else
        // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
        ; // instrução nula!
    // CO ≡ x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

Basta agora verificar se a PC em conjunção com cada uma das guardas implica a respectiva asserção deduzida. Ou seja, sabendo o que se sabe desde o início, a pré-condição PC, e sabendo que a primeira instrução alternativa será executada, e portanto a guarda G_1, será que a pré-condição mais fraca dessa instrução alternativa se verifica? E o mesmo para a segunda instrução alternativa? Resumindo, tem de se verificar se:

1. $PC \wedge G_1 \Rightarrow \mathtt{y} \leq \mathtt{x} \wedge ((\mathtt{y} = x \wedge \mathtt{x} = y) \vee (\mathtt{y} = y \wedge \mathtt{x} = x))$ e
2. $PC \wedge G_2 \Rightarrow \mathtt{x} \leq \mathtt{y} \wedge ((\mathtt{x} = x \wedge \mathtt{y} = y) \vee (\mathtt{x} = y \wedge \mathtt{y} = x))$ são implicações verdadeiras.

É fácil verificar que o são de facto.

Resumo

Em geral, para demonstrar a correcção de uma instrução de selecção com n alternativas, ou seja, com n instruções alternativas

```
// PC
if(C₁)
    // G₁ ≡ C₁.
    instrução₁
else if(C₂)
    // G₂ ≡ ¬G₁ ∧ C₂  (ou, ¬C₁ ∧ C₂).
    instrução₂
else if(C₃)
    // G₃ ≡ ¬G₁ ∧ ¬G₂ ∧ C₃  (ou, ¬C₁ ∧ ¬C₂ ∧ C₃).
    instrução₃
...
else if(Cₙ₋₁)
    // Gₙ₋₁ ≡ ¬G₁ ∧ ··· ∧ ¬Gₙ₋₂ ∧ Cₙ₋₁  (ou, ¬C₁ ∧ ··· ∧ ¬Cₙ₋₂ ∧ Cₙ₋₁).
    instruçãoₙ₋₁
else
    // Gₙ ≡ ¬G₁ ∧ ··· ∧ ¬Gₙ₋₁  (ou, ¬C₁ ∧ ··· ∧ ¬Cₙ₋₁).
    instruçãoₙ
// CO
```

seguem-se os seguintes passos:

Demonstração directa Partindo da PC:

1. Para cada instrução alternativa $\texttt{instrução}_i$ (com $i = 1 \ldots n$), deduz-se a respectiva CO_i admitindo que a pré-condição PC da instrução de selecção e a guarda G_i da instrução alternativa são ambas verdadeiras. Ou seja, deduz-se CO_i tal que:

   ```
   // PC ∧ G_i
   instrução_i
   // CO_i
   ```

2. Demonstra-se que $(CO_1 \vee CO_2 \vee \cdots \vee CO_n) \Rightarrow CO$. Ou, o que é o mesmo, que $(CO_1 \Rightarrow CO) \wedge (CO_2 \Rightarrow CO) \wedge \cdots \wedge (CO_n \Rightarrow CO)$.

Demonstração inversa Partindo da CO:

1. Para cada instrução alternativa $\texttt{instrução}_i$ (com $i = 1 \ldots n$), determina-se a pré-condição mais fraca PC_i que leva forçosamente à CO da instrução de selecção. Ou seja, determina-se a PC_i mais fraca tal que:

   ```
   // PC_i
   instrução_i
   // CO
   ```

2. Demonstra-se que $PC \wedge G_i \Rightarrow PC_i$ para $i = 1 \ldots n$.

Não é necessário verificar se pelo menos uma guarda é sempre verdadeira, porque, por construção da instrução de selecção, esta termina sempre com um `else`. Se isso não acontecer, é necessário fazer a demonstração para a instrução de selecção equivalente em que todos os `if` têm o respectivo `else`, o que pode implicar introduzir uma instrução alternativa nula.

4.3 Desenvolvimento de instruções de selecção

As secções anteriores apresentaram a noção de asserção e sua relação com as instruções de selecção. Falta agora ver como esse formalismo pode ser usado para desenvolver programas.

Regresse-se ao problema inicial, da escrita de uma função para calcular o valor absoluto de um valor inteiro. O objectivo é, portanto, preencher o corpo da função

```
/** Devolve o valor absoluto do argumento.
    @pre   PC ≡ V (ou seja, sem restrições).
    @post  CO ≡ absoluto = |x|, ou seja,
              0 ≤ absoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    ...
}
```

onde se usou o predicado $\mathcal{V}$ (verdadeiro) para indicar que a função não tem pré-condição.

Para simplificar o desenvolvimento, pode-se começar o corpo pela definição de uma variável local para guardar o resultado e terminar com a devolução do seu valor, o que permite escrever as asserções principais da função em termos do valor desta variável[2]:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
               0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
    ...
    // CO ≡ r = |x|, ou seja, 0 ≤ r ∧ (r = −x ∨ r = x).

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

Antes de começar o desenvolvimento, é necessário perceber se para a resolução do problema é necessário recorrer a uma instrução de selecção. Neste caso é óbvio que sim, pois tem de se discriminar entre valores negativos e positivos (e talvez nulos) de `x`.

O desenvolvimento usado será baseado nos objectivos. Este é um princípio importante da programação [8] *a programação deve ser orientada pelos objectivos*. É certo que a pré-condição afecta a solução de qualquer problema, mas os problemas são essencialmente determinados pela condição objectivo. De resto, com se pode verificar depois de alguma prática, mais inspiração para a resolução de um problema pode ser obtida por análise da condição objectivo do que por análise da pré-condição. Por exemplo, é comum não haver qualquer pré-condição na especificação de um problema, donde nesses casos só a condição objectivo poderá ser usada como fonte de inspiração.

4.3.1 Escolha das instruções alternativas

O primeiro passo do desenvolvimento corresponde a identificar possíveis instruções alternativas que pareçam poder levar à veracidade da CO. É fácil verificar que há duas possíveis

[2] A semântica de uma instrução de retorno é muito semelhante à de uma instrução de atribuição. A pré-condição mais fraca pode ser obtida por substituição, na CO da função,do nome da função pela expressão usada na instrução de retorno. Assim, a pré-condição mais fraca da instrução `return r;` que leva à condição objectivo

$$CO \equiv \texttt{absoluto} = |x|\text{, ou seja, } 0eq\texttt{absoluto} \wedge (\texttt{absoluto} = -\texttt{x} \vee \texttt{absoluto} = \texttt{x}).$$

é

$$CO \equiv \texttt{r} = |x|\text{, ou seja, } 0 \leq \texttt{r} \wedge (\texttt{r} = -\texttt{x} \vee \texttt{r} = \texttt{x}).$$

No entanto, a instrução de retorno difere da instrução de atribuição pelo numa coisa: a instrução de retorno termina a execução da função.

instruções nessas condições: `r = -x;` e `r = x;`. Estas possíveis instruções podem ser obtidas por simples observação da CO.

4.3.2 Determinação das pré-condições mais fracas

O segundo passo corresponde a verificar em que circunstâncias estas instruções alternativas levam à veracidade da CO. Ou seja, quais as pré-condições PC_i mais fracas que garantem que, depois da respectiva instrução i, a condição CO é verdadeira. Comece-se pela primeira instrução:

```
r = -x;
// CO ≡ 0 ≤ r ∧ (r = −x ∨ r = x).
```

Usando a regra da substituição discutida na Secção 4.2.3, chega-se a

```
// CO ≡ 0 ≤ −x ∧ (−x = −x ∨ −x = x), ou seja, x ≤ 0 ∧ (V ∨ x = 0), ou seja, x ≤ 0.
r = -x;
// CO ≡ 0 ≤ r ∧ (r = −x ∨ r = x).
```

Logo, a primeira instrução, `r = -x;`, só conduz aos resultados pretendidos desde que `x` tenha inicialmente um valor não-positivo, i.e., $PC_1 \equiv$ $\texttt{x} \leq 0$.

A mesma verificação pode ser feita para a segunda instrução

```
// CO ≡ 0 ≤ x ∧ (x = −x ∨ x = x), ou seja, 0 ≤ x ∧ (x = 0 ∨ V), ou seja, 0 ≤ x.
r = x;
// CO ≡ 0 ≤ r ∧ (r = −x ∨ r = x).
```

Logo, a segunda instrução, `r = x;`, só conduz aos resultados pretendidos desde que `x` tenha inicialmente um valor não-negativo, i.e., $PC_2 \equiv 0 \leq \texttt{x}$.

O corpo da função pode-se agora escrever

```
/** Devolve o valor absoluto do argumento.
    @pre   PC ≡ V (ou seja, sem restrições).
    @post  CO ≡ absoluto = |x|, ou seja,
               0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
    if(C1)
        // G1
        // PC1 ≡ x ≤ 0.
        r = -x;
```

```
    else
        // $G_2$
        // $PC_2 \equiv 0 \leq x$.
        r = x;
    // $CO \equiv r = |x|$, ou seja, $0 \leq r \wedge (r = -x \vee r = x)$.

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

4.3.3 Determinação das guardas

O terceiro passo corresponde a determinar as guardas G_i de cada uma das instruções alternativas. De acordo com o que se viu anteriormente, para que a instrução de selecção resolva o problema, é necessário que $PC \wedge G_i \Rightarrow PC_i$. Só assim se garante que, sendo a guarda G_i verdadeira, a instrução alternativa i conduz à condição objectivo desejada. Neste caso PC é sempre $\mathcal{V}$, pelo que dizer que $PC \wedge G_i \Rightarrow PC_i$ é o mesmo que dizer que $G_i \Rightarrow PC_i$, e portanto a forma mais simples de escolher as guardas é fazer simplesmente $G_i = PC_i$. Ou seja,

```
/** Devolve o valor absoluto do argumento.
    @pre  $PC \equiv \mathcal{V}$ (ou seja, sem restrições).
    @post $CO \equiv$ absoluto $= |x|$, ou seja,
              $0 eq$ absoluto $\wedge$ (absoluto $= -x \vee$ absoluto $= x$). */
int absoluto(int const x)
{
    // $PC \equiv \mathcal{V}$ (ou seja, sem restrições).
    int r;
    if($C_1$)
        // $G_1 \equiv x \leq 0$.
        r = -x;
    else
        // $G_2 \equiv 0 eq x$.
        r = x;
    // $CO \equiv r = |x|$, ou seja, $0 eq r \wedge (r = -x \vee r = x)$.

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

4.3.4 Verificação das guardas

O quarto passo corresponde a verificar se a pré-condição PC da instrução de selecção implica a veracidade de pelo menos uma das guardas G_i das instruções alternativas. Se isso não acontecer, significa que pode haver casos para os quais nenhuma das guardas seja verdadeira. Se isso acontecer o problema ainda não está resolvido, sendo necessário determinar instruções alternativas adicionais e respectivas guardas até todos os possíveis casos estarem cobertos.

Neste caso a PC não impõe qualquer restrição, ou seja $PC \equiv \mathcal{V}$. Logo, tem de se verificar se $\mathcal{V} \Rightarrow (G_1 \vee G_2)$. Neste caso $G_1 \vee G_2$ é $\mathtt{x} \leq 0 \vee 0 \leq \mathtt{x}$, ou seja, $\mathcal{V}$. Como $\mathcal{V} \Rightarrow \mathcal{V}$, o problema está quase resolvido.

4.3.5 Escolha das condições

No quinto e último passo determinam-se as condições das instruções de selecção encadeadas de modo a obter as guardas entretanto determinadas. De acordo com o que se viu na Secção 4.2.4, $G_1 = C_1$, pelo que a função fica

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
    if(x <= 0)
        // G1 ≡ x ≤ 0.
        r = -x;
    else
        // G2 ≡ 0 < x.
        r = x;
    // CO ≡ r = |x|, ou seja, 0eqr ∧ (r = −x ∨ r = x).

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

A segunda guarda foi alterada, pois de acordo com a Secção 4.2.4 $G_2 = \neg G_1 = 0 < \mathtt{x}$. Esta guarda é mais forte que a guarda originalmente determinada, pelo que a alteração não traz qualquer problema. Na realidade o que aconteceu foi que a semântica da instrução `if` do C++ forçou à escolha de qual das instruções alternativas lida com o caso $\mathtt{x} = 0$.

Finalmente podem-se eliminar as asserções intermédias:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
```

```
{
    int r;
    if(x <= 0)
        // G1 ≡ x ≤ 0.
        r = -x;
    else
        // G2 ≡ 0 < x.
        r = x;

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

4.3.6 Alterando a solução

A solução obtida pode ser simplificada se se observar que, depois de terminada a instrução de selecção, a função se limita a devolver o valor guardado em `r` por uma das duas instruções alternativas: é possível eliminar essa variável e devolver imediatamente o valor apropriado:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
        // G1 ≡ x ≤ 0.
        return -x;
    else
        // G2 ≡ 0 < x.
        return x;
}
```

Por outro lado, se a primeira instrução alternativa (imediatamente abaixo do `if`) termina com uma instrução `return`, então não é necessária uma instrução de selecção, bastando uma instrução condicional:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
```

```
            return -x;
        return x;
    }
```

Este último formato não é forçosamente mais claro ou preferível ao anterior, mas é muito comum encontrá-lo em programas escritos em C++. É uma expressão idiomática do C++. Uma desvantagem destes formatos é que não permitem usar instruções de asserção para verificar a validade da condição objectivo.

4.3.7 Metodologia

Existem (pelo menos) dois métodos semi-formais para o desenvolvimento de instruções de selecção. O primeiro foi usado para o desenvolvimento na secção anterior, e parte dos objectivos:

1. Determina-se n instruções `instrução`$_i$, com $i = 1 \ldots n$, que pareçam poder levar à veracidade da CO. Tipicamente estas instruções são obtidas por análise da CO.
2. Determina-se as pré-condições mais fracas PC_i de cada uma dessas instruções de modo que conduzam à CO.
3. Determina-se uma guarda G_i para cada alternativa i de modo que $PC \wedge G_i \Rightarrow PC_i$. Se o problema não tiver pré-condição (ou melhor, se $PC \equiv \mathcal{V}$), então pode-se fazer $G_i = PC_i$. Note-se que uma guarda $G_i = \mathcal{F}$ resolve sempre o problema, embora seja tão forte que nunca leve à execução da instrução respectiva! Por isso *deve-se sempre escolher as guardas o mais fracas possível*[3].
4. Verifica-se se, em todas as circunstâncias, pelo menos uma das guardas encontradas é verdadeira. Isto é, verifica-se se $PC \Rightarrow (G_1 \vee \cdots \vee G_n)$. Se isso não acontecer, é necessário acrescentar mais instruções alternativas às encontradas no ponto 1. Para o fazer, identifica-se que casos ficaram por cobrir analisando as guardas já encontradas e a PC.

No segundo método tenta-se primeiro determinar as guardas e só depois se desenvolve as respectivas instruções alternativas:

1. Determina-se os n casos possíveis interessantes, restritos àqueles que verificam a PC, que cobrem todas as hipóteses. Cada caso possível corresponde a uma guarda G_i. A verificação da PC tem de implicar a verificação de pelo menos uma das guardas, ou seja, $PC \Rightarrow (G_1 \vee G_n)$.
2. Para cada alternativa i, encontra-se uma instrução `instrução`$_i$ tal que

   // $PC \wedge G_i$
   `instrução`$_i$
   // CO_i
   // CO

[3]Excepto, naturalmente, quando se reconhece que as guardas se sobrepõem. Nesse caso pode ser vantajoso fortalecer as guardas de modo a minimizar as sobreposições, desde que isso não conduza a guardas mais complicadas: a simplicidade é uma enorme vantagem.

ou seja, tal que, se a pré-condição PC e a guarda G_i forem verdadeiras, então a instrução leve forçosamente à veracidade da condição objectivo CO.

Em qualquer dos casos, depois de encontradas as guardas e as respectivas instruções alternativas, é necessário escrever a instrução de selecção com o número de instruções `if` apropriado (para n instruções alternativas são necessárias $n-1$ instruções `if` encadeadas) e escolher as condições C_i apropriadas de modo a obter as guardas pretendidas. Pode-se começar por fazer $C_i = G_i$, com $i = 1 \cdots n-1$, e depois identificar sobreposições e simplificar as condições C_i. Muitas vezes as guardas encontrada contêm sobreposições (e.g., $0 \leq$ `x` e `x` ≤ 0 sobrepõem-se no valor 0) que podem ser eliminadas ao escolher as condições de cada `if`.

4.3.8 Discussão

As vantagens das metodologias informais apresentadas face a abordagens mais ou menos *ad-hoc* do desenvolvimento são pelo menos:

1. Forçam à especificação rigorosa do problema, e portanto à sua compreensão profunda.
2. O desenvolvimento é acompanhado da demonstração de correcção, o que reduz consideravelmente a probabilidade de erros.
3. Não são descurados aparentes pormenores que, na realidade, podem dar origem a erros graves e difíceis de corrigir.

É claro que a experiência do programador e a sua maior ou menor inventiva muitas vezes levem a desenvolvimentos "ao sabor da pena". Não é forçosamente errado, desde que feito em consciência. Recomenda-se, nesses casos, que se tente fazer *a posteriori* uma demonstração de correcção (e não um teste!) para garantir que a inspiração funcionou...

Para se verificar na prática a importância da especificação cuidada do problema, tente-se resolver o seguinte problema:

Escreva uma função que, dados dois argumentos do tipo `int`*, devolva o booleano verdadeiro se o primeiro for múltiplo do segundo e falso no caso contrário.*

Neste ponto o leitor deve parar de ler e tentar resolver o problema.

⋮

⋮

A abordagem típica do programador é considerar que um inteiro n é múltiplo de outro inteiro m se o resto da divisão de n por m for zero, e passar directo ao código:

```
bool éMúltiplo(int const m, int const n)
{
    if(m % n == 0)
```

```
        return true;
    else
        return false;
}
```

Implementada a função, o programador fornece-a à sua equipa para utilização. Depois de algumas semanas de utilização, o programa em desenvolvimento, na fase de testes, aborta com uma mensagem (ambiente Linux):

```
Floating exception (core dumped)
```

Depois de umas horas no depurador, conclui-se que, se o segundo argumento da função for zero, a função tenta uma divisão por zero, com a consequente paragem do programa.

Neste ponto o leitor deve parar de ler e tentar resolver o problema.

⋮

⋮

Chamado o programador original da função, este observa que não há múltiplos de zero, e portanto tipicamente corrige a função para

```
bool éMúltiplo(int const m, int const n)
{
    if(n != 0)
        if(m % n == 0)
            return true;
        else
            return false;
    else
        return false;
}
```

Corrigida a função e integrada de novo no programa em desenvolvimento, e repetidos os testes, não se encontra qualquer erro e o desenvolvimento continua. Finalmente o programa é fornecido ao cliente. O cliente utiliza o programa durante meses até que detecta um problema estranho, incompreensível. Como tem um contrato de manutenção com a empresa que desenvolveu o programa, comunica-lhes o problema. A equipa de manutenção, da qual o programador original não faz parte, depois de algumas horas de execução do programa em modo depuração e de tentar reproduzir as condições em que o erro ocorre (que lhe foram fornecidas de uma forma parcelar pelo cliente), acaba por detectar o problema: a função devolve `false` quando lhe são passados dois argumentos zero! Mas zero é múltiplo de zero! Rogando pragas ao programador original da função, esta é corrigida para:

```
bool éMúltiplo(int const m, int const n)
{
    if(n != 0)
        if(m % n == 0)
            return true;
        else
            return false;
    else
        if(m == 0)
            return true;
        else
            return false;
}
```

O código é testado e verifica-se que os erros estão corrigidos.

Depois, o programador olha para o código e acha-o demasiado complicado: para quê as instruções de selecção se uma simples instrução de retorno basta?

Neste ponto o leitor deve parar de ler e tentar resolver o problema com uma única instrução de retorno.

⋮

⋮

Basta devolver o resultado de uma expressão booleana que reflicta os casos em que `m` é múltiplo de `n`:

```
bool éMúltiplo(int const m, int const n)
{
    return (m % n == 0 and n != 0) or (m == 0 and n == 0);
}
```

Como a alteração é meramente cosmética, o programador não volta a testar e o programa corrigido é fornecido ao cliente. Poucas horas depois de ser posto ao serviço o programa aborta. O cliente recorre de novo aos serviços de manutenção, desta vez furioso. A equipa de manutenção verifica rapidamente que a execução da função leva a uma divisão por zero como originalmente. Desta vez a correcção do problema é simples: basta inverter os operandos da primeira conjunção:

```
bool éMúltiplo(int const m, int const n)
{
    return (n != 0 and m % n == 0) or (m == 0 and n == 0);
}
```

Esta simples troca corrige o problema porque nos operadores lógicos o cálculo é atalhado, i.e., o operando esquerdo é calculado em primeiro lugar e, se o resultado ficar imediatamente determinado (ou seja, se o primeiro operando numa conjunção for falso ou se o primeiro operando numa disjunção for verdadeiro) o operando direito não chega a ser calculado (ver Secção 2.7.3).

A moral desta estória é que "quanto mais depressa, mais devagar"... O programador deve evitar as soluções rápidas, pouco pensadas e menos verificadas.

Ah! E faltou dizer que há uma solução ainda mais simples:

```
bool éMúltiplo(int const m, int const n)
{
    return (n != 0 and m % n == 0) or m == 0;
}
```

Talvez tivesse sido boa ideia ter começado por especificar a função. Se tal tivesse acontecido ter-se-ia evitado os erros e ter-se-ia chagado imediatamente à solução mais simples...

4.3.9 Outro exemplo de desenvolvimento

Apresenta-se brevemente o desenvolvimento de uma outra instrução alternativa, um pouco mais complexa. Neste caso pretende-se escrever um procedimento de interface

```
void limita(int& x, int const mín, int const máx)
```

que limite o valor de `x` ao intervalo $[\mathtt{mín}, \mathtt{máx}]$, onde se assume que $\mathtt{mín} \leq \mathtt{máx}$. I.e., se o valor de `x` for inferior a `mín`, então deve ser alterado para `mín`, se for superior a `máx`, deve ser alterado para `máx`, e se pertencer ao intervalo, deve ser deixado com o valor original.

Como habitualmente, começa por se escrever a especificação do procedimento, onde x é uma variável matemática usada para representar o valor inicial da variável de programa `x`:

```
/** Limita x ao intervalo [mín, máx].
    @pre  PC ≡ mín ≤ máx ∧ x = x.
    @post CO ≡ (x = mín ∧ x ≤ mín) ∨ (x = máx ∧ máx ≤ x)∨
               (x = x ∧ mín ≤ x ≤ máx). */
void limita(int& x, int const mín, int const máx)
{
}
```

A observação da condição objectivo conduz imediatamente às seguintes possíveis instruções alternativas:

1. `;` // para manter `x` com o valor inicial x.
2. `x = mín;`

3. `x = máx;`

As pré-condições mais fracas para que se verifique a condição objectivo do procedimento depois de cada uma das instruções são

$$\begin{aligned}
PC_1 &\equiv (\mathtt{x} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{x} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee (\mathtt{x} = \mathtt{máx} \wedge \mathtt{máx} \leq x) \\
PC_2 &\equiv (\mathtt{mín} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{mín} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{máx} \leq x) \\
&\equiv (\mathtt{mín} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee x \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{máx} \leq x) \\
PC_3 &\equiv (\mathtt{máx} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{máx} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee (\mathtt{máx} = \mathtt{máx} \wedge \mathtt{máx} \leq x) \\
&\equiv (\mathtt{máx} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{máx} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee \mathtt{máx} \leq x
\end{aligned}$$

A determinação das guardas faz-se de modo a que $PC \wedge G_i \Rightarrow PC_i$.

No primeiro caso tem-se da pré-condição PC que $\mathtt{x} = x$, pelo que a guarda mais fraca é

$$\begin{aligned}
G_1 &\equiv (\mathtt{mín} \leq \mathtt{x} \wedge \mathtt{x} \leq \mathtt{máx}) \vee \mathtt{x} = \mathtt{mín} \vee \mathtt{x} = \mathtt{máx} \\
&\equiv \mathtt{mín} \leq \mathtt{x} \wedge \mathtt{x} \leq \mathtt{máx}
\end{aligned}$$

pois os casos $\mathtt{x} = \mathtt{mín}$ e $\mathtt{x} = \mathtt{máx}$ são cobertos pela primeira conjunção dado que a pré-condição PC garante que $\mathtt{mín} \leq \mathtt{máx}$.

No segundo caso, pelas mesmas razões, tem-se que

$$\begin{aligned}
G_2 &\equiv (\mathtt{mín} = \mathtt{x} \wedge \mathtt{mín} \leq \mathtt{x} \leq \mathtt{máx}) \vee \mathtt{x} \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{máx} \leq \mathtt{x}) \\
&\equiv \mathtt{mín} = \mathtt{x} \vee \mathtt{x} \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{máx} \leq \mathtt{x}) \\
&\equiv \mathtt{x} \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{máx} \leq \mathtt{x})
\end{aligned}$$

pois da pré-condição PC sabe-se que $\mathtt{mín} \leq \mathtt{máx}$, logo se $\mathtt{x} = \mathtt{mín}$ também será $\mathtt{x} \leq \mathtt{máx}$.

Da mesma forma se obtém

$$G_3 \equiv (\mathtt{máx} = \mathtt{mín} \wedge \mathtt{x} \leq \mathtt{mín}) \vee \mathtt{máx} \leq \mathtt{x}$$

É evidente que há sempre pelo menos uma destas guardas válidas quaisquer que sejam os valores dos argumentos verificando a PC. Logo, não são necessárias quaisquer outras instruções alternativas.

Ou seja, a estrutura da instrução de selecção é (trocando a ordem das instruções de modo a que a instrução nula fique em último lugar):

```
if(C1)
    // G2 ≡ x ≤ mín ∨ (mín = máx ∧ máx ≤ x).
    x = mín;
else if(C2)
    // G3 ≡ (máx = mín ∧ x ≤ mín) ∨ máx ≤ x.
    x = máx;
else
    // G1 ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

A observação das guardas demonstra que elas são redundantes para algumas combinações de valores. Por exemplo, se `mín` = `máx` qualquer das guardas G_2 e G_3 se verifica. É fácil verificar, portanto, que as guardas podem ser reforçadas para:

```
if(C₁)
    // G₂ ≡ x ≤ mín.
    x = mín;
else if(C₂)
    // G₃ ≡ máx ≤ x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

de modo a que, caso `mín` = `máx`, seja válida apenas uma das guardas (excepto, claro, quando `x` = `mín` = `máx`, em que se verificam de novo as duas guardas). De igual modo se podem eliminar as redundâncias no caso de `x` ter como valor um dos extremos do intervalo, pois nesse caso a guarda G_1 "dá conta do recado":

```
if(C₁)
    // G₂ ≡ x < mín.
    x = mín;
else if(C₂)
    // G₃ ≡ máx < x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

As condições das instruções `if` são:

```
if(x < mín)
    // G₂ ≡ x < mín.
    x = mín;
else if(máx < x)
    // G₃ ≡ máx < x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

Finalmente, pode-se eliminar o último `else`, pelo que o procedimento fica, já equipado com as instruções de asserção:

```
/** Limita x ao intervalo [mín, máx].
    @pre PC ≡ mín ≤ máx ∧ x = x.
    @post CO ≡ (x = mín ∧ x ≤ mín) ∨ (x = máx ∧ máx ≤ x)∨
               (x = x ∧ mín ≤ x ≤ máx). */
void limita(int& x, int const mín, int const máx)
{
    assert(mín <= máx);

    if(x < mín)
        x = mín;
    else if(máx < x)
        x = máx;

    assert(mín <= x and x <= máx);
}
```

4.4 Variantes das instruções de selecção

4.4.1 O operador ? :

Seja a função que calcula o valor absoluto de um número desenvolvida nas secções anteriores:

```
/** Devolve o valor absoluto do argumento.
    @pre PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
               0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
        return -x;
    return x;
}
```

Será possível simplificá-la mais? Sim. A linguagem C++ fornece um operador ternário (com três operandos), que permite escrever a solução como[4]

```
/** Devolve o valor absoluto do argumento.
    @pre PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
               0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
```

[4]O leitor mais atento notou que se alterou a instrução que lida com o caso em que $x = 0$. É que trocar o sinal de zero é uma simples perda de tempo...

```
    return x < 0 ? -x : x;
}
```

Este operador `? :`, também conhecido por *se aritmético*, tem a seguinte sintaxe:

```
condição ? expressão1 : expressão2
```

O resultado do operador é o resultado da expressão *expressão1*, se *condição* for verdadeira (nesse caso *expressão2* não chega a ser calculada), ou o resultado da expressão *expressão2*, se `condição` for falsa (nesse caso *expressão1* não chega a ser calculada).

4.4.2 A instrução switch

Suponha-se que se pretende escrever um procedimento que, dado um inteiro entre 0 e 9, o escreve no ecrã por extenso e em português (escrevendo "erro" se o inteiro for inválido). Os nomes dos dígitos decimais em português, como em qualquer outra língua natural, não obedecem a qualquer regra lógica. Assim, o procedimento terá de lidar com cada um dos 10 casos separadamente. Usando a instrução de selecção encadeada:

```
/** Escreve no ecrã, por extenso, um dígito inteiro entre 0 e 9.
    @pre  PC ≡ 0 ≤ dígito < 10.
    @post CO ≡ ecrã contém, para além do que continha originalmente, o dígito
               dado pelo inteiro dígito. */
void escreveDígitoPorExtenso(int const dígito)
{
    assert(0 <= dígito and dígito < 10);

    if(dígito == 0)
        cout << "zero";
    else if(dígito == 1)
        cout << "um";
    else if(dígito == 2)
        cout << "dois";
    else if(dígito == 3)
        cout << "três";
    else if(dígito == 4)
        cout << "quatro";
    else if(dígito == 5)
        cout << "cinco";
    else if(dígito == 6)
        cout << "seis";
    else if(dígito == 7)
        cout << "sete";
    else if(dígito == 8)
        cout << "oito";
```

```
    else
        cout << "nove";
}
```

Existe uma solução mais usual para este problema e que faz uso da instrução de selecção `switch`. Quando é necessário comparar uma variável com um número discreto de diferentes valores, e executar uma acção diferente em cada um dos casos, deve-se usar esta instrução. Esta instrução permite clarificar a solução do problema apresentado:

```
/** Escreve no ecrã, por extenso, um dígito inteiro entre 0 e 9.
    @pre  PC ≡ 0 ≤ dígito < 10.
    @post CO ≡ ecrã contém, para além do que continha originalmente, o dígito
               dado pelo inteiro dígito. */
void escreveDígitoPorExtenso(int const dígito)
{
    assert(0 <= dígito and dígito < 10);

    switch(dígito) {
      case 0:
        cout << "zero";
        break;

      case 1:
        cout << "um";
        break;

      case 2:
        cout << "dois";
        break;

      case 3:
        cout << "três";
        break;

      case 4:
        cout << "quatro";
        break;

      case 5:
        cout << "cinco";
        break;

      case 6:
        cout << "seis";
        break;
```

```
        case 7:
          cout << "sete";
          break;

        case 8:
          cout << "oito";
          break;

        case 9:
          cout << "nove";
  }
```

Esta instrução não permite a especificação de gamas de valores nem de desigualdades: construções como `case 1..10:` ou `case < 10:` são inválidas. Assim, é possível usar como expressão de controlo do `switch` (i.e., a expressão que se coloca entre parênteses após a palavra-chave `switch`) apenas expressões de um dos tipos inteiros ou de um tipo enumerado (ver Capítulo 6), devendo as constantes colocadas nos casos a diferenciar ser do mesmo tipo.

É possível agrupar vários casos ou alternativas:

```
switch(valor) {
  case 1:
  case 2:
  case 3:
    cout << "1, 2 ou 3";
    break;
  ...
}
```

Isto acontece porque a construção `case` n`:` apenas indica qual o ponto de entrada nas instruções que compõem o `switch` quando a sua expressão de controlo tem valor n. A execução do corpo do `switch` (o bloco de instruções entre `{}`) só termina quando for atingida a chaveta final ou quando for executada uma instrução de `break`. Terminado o `switch`, a execução continua sequencialmente após a chaveta final. A consequência deste "agulhamento" do fluxo de instrução é que, se no exemplo anterior se eliminarem os `break`

```
/** Escreve no ecrã, por extenso, um dígito inteiro entre 0 e 9.
    @pre  PC ≡ 0 ≤ dígito < 10.
    @post CO ≡ ecrã contém, para além do que continha originalmente, o dígito
               dado pelo inteiro dígito. */
void escreveDígitoPorExtenso(int const dígito)
{
    // Código errado!
    switch(dígito) {
      case 0:
        cout << "zero";
```

```
        case 1:
          cout << "um";

        case 2:
          cout << "dois";

        case 3:
          cout << "três";

        case 4:
          cout << "quatro";

        case 5:
          cout << "cinco";

        case 6:
          cout << "seis";

        case 7:
          cout << "sete";

        case 8:
          cout << "oito";

        case 9:
          cout << "nove";
    }
```

uma chamada `escreveDígitoPorExtenso(7)` resulta em:

```
seteoitonove
```

que não é de todo o que se pretendia!

Pelas razões que se indicaram atrás, não é possível usar a instrução `switch` (pelo menos de uma forma elegante) como alternativa às instruções de selecção em:

```
/** Escreve no ecrã a ordem de grandeza de um valor.
    @pre  PC ≡ 0 ≤ v < 10000.
    @post CO ≡ ecrã contém, para além do que continha originalmente, a
               ordem de grandeza de v. */
void escreveGrandeza(double v)
{
    assert(0 <= v and v < 10000);
```

```
    if(v < 10)
        cout << "unidades";
    else if(v < 100)
        cout << "dezenas";
    else if(v < 1000)
        cout << "centenas";
    else
        cout << "milhares";
}
```

A ordem pela qual se faz as comparações em instruções de selecção encadeadas pode ser muito relevante em termos do tempo de execução do programa, embora seja irrelevante no que diz respeito aos resultados. Suponha-se que os valores passados como argumento a `escreveGrandeza()` são equiprováveis, i.e., é tão provável ser passado um número como qualquer outro. Nesse caso consegue-se demonstrar que, em média, são necessárias 2,989 comparações ao executar o procedimento e que, invertendo a ordem das instruções alternativas para

```
/** Escreve no ecrã a ordem de grandeza de um valor.
    @pre  PC ≡ 0 ≤ v < 10000.
    @post CO ≡ ecrã contém, para além do que continha originalmente, a
               ordem de grandeza de v. */
void escreveGrandeza(double v)
{
    assert(0 <= v and v < 10000);

    if(1000 <= x)
        cout << "milhares";
    else if(100 <= x)
        cout << "centenas";
    else if(10 <= x)
        cout << "dezenas";
    else
        cout << "unidades";
}
```

são necessárias 1,11 comparações[5]!

No caso da instrução de selecção `switch`, desde que todos os conjuntos de instruções tenham o respectivo `break`, a ordem dos casos é irrelevante.

4.5 Instruções de iteração

Na maioria dos programas há conjuntos de operações que é necessário repetir várias vezes. Para controlar o fluxo do programa de modo a que um conjunto de instruções sejam repetidos

[5]Demonstre-o!

condicionalmente (em ciclos) *usam*-se as *instruções de iteração* ou repetição, que serão estudadas até ao final deste capítulo.

O conhecimento da sintaxe e da semântica das instruções de iteração do C++ não é suficiente para o desenvolvimento de bom código que as utilize. Por um lado, o desenvolvimento de bom código obriga à demonstração formal ou informal do seu correcto funcionamento. Por outro lado, o desenvolvimento de ciclos não é uma tarefa fácil, sobretudo para o principiante. Assim, nas próximas secções apresenta-se um método de demonstração da correcção de ciclos e faz-se uma pequena introdução à metodologia de Dijkstra, que fundamenta e disciplina a construção de ciclos.

4.5.1 A instrução de iteração `while`

O `while` é a mais importante das instruções de iteração. A sua sintaxe é simples:

```
while(expressão_booleana)
    instrução_controlada
```

A execução da instrução `while` consiste na execução repetida da instrução *`instrução_controlada`* enquanto *`expressão_booleana`* tiver o valor lógico verdadeiro. A *`expressão_booleana`* é sempre calculada antes da instrução *`instrução_controlada`*. Assim, se a expressão for inicialmente falsa, a instrução *`instrução_controlada`* não chega a ser executada, passando o fluxo de execução para a instrução subsequente ao `while`. A instrução *`instrução_controlada`* pode consistir em qualquer instrução do C++, e.g., um bloco de instruções ou um outro ciclo.

À expressão booleana de controlo do `while` chama-se a *guarda* do ciclo (representada muitas vezes por G), enquanto à instrução controlada se chama *passo*. Assim, o passo é repetido enquanto a guarda for verdadeira. Ou seja

```
while(G)
    passo
```

A execução da instrução de iteração `while` pode ser representada pelo diagrama de actividade na Figura 4.4.

Exemplo

O procedimento que se segue escreve no ecrã uma linha com todos os números inteiros de zero a `n` (*exclusive*), sendo `n` o seu único parâmetro (não se colocaram instruções de asserção para simplificar):

```
/** Escreve inteiros de 0 a n (exclusive) no ecrã.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ ecrã contém, para além do que continha originalmente, os
               inteiros entre 0 e n exclusive, em representação decimal. */
```

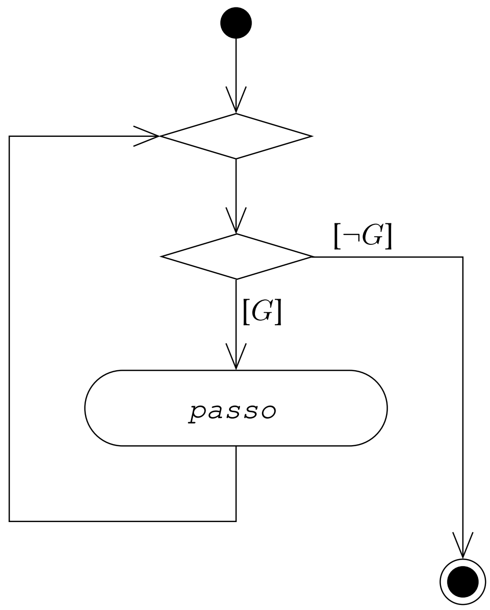

Figura 4.4: Diagrama de actividade da instrução de iteração `while`.

```
void escreveInteiros(int const n) // 1
{                                 // 2
    int i = 0;                    // 3
    while(i != n) {               // 4
        cout << i << ' ';         // 5
        ++i;                      // 6
    }                             // 7
    cout << endl;                 // 8
}                                 // 9
```

O diagrama de actividade correspondente encontra-se na Figura 4.5.

Seguindo o diagrama, começa por se construir a variável `i` com valor inicial 0 (linha 3) e em seguida executa-se o ciclo `while` que:

1. Verifica se a guarda $\texttt{i} \neq \texttt{n}$ é verdadeira (linha 4).
2. Se a guarda for verdadeira, executa as instruções nas linhas 5 e 6, voltando depois a verificar a guarda (volta a 1.).
3. Se a guarda for falsa, o ciclo termina.

Depois de terminado o ciclo, escreve-se um fim-de-linha (linha 8) e o procedimento termina.

4.5.2 Variantes do ciclo `while`: `for` e `do while`

A maior parte dos ciclos usando a instrução `while` têm a forma

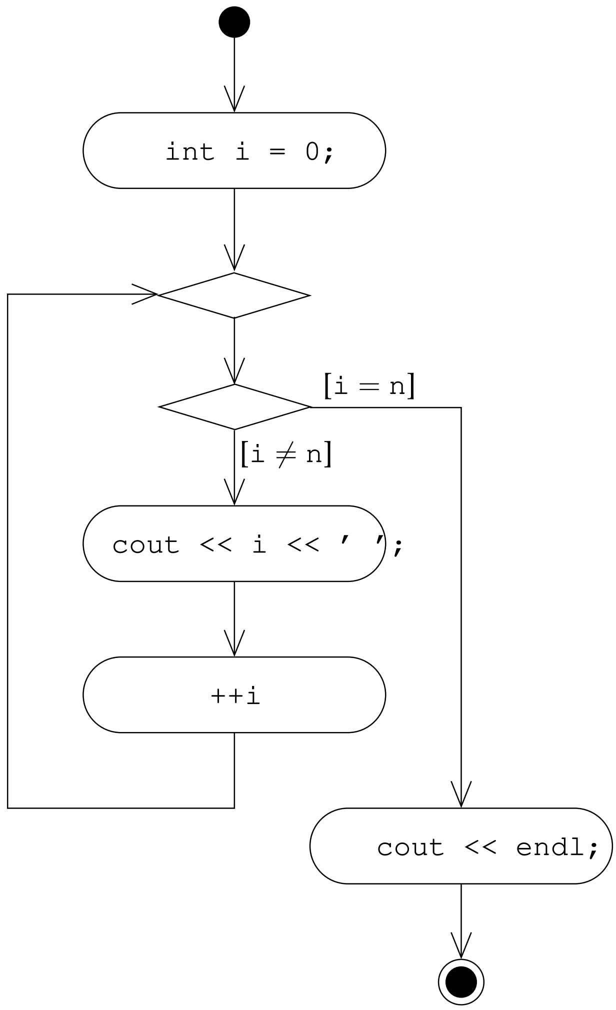

Figura 4.5: Diagrama de actividade do procedimento `escreveInteiros()`.

```
inic
while(G) {
    acção
    prog
}
```

onde `inic` são as instruções de *inicialização* que preparam as variáveis envolvidas no ciclo, `prog` são instruções correspondentes ao chamado *progresso* do ciclo que garantem que o ciclo termina em tempo finito, e `acção` são instruções correspondentes à chamada *acção* do ciclo. Nestes ciclos, portanto, o passo subdivide-se em acção e progresso, ver Figura 4.6.

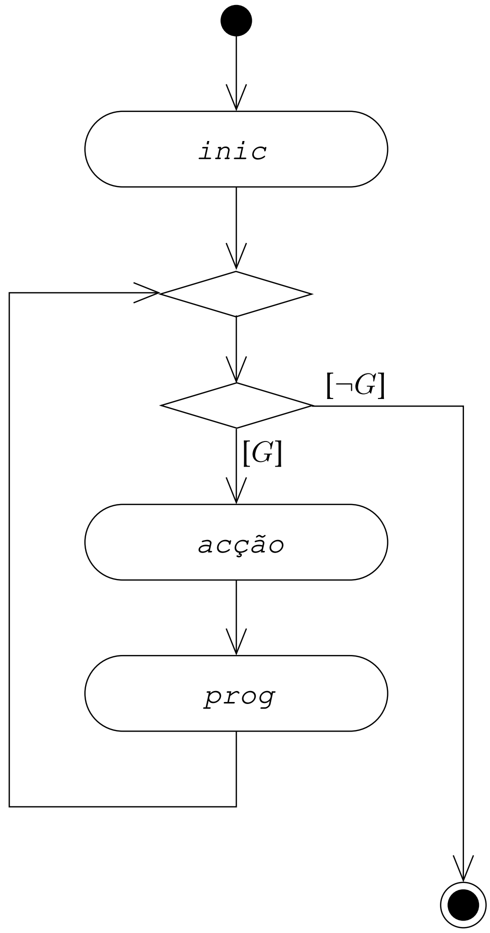

Figura 4.6: Diagrama de actividade de um `ciclo` típico.

Existe uma outra instrução de iteração, a instrução `for`, que permite abreviar este tipo de ciclos:

```
for(inic; G; prog)
    acção
```

em que *inic* e *prog* têm de ser expressões, embora *inic* possa definir também uma variável locais.

O procedimento `escreveInteiros()` já desenvolvido pode ser reescrito como

```
/** Escreve inteiros de 0 a n (exclusive) no ecrã.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ ecrã contém, para além do que continha originalmente, os
              inteiros entre 0 e n exclusive, em representação decimal. */
void escreveInteiros(int const n) // 1
{                                 // 2
    for(int i = 0; i != n; ++i)   // 3
        cout << i << ' ';         // 4
    cout << endl;                 // 5
}                                 // 6
```

Ao converter o `while` num `for` aproveitou-se para restringir ainda mais a visibilidade da variável `i`. Esta variável, estando definida no *cabeçalho* do `for`, só é visível nessa instrução (linhas 3 e 4). Relembra-se que é de toda a conveniência que a visibilidade das variáveis seja sempre o mais restrita possível à zona onde de facto são necessárias.

Qualquer das expressões no cabeçalho de uma instrução `for` (*inic*, G e *prog*) pode ser omitida. A omissão da guarda G é equivalente à utilização de uma guarda sempre verdadeira, gerando por isso um ciclo infinito, ou laço (*loop*). Ou seja, escrever

```
for(inic; ; prog)
    acção
```

é o mesmo que escrever

```
for(inic; true; prog)
    acção
```

ou mesmo

```
inic;
while(true) {
    acção
    prog
}
```

cujo diagrama de actividade se vê na Figura 4.7.

No entanto, os ciclos infinitos só são verdadeiramente úteis se o seu passo contiver alguma instrução `return` ou `break` (ver Secção 4.5.4) que obrigue o ciclo a terminar alguma vez (nesse caso, naturalmente, deixam de ser infinitos...).

Um outro tipo de ciclo corresponde à instrução `do while`. A sintaxe desta instrução é:

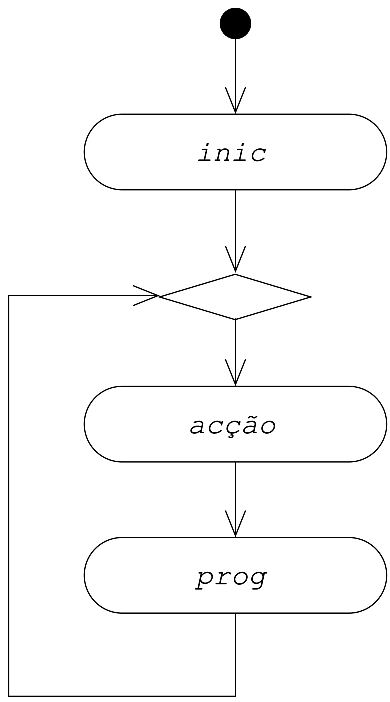

Figura 4.7: Diagrama de actividade de um laço.

```
do
    instrução_controlada
while(expressão_booleana);
```

A execução da instrução `do while` consiste na execução repetida de *`instrução_controlada`* enquanto *`expressão_booleana`* tiver o valor verdadeiro. Mas, ao contrário do que se passa no caso da instrução `while`, *`expressão_booleana`* é sempre calculada e verificada *depois* da instrução *`instrução_controlada`*. Quando o resultado é falso o fluxo de execução passa para a instrução subsequente ao `do while`. Note-se que *`instrução_controlada`* pode consistir em qualquer instrução do C++, e.g., um bloco de instruções.

Tal como no caso da instrução `while`, à instrução controlada pela instrução `do while` chama-se *passo* e à condição que a controla chama-se *guarda*. A execução da instrução de iteração `do while` pode ser representada pelo diagrama de actividade na Figura 4.8.

Ao contrário do que se passa com a instrução `while`, durante a execução de uma instrução `do while` o passo é executado sempre pelo menos uma vez.

Exemplo com `for`

O ciclo `for` é usado frequentemente para repetições ou contagens simples. Por exemplo, se se pretender escrever no ecrã os inteiros de 0 a 9 (um por linha), pode-se usar o seguinte código:

```
for(int i = 0; i != 10; ++i)
    cout << i << endl;
```

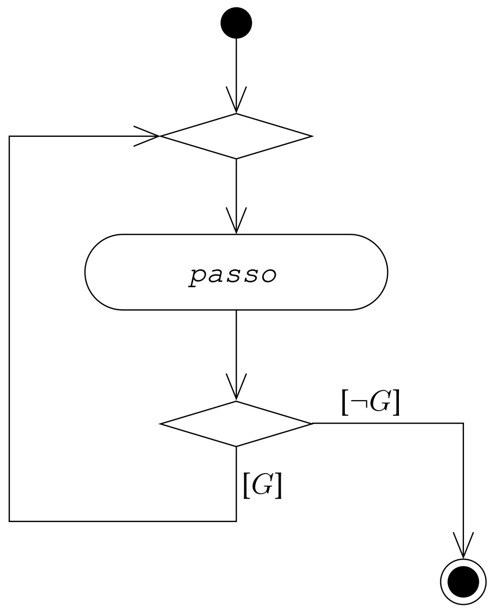

Figura 4.8: Diagrama de actividade da instrução de iteração `do while`.

Se se pretender escrever no ecrã uma linha com 10 asteriscos, pode-se usar o seguinte código:

```
for(int i = 0; i != 10; ++i)
    cout << '*';
cout << endl; // para terminar a linha.
```

É importante observar que, em C++, é típico começar as contagens em zero e usar como guarda o total de repetições a efectuar. Nos ciclos acima a variável `i` toma todos os valores entre 0 e 10 *inclusive*, embora para `i` $= 10$ a acção do ciclo não chegue a ser executada. Alterando o primeiro ciclo de modo a que a definição da variável `i` seja feita fora do ciclo e o seu valor no final do ciclo mostrado no ecrã

```
int i = 0;
for(; i != 10; ++i)
    cout << i << endl;
cout << "O valor final é " << i << '.' << endl;
```

então a execução deste troço de código resultaria em

```
0
1
2
3
4
5
```

```
6
7
8
9
O valor final é 10.
```

Exemplo com `do while`

É muito comum usar-se o ciclo `do while` quando se pretende validar uma entrada de dados por um utilizador do programa.

Suponha-se que se pretende que o utilizador introduza um número inteiro entre 0 e 100 *inclusive*). Se se pretender obrigá-lo à repetição da entrada de dados até que introduza um valor nas condições indicadas, pode-se usar o seguinte código:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    cout << "Introduza valor (0 a 100): ";
    cin >> n;
    while(n < 0 or 100 < n) {
        cout << "Valor incorrecto! << endl;
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
    }

    assert(0 <= n and n <= 100);
}
```

A instrução `cin >> n` e o pedido de um valor aparecem duas vezes, o que não parece uma solução muito elegante. Isto deve-se a que, antes de entrar no ciclo para a primeira iteração, tem de fazer sentido verificar se `n` está ou não dentro dos limites impostos, ou seja, a variável `n` tem de ter um valor que não seja arbitrário. A alternativa usando o ciclo `do while` seria:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(n < 0 or 100 < n)
```

```
        cout << "Valor incorrecto!" << endl;
    } while(n < 0 or 100 < n);

    assert(0 <= n and n <= 100);
}
```

Este ciclo executa o bloco de instruções controlado pelo menos uma vez, dado que a guarda só é avaliada depois da primeira execução. Assim, só é necessária uma instrução `cin >> n` e um pedido de valor. A contrapartida é a necessidade da instrução alternativa `if` dentro do ciclo, com a consequente repetição da guarda... Mais tarde se verão formas alternativas de escrever este ciclo.

Em geral os ciclos `while` e `for` são suficientes, sendo muito raras as ocasiões em que a utilização do ciclo `do while` resulta em código realmente mais claro. No entanto, é má ideia resolver o problema atribuindo um valor inicial à variável `n` que garanta que a guarda é inicialmente verdadeira, de modo a conseguir utilizar o ciclo `while`:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    // Truque sujo: inicialização para a guarda ser inicialmente verdadeira. Má ideia!
    n = -1;
    while(n < 0 or 100 < n) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
    }

    assert(0 <= n and n <= 100);
}
```

Quando se tiver de inicializar uma variável de modo a que o passo de um ciclo seja executado pelo menos uma vez, é melhor recorrer a um ciclo `do while`.

Equivalências entre instruções de iteração

É sempre possível converter um ciclo de modo a usar qualquer das instruções de iteração, como indicado na Tabela 4.1. No entanto, a maior parte dos problemas resolvem-se de um modo mais óbvio e mais legível com uma destas instruções do que com as outras.

Tabela 4.1: Equivalências entre ciclos. Estas equivalências não são verdadeiras se as instruções controladas incluírem as instruções `return`, `break`, `continue`, ou `goto` (ver Secção 4.5.4). Há diferenças também quanto ao âmbito das variáveis definidas nestas instruções.

`{` *inic* `while(`G`)` `{` *acção* *prog* `}` `}`	é equivalente a	`for(`*inic*`;` G`;` *prog*`)` *acção*
`{` *inic* `while(`G`)` *passo*	é equivalente a	`for(`*inic*`;` G`;)` *passo*
 `inic` `while(`G`)` *passo*	é equivalente a	// Má ideia... `inic` `if(`G`)` `do` `passo` `while(`G`);`
`do` *passo* `while(`G`)`	é equivalente a	`{` *passo* `while(`G`)` *passo*

4.5.3 Exemplo simples

A título de exemplo de utilização simultânea de instruções de iteração e de selecção e de instruções de iteração encadeadas, segue-se um programa que escreve no ecrã um triângulo rectângulo de asteriscos com o interior vazio:

```
#include <iostream>

using namespace std;

/** Escreve um triângulo oco de asteriscos com a altura passada como argumento.
    @pre PC ≡ 0 ≤ altura.
    @post CO ≡ ecrã contém, para além do que continha originalmente,
              altura linhas adicionais representando um triângulo
              rectângulo oco de asteriscos. */
void escreveTriânguloOco(int const altura)
{
    assert(0 <= altura);

    for(int i = 0; i != altura; ++i) {
        for(int j = 0; j != i + 1; ++j)
            if(j == 0 or j == i or i == altura - 1)
                cout << '*';
            else
                cout << ' ';
        cout << endl;
    }
}

int main()
{
    cout << "Introduza a altura do triângulo: ";
    int altura;
    cin >> altura;
    escreveTriânguloOco(altura);
}
```

Sugere-se que o leitor faça o traçado deste programa e que o compile e execute em modo de depuração para compreender bem os dois ciclos encadeados.

4.5.4 `return`, `break`, `continue`, e `goto` em ciclos

Se um ciclo estiver dentro de uma rotina e se pretender retornar (sair da rotina) a meio do ciclo, então pode-se usar a instrução de retorno. Por exemplo, se se pretender escrever uma função que devolva verdadeiro caso o seu parâmetro (inteiro) seja primo e falso no caso contrário, ver-se-á mais tarde que uma possibilidade é (ver Secção 4.7.5):

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre   PC ≡ 0 ≤ n.
    @post  CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n).  */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n <= 1)
        return false;
    for(int i = 2; i != n; ++i)
        if(n % i == 0)
            // Se se encontrou um divisor ≥ 2 e < n, então n não é primo e pode-se
            // retornar imediatamente:
            return false;
    return true;
}
```

Este tipo de terminação abrupta de ciclos pode ser muito útil, mas também pode contribuir para tornar difícil a compreensão dos programas. Deve portanto ser usado com precaução e parcimónia e apenas em rotinas muito curtas. Noutros casos torna-se preferível usar técnicas de reforço das guardas do ciclo (ver também Secção 4.7.5).

As instruções de `break`, `continue`, e `goto` oferecem outras formas de alterar o funcionamento normal dos ciclos. A sintaxe da última encontra-se em qualquer livro sobre o C++ (e.g., [12]), e não será explicada aqui, desaconselhando-se vivamente a sua utilização.

A instrução `break` serve para terminar abruptamente a instrução `while`, `for`, `do while`, ou `switch` mais interior dentro da qual se encontre. Ou seja, se existirem duas dessas instruções encadeadas, uma instrução `break` termina apenas a instrução interior. A execução continua na instrução subsequente à instrução interrompida.

A instrução `continue` é semelhante à instrução `break`, mas serve para começar antecipadamente a próxima iteração do ciclo (apenas no caso das instruções `while`, `for` e `do while`). Desaconselha-se vivamente a utilização desta instrução.

À instrução `break` aplica-se a recomendação feita quanto à utilização da instrução `return` dentro de ciclos: usar pouco e com cuidado. No entanto, ver-se-á no próximo exemplo que uma utilização adequada das instruções `return` e `break` pode conduzir código simples e elegante.

Exemplo

Considerem-se de novo os dois ciclos alternativos para validar uma entrada de dados por um utilizador:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
```

```
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    cout << "Introduza valor (0 a 100): ";
    cin >> n;
    while(n < 0 or 100 < n) {
        cout << "Valor incorrecto!" << endl;
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
    }

    assert(0 <= n and n <= 100);
}
```

e

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
    } while(n < 0 or 100 < n);

    assert(0 <= n and n <= 100);
}
```

Nenhuma é completamente satisfatória. A primeira porque obriga à repetição da instrução de leitura do valor, que portanto aparece antes da instrução `while` e no seu passo. A segunda porque obriga a uma instrução de selecção cuja guarda é idêntica à guarda do ciclo. O problema está em que teste da guarda deveria ser feito não antes do passo (como na instrução `while`), nem depois do passo (como na instrução `do while`), mas dentro do passo! Ou seja, negando a guarda da instrução de selecção e usando uma instrução `break`,

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
```

```
        cin >> n;
        if(0 <= n and n <= 100)
            break;
        cout << "Valor incorrecto!" << endl;
    } while(n < 0 or 100 < n);

    assert(0 <= n and n <= 100);
}
```

que, como a guarda do ciclo é sempre verdadeira quando é verificada (quando `n` é válido o ciclo termina na instrução `break`, sem se chegar a testar a guarda), é equivalente a

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(0 <= n and n <= 100)
            break;
        cout << "Valor incorrecto!" << endl;
    } while(true);

    assert(0 <= n and n <= 100);
}
```

que é mais comum escrever como

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(0 <= n and n <= 100)
            break;
        cout << "Valor incorrecto!" << endl;
    }

    assert(0 <= n and n <= 100);

    return n;
}
```

A Figura 4.9 mostra o diagrama de actividade da função.

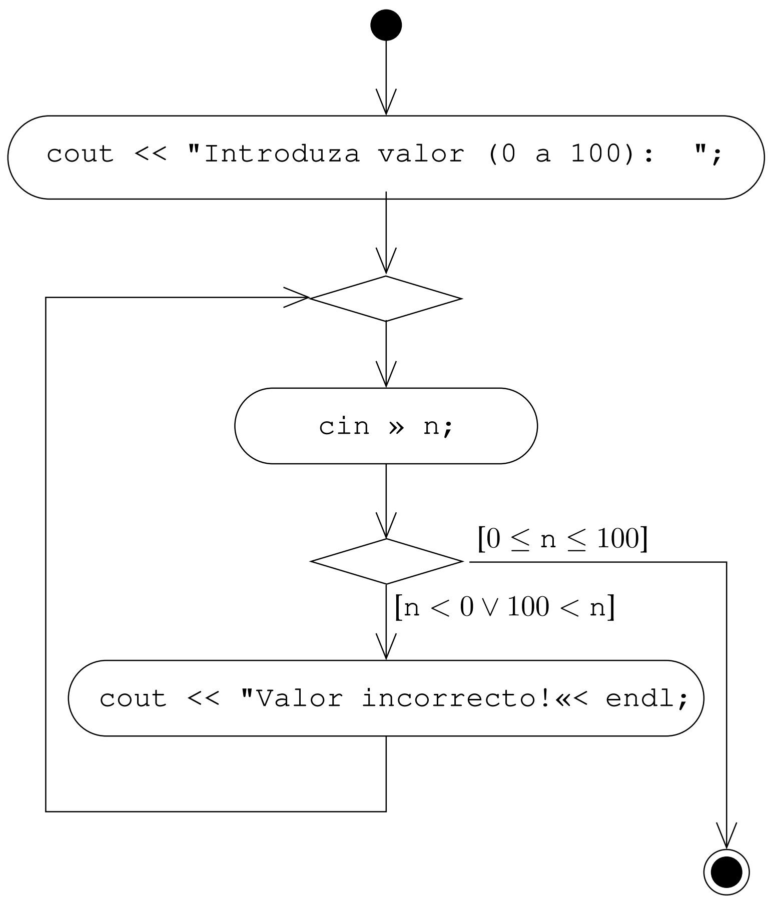

Figura 4.9: Diagrama de actividade da função `inteiroPedidoDe0a100()`.

Os ciclos `do while(true)`, `for(;;)`, e `while(true)` são ciclos infinitos ou laços, que só terminam com recurso a uma instrução `break` ou `return`. Não há nada de errado em ciclos desta forma, desde que recorram a uma e uma só instrução de terminação do ciclo. Respeitando esta restrição, um laço pode ser analisado e a sua correcção demonstrada recorrendo à noção de invariante, como usual.

Uma última versão do ciclo poderia ser escrita recorrendo a uma instrução de retorno, desde que se transformasse o procedimento numa função:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        int n;
```

```
        cin >> n;
        if(0 <= n and n <= 100) {
            assert(0 <= n and n <= 100);
            return n;
        }
        cout << "Valor incorrecto!" << endl;
    }
}
```

Note-se que se passou a definição da variável `n` para o mais perto possível da sua primeira utilização[6].

Esta última versão, no entanto, não é muito recomendável, pois a função tem efeitos laterais: o canal `cin` sofre alterações durante a sua execução. Em geral é má ideia desenvolver mistos de funções e procedimentos, i.e., funções com efeitos laterais ou, o que é o mesmo, procedimentos que devolvem valores.

Nesta altura é conveniente fazer uma pequena digressão e explicar como se pode no procedimento `lêInteiroPedidoDe0a100()` lidar não apenas com erros do utilizador quanto ao valor do inteiro introduzido, mas também com erros mais graves, como a introdução de uma letra em vez de uma sequência de dígitos.

Uma versão "à prova de bala"

Que acontece no procedimento `lêInteiroPedidoDe0a100()` quando o utilizador introduz dados errados, i.e., dados que não podem ser interpretados como um valor inteiro? Infelizmente, o ciclo torna-se infinito, repetindo a mensagem

[6]Noutras linguagens existe uma instrução `loop` para este efeito, que pode ser simulada em C++ recorrendo ao pré-processador (ver Secção 9.2.1):

```
#define loop while(true)
```

Depois desta definição o ciclo pode-se escrever:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    loop {
        cout << "Introduza valor (0 a 100): ";
        int n;
        cin >> n;
        if(0 <= n and n <= 100) {
            assert(0 <= n and n <= 100);
            return n;
        }
        cout << "Valor incorrecto!" << endl;
    }
}
```

```
Introduza valor (0 a 100): Valor incorrecto!
```

eternamente. Isso deve-se ao facto de o canal de entrada `cin` de onde se faz a extracção do valor inteiro, ficar em estado de erro. Uma característica interessante de um canal em estado de erro é que qualquer tentativa de extracção subsequente falhará! Assim, para resolver o problema é necessário limpar explicitamente essa condição de erro usando a instrução

```
cin.clear();
```

que corresponde à invocação de uma operação `clear()` do tipo `istream`, ao qual `cin` pertence (o significado de "operação" neste contexto será visto no Capítulo 7).

Assim, o código original pode ser modificado para

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(not cin) {
            cout << "Isso não é um inteiro!" << endl;
            cin.clear();
        } else if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
        else
            break;
    }

    assert(0 <= n and n <= 100);

    return n;
}
```

onde se tirou partido do facto de um canal poder ser interpretado como um valor booleano, correspondendo o valor falso a um canal em estado de erro.

O problema do código acima é que... fica tudo na mesma... Isto acontece porque o caractere erróneo detectado pela operação de extracção mantém-se no canal `cin` apesar de se ter limpo a condição de erro, logo a próxima extracção tem de forçosamente falhar, tal como a primeira, e assim sucessivamente. A solução passa por remover os caracteres erróneos do canal `cin` sempre que se detectar um erro. A melhor forma de o fazer é eliminar toda a linha introduzida pelo utilizador: é mais simples e mais intuitivo para o utilizador do programa.

Para eliminar toda a linha, basta extrair do canal caracteres até se encontrar o caractere que representa o fim-de-linha, i.e., `'\n'`. Para que a extracção seja feita para cada caractere individualmente, coisa que não acontece com o operador `>>`, que por omissão ignora todos os espaços em branco que encontra[7], usa-se a operação `get()` da classe `istream`:

```
/** Extrai todos os caracteres do canal cin até encontrar o fim-de-linha.
    @pre PC ≡ V.
    @post CO ≡Todos os caracteres no canal cin até ao primeiro fim-de-linha
              inclusiva foram extraídos. */
void ignoraLinha()
{
    char caractere;
    do
        cin.get(caractere);
    while(cin and caractere != '\n')
}

/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(not cin) {
            cout << "Isso não é um inteiro!" << endl;
            cin.clear();
            ignoraLinha();
        } else if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
        else
            break;
    }

    assert(0 <= n and n <= 100);

    return n;
}
```

4.5.5 Problemas comuns

De longe o problema mais comum ao escrever ciclos é o “falhanço por um” (*off by one*). Por exemplo, quando se desenvolve um ciclo `for` para escrever 10 asteriscos no ecrã, é comum

[7] Espaços em branco são os caracteres espaço, tabulador, tabulador vertical e fim-de-linha.

errar a guarda do ciclo e escrever

```
for(int i = 0; i <= 10; ++i)
    cout << '*';
```

que escreve um asterisco a mais. É necessário ter cuidado com a guarda dos ciclos, pois estes erros são comuns e muito difíceis de detectar. Na Secção 4.7 estudar-se-ão metodologias de desenvolvimento de ciclos que minimizam grandemente a probabilidade de estes erros ocorrerem.

Outro problema comum corresponde a colocar um ; após o cabeçalho de um ciclo. Por exemplo:

```
int i = 0;
while(i != 10); // Isto é uma instrução nula!
{
    cout << '*';
    ++i;
}
```

Neste caso o ciclo nunca termina, pois o passo do `while` é a instrução nula, que naturalmente não afecta o valor de `i`. Um caso pior ocorre quando se usa um ciclo `for`:

```
for(int i = 0; i != 10; ++i);
    cout << '*';
```

Este caso é mais grave porque o ciclo termina[8]. Mas, ao contrário do que o programador pretendia, este pedaço de código escreve apenas um asterisco, e não 10!

4.6 Asserções com quantificadores

A especificação de de um problema sem quaisquer ambiguidades é, como se viu, o primeiro passo a realizar para a sua solução. A especificação de um problema faz-se tipicamente indicando a pré-condição (PC) e a condição objectivo (CO). A pré-condição é um predicado acerca das variáveis do problema que se assume ser verdadeiro no início. A condição objectivo é um predicado que se pretende que seja verdadeiro depois de resolvido o problema, i.e., depois de executado o troço de código que o resolve. A pré-condição e a condição objectivo não passam, como se viu antes, de asserções ou afirmações feitas acerca das variáveis de um programa, i.e., acerca do seu estado, sendo o estado de um programa dado pelo valor das suas variáveis em determinado instante de tempo. Assim, a pré-condição estabelece limites ao estado do programa *imediatamente antes* de começar a sua resolução e a condição objectivo estabelece limites ao estado do programa *imediatamente depois* de terminar a sua resolução.

[8]Se não terminar é mais fácil perceber que alguma coisa está errada no código!

A escrita de asserções para problemas um pouco mais complicados que os vistos até aqui requer a utilização de quantificadores. Quantificadores são formas matemáticas abreviadas de escrever expressões que envolvem a repetição de uma dada operação. Exemplos são os quantificadores aritméticos somatório e produto, e os quantificadores lógicos universal ("qualquer que seja") e existencial ("existe pelo menos um").

Apresentam-se aqui algumas notas sobre os quantificadores que são mais úteis para a construção de asserções acerca do estado dos programas. A notação utilizada encontra-se resumida no Apêndice A.

Os quantificadores terão sempre a forma

$$(\mathbf{X}\, v \,:\, \text{predicado}(v) \,:\, \text{expressão}(v))$$

onde

X indica a operação realizada: **S** para a soma (somatório), **P** para o produto ("produtório" ou "piatório"), **Q** para a conjunção ("qualquer que seja"), e **E** para a disjunção ("existe pelo menos um").

v é uma variável muda, que tem significado apenas dentro do quantificador, e que se assume normalmente pertencer ao conjunto dos inteiros. Se pertencer a outro conjunto tal pode ser indicado explicitamente. Por exemplo:

$$\left(\mathbf{Q}\, x \in \mathbb{R} \,:\, \sqrt{2} \leq |x| \,:\, 0 \leq x^2 - 2\right).$$

Um quantificador pode possuir mais do que uma variável muda. Por exemplo:

$$(\mathbf{S}\, i, j \,:\, 0 \leq i < m \wedge 0 \leq j < n \,:\, f(i, j))$$

$\text{predicado}(v)$ é um predicado envolvendo a variável muda v e que define implicitamente o conjunto de valores de v para os quais deve ser realizada a operação.

$\text{expressão}(v)$ é uma expressão envolvendo a variável muda v e que deve ter um resultado aritmético no caso dos quantificadores aritméticos e lógico no caso dos quantificadores lógicos. I.e., no caso dos quantificadores lógicos "qualquer que seja" e "existe pelo menos um", essa expressão deve ser também um predicado.

4.6.1 Somas

O quantificador soma corresponde ao usual somatório. Na notação utilizada,

$$(\mathbf{S}\, j \,:\, m \leq j < n \,:\, \text{expressão}(j))$$

tem exactamente o mesmo significado que o somatório de $\text{expressão}(j)$ com j variando entre m e $n-1$ *inclusive*. Numa notação mais clássica escrever-se-ia

$$\sum_{j=m}^{n-1} \text{expressão}(j).$$

Por exemplo, é um facto conhecido que o somatório dos primeiros n inteiros não-negativos é $\frac{n(n-1)}{2}$, ou seja,

$$(\mathbf{S}\, j\ :\ 0 \leq j < n\ :\ j) = \frac{n(n-1)}{2} \text{ se } 0 < n.$$

Que acontece se $n \leq 0$? Neste caso é evidente que a variável muda j não pode tomar quaisquer valores, e portanto o resultado é a soma de zero termos. A soma de zero termos é zero, por ser 0 o elemento neutro da soma. Logo,

$$(\mathbf{S}\, j\ :\ 0 \leq j < n\ :\ j) = 0 \text{ se } n \leq 0.$$

Em geral pode dizer que

$$(\mathbf{S}\, j\ :\ m \leq j < n\ :\ \text{expressão}(j)) = 0 \text{ se } n \leq m.$$

Se se pretender desenvolver uma função que calcule a soma dos `n` primeiros números ímpares positivos (sendo `n` um parâmetro da função), pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo, como habitual:

```
/** Devolve a soma dos primeiros n ímpares positivos.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ somaÍmpares = (S j : 0 ≤ j < n : 2j + 1) */
int somaÍmpares(int const n)
{
    ...
}
```

Mais uma vez fez-se o parâmetro da função constante de modo a deixar claro que a função não lhe altera o valor.

4.6.2 Produtos

O quantificador produto corresponde ao produto de factores por vezes conhecido como "produtório" ou mesmo "piatório". Na notação utilizada,

$$(\mathbf{P}\, j\ :\ 0 \leq j < n\ :\ \text{expressão}(j))$$

tem exactamente o mesmo significado que o usual produto de $\text{expressão}(j)$ com j variando entre m e $n-1$ *inclusive*. Numa notação mais clássica escrever-se-ia

$$\prod_{j=m}^{n-1} \text{expressão}(j).$$

Por exemplo, a definição de factorial é

$$n! = (\mathbf{P}\, j\ :\ 1 \leq j < n+1\ :\ j)\,.$$

O produto de zero termos é um, por ser 1 o elemento neutro da multiplicação. Ou seja,

$$(\mathbf{P}\, j\,:\, m \leq j < n\,:\, \text{expressão}(j)) = 1 \text{ se } n \leq m.$$

Se se pretender desenvolver uma função que calcule o factorial de `n` (sendo `n` um parâmetro da função), pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve o factorial de n.
    @pre   PC ≡ 0 ≤ n.
    @post CO ≡ factorial = n! = (P j : 1 ≤ j < n + 1 : j). */
int factorial(int const n)
{
    ...
}
```

4.6.3 Conjunções e o quantificador universal

O quantificador universal corresponde à conjunção (e) de vários predicados usualmente conhecida por "qualquer que seja"[9]. Na notação utilizada,

$$(\mathbf{Q}\, j\,:\, m \leq j < n\,:\, \text{predicado}(j))$$

tem exactamente o mesmo significado que a conjunção dos predicados $\text{predicado}(j)$ com j variando entre m e $n-1$ *inclusive*. Numa notação mais clássica escrever-se-ia

$$\bigwedge_{j=m}^{n-1} \text{predicado}(j)$$

ou ainda

$$\forall m \leq j < n\,:\, \text{predicado}(j).$$

A conjunção de zero predicados tem valor verdadeiro, por ser $\mathcal{V}$ o elemento neutro da conjunção. Ou seja[10],

$$(\mathbf{Q}\, j\,:\, m \leq j < n\,:\, \text{predicado}(j)) = \mathcal{V} \text{ se } n \leq m.$$

Por exemplo, a definição do predicado $\text{primo}(n)$ que tem valor $\mathcal{V}$ se n é primo e $\mathcal{F}$ no caso contrário, é

$$\text{primo}(n) = \begin{cases} (\mathbf{Q}\, j\,:\, 2 \leq j < n\,:\, n \div j \neq 0) & \text{se } 2 \leq n, \text{ e} \\ \mathcal{F} & \text{se } 0 \leq n < 2, \end{cases}$$

[9] Se não é claro para si que o quantificador universal corresponde a uma sequência de conjunções, pense no significado de escrever "todos os humanos têm cabeça". A tradução para linguagem mais matemática seria "qualquer que seja h pertencente ao conjunto dos humanos, h tem cabeça". Como o conjunto dos humanos é finito, pode-se escrever por extenso, listando todos os possíveis humanos: "o António tem cabeça **e** o Sampaio tem cabeça **e** ...". Isto é, a conjunção de todas as afirmações.

[10] A afirmação "todos os marcianos têm cabeça" é verdadeira, pois não existem marcianos. Esta propriedade é menos intuitiva que no caso dos quantificadores soma e produto, mas é importante.

ou seja,

$$\text{primo}(n) = ((\mathbf{Q}\, j\,:\, 2 \leq j < n\,:\, n \div j \neq 0) \wedge 2 \leq n) \text{ para } 0 \leq n,$$

sendo $\div$ a operação resto da divisão inteira[11].

Se se pretender desenvolver uma função que devolva o valor lógico verdadeiro quando `n` (sendo `n` um parâmetro da função) é um número primo, e falso no caso contrário, pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    ...
}
```

4.6.4 Disjunções e o quantificador existencial

O quantificador existencial corresponde à disjunção (ou) de vários predicados usualmente conhecida por "existe pelo menos um"[12]. Na notação utilizada,

$$(\mathbf{E}\, j\,:\, m \leq j < n\,:\, \text{predicado}(j))$$

tem exactamente o mesmo significado que a disjunção dos predicados $\text{predicado}(j)$ com j variando entre m e $n-1$ *inclusive* e pode-se ler como "existe um valor j entre m e n *exclusive* tal que $\text{predicado}(i)$ é verdadeiro". Numa notação mais clássica escrever-se-ia

$$\bigvee_{j=m}^{n-1} \text{predicado}(j)$$

ou ainda

$$\exists m \leq j < n\,:\, \text{predicado}(j).$$

A disjunção de zero predicados tem valor falso, por ser $\mathcal{F}$ o elemento neutro da disjunção. Ou seja[13],

$$(\mathbf{E}\, j\,:\, m \leq j < n\,:\, \text{predicado}(j)) = \mathcal{F} \text{ se } n \leq m.$$

Este quantificador está estreitamente relacionado com o quantificador universal. É sempre verdade que

$$\neg\left(\mathbf{Q}\, j\,:\, m \leq j < n\,:\, \text{predicado}(j)\right) = \left(\mathbf{E}\, j\,:\, m \leq j < n\,:\, \neg\text{predicado}(j)\right),$$

[11] Como o operador `%` em C++.

[12] Se não é claro para si que o quantificador existencial corresponde a uma sequência de disjunções, pense no significado de escrever "existe pelo menos um humano com cabeça". A tradução para linguagem mais matemática seria "existe pelo menos um h pertencente ao conjunto dos humanos tal que h tem cabeça". Como o conjunto dos humanos é finito, pode-se escrever por extenso, listando todos os possíveis humanos: "o Zé tem cabeça **ou** o Sampaio tem cabeça **ou** ...". Isto é, a disjunção de todas as afirmações.

[13] A afirmação "existe pelo menos um marciano com cabeça" é falsa, pois não existem marcianos.

ou seja, se não é verdade que para qualquer j o predicado $\text{predicado}(j)$ é verdadeiro, então existe pelo menos um j para o qual o predicado $\text{predicado}(j)$ é falso. Aplicado à definição de número primo acima, tem-se

$$\begin{aligned}\neg\text{primo}(n) &= \neg\left((\mathbf{Q}\, j\,:\,2 \leq j < n\,:\,n \div j \neq 0) \wedge 2 \leq n\right)\\ &= \neg\,(\mathbf{Q}\, j\,:\,2 \leq j < n\,:\,n \div j \neq 0) \vee n < 2\\ &= (\mathbf{E}\, j\,:\,2 \leq j < n\,:\,n \div j = 0) \vee n < 2\end{aligned}$$

para $0 \leq n$. I.e., um inteiro não-negativo não é primo se for inferior a dois ou se for divisível por algum inteiro superior a 1 e menor que ele próprio.

Se se pretender desenvolver uma função que devolva o valor lógico verdadeiro quando existir um número primo entre `m` e `n` *exclusive* (sendo `m` e `n` parâmetros da função, com `m` não-negativo), e falso no caso contrário, pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve verdadeiro se só se existir um número primo no intervalo [m, n[.
    @pre  PC ≡ 0 ≤ m.
    @post CO ≡ existePrimoNoIntervalo = (E j : m ≤ j < n : primo(j)). */
bool existePrimoNoIntervalo(int const m, int const n)
{
    ...
}
```

4.6.5 Contagens

O quantificador de contagem

$$(\mathbf{N}\, j\,:\,m \leq j < n\,:\,\text{predicado}(j))$$

tem como valor (inteiro) o número de predicados $\text{predicados}(j)$ verdadeiros para j variando entre m e $n-1$ *inclusive*. Por exemplo,

$$(\mathbf{N}\, j\,:\,1 \leq j < 10\,:\,\text{primo}(j)) = 4,$$

ou seja, "existem quatro primos entre 1 e 10 *exclusive*", é uma afirmação verdadeira.

Este quantificador é extremamente útil quando é necessário especificar condições em que o número de ordem é fundamental. Se se pretender desenvolver uma função que devolva o `n`-ésimo número primo (sendo `n` parâmetro da função), pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve o n-ésimo número primo.
    @pre  PC ≡ 1 ≤ n.
    @post CO ≡ primo(primo) ∧ (N j : 2 ≤ j < primo : primo(j)) = n − 1.
int primo(int const n)
{
    ...
}
```

A condição objectivo afirma que o valor devolvido é um primo e que existem exactamente $\mathtt{n}-1$ primos de valor inferior.

4.6.6 O resto da divisão

Apresentou-se atrás o operador resto da definição inteira $\div$ sem que se tenha definido formalmente. A definição pode ser feita à custa de alguns dos quantificadores apresentados: $m \div n$, com $0 \leq m$ e $0 < n$[14], é o único elemento do conjunto $\{0 \leq r < n : (\mathbf{E}\, q : 0 \leq q : m = qn + r)\}$. Claro que a definição está incompleta enquanto não se demonstrar que de facto o conjunto tem um único elemento, ou seja que, quando $0 \leq m$ e $0 < n$, se tem

$$(\mathbf{N}\, r : 0 \leq r < n : (\mathbf{E}\, q : 0 \leq q : m = qn + r)) = 1.$$

Deixa-se a demonstração como exercício para o leitor.

4.7 Desenvolvimento de ciclos

O desenvolvimento de programas usando ciclos é simultaneamente uma arte [10] e uma ciência [8]. Embora a intuição seja muito importante, muitas vezes é importante usar metodologias mais ou menos formais de desenvolvimento de ciclos que permitam garantir simultaneamente a sua correcção. Embora esta matéria seja formalizada na disciplina de Computação e Algoritmia, apresentam-se aqui os conceitos básicos da metodologia de Dijkstra para o desenvolvimento de ciclos. Para uma apresentação mais completa consultar [8].

Suponha-se que se pretende desenvolver uma função para calcular a potência `n` de `x`, isto é, uma função que, sendo `x` e `n` os seus dois parâmetros, devolva $\mathtt{x}^{\mathtt{n}}$. A sua estrutura básica é

```
double potência(double const x, int const n)
{
    ...
}
```

É claro que $\mathtt{x}^{\mathtt{n}} = \mathtt{x} \times \mathtt{x} \times \cdots \times \mathtt{x}$, ou seja, $\mathtt{x}^{\mathtt{n}}$ pode ser obtido por multiplicação repetida de `x`. Assim, uma solução passa por usar um ciclo que no seu passo faça cada uma dessas multiplicações:

[14] A definição de resto pode ser generalizada para englobar valores negativos de m e n. Em qualquer dos casos tem de existir um quociente q tal que $m = qn + r$. Mas o intervalo onde r se encontra varia:

$$\begin{array}{rll} 0 \leq r < n & \text{se} & 0 \leq m \wedge 0 < n \text{ (neste caso } 0 \leq q) \\ 0 \leq r < -n & \text{se} & 0 \leq m \wedge n < 0 \text{(neste caso } q \leq 0) \\ -n < r \leq 0 & \text{se} & m \leq 0 \wedge 0 < n \text{(neste caso } q \leq 0) \\ n < r \leq 0 & \text{se} & m \leq 0 \wedge n < 0 \text{(neste caso } 0 \leq q) \end{array}$$

```
double potência(double const x, int const n)
{
    int i = 1;     // usada para contar o número de x já incluídos no produto.
    double r = x; // usada para acumular os produtos de x.
    while(i <= n) {
        r *= x; // o mesmo que r = r * x;
        ++i;    // o mesmo que i = i + 1;
    }
    return r;
}
```

Será que o ciclo está correcto? Fazendo um traçado da função admitindo que é chamada com os argumentos 5,0 e 2, verifica-se facilmente que devolve 125,0 e não 25,0! Isso significa que é feita uma multiplicação a mais. Observando com atenção a guarda da instrução de iteração, conclui-se que esta não deveria deixar o contador `i` atingir o valor de `n`. Ou seja, a guarda deveria ser $\texttt{i} < \texttt{n}$ e não $\texttt{i} \leq \texttt{n}$. Corrigindo a função:

```
double potência(double const x, int const n)
{
    int i = 1;
    double r = x;
    while(i < n) {
        r *= x;
        ++i;
    }
    return r;
}
```

Estará o ciclo definitivamente correcto? Fazendo traçados da função admitindo que é chamada com argumentos 5 e 2 e com 5 e 3, facilmente se verifica que devolve respectivamente 25 e 125, pelo que aparenta estar correcta. Mas estará correcta para todos os valores? E se os argumentos forem 5 e 0? Nesse caso a função devolve 5 em vez do valor correcto, que é $5^0 = 1$! Observando com atenção a inicialização do contador e do acumulador dos produtos, conclui-se que estes deveriam ser inicializados com 0 e 1, respectivamente. Corrigindo a função:

```
double potência(double const x, int const n)
{
    int i = 0;
    double r = 1.0;
    while(i < n) {
        r *= x;
        ++i;
    }
    return r;
}
```

Neste momento a função parece estar correcta. Mas que acontece se o expoente for negativo? Fazendo o traçado da função admitindo que é chamada com os argumentos 5 e -1, facilmente se verifica que devolve 1 em vez de $5^{-1} = 0{,}2$!

Os vários problemas que surgiram ao longo desde desenvolvimento atribulado deveram-se a que:

1. O problema não foi bem especificado através da escrita da pré-condição e da condição objectivo. Em particular a pré-condição deveria estabelecer claramente se a função sabe lidar com expoentes negativos ou não.
2. O desenvolvimento foi feito "ao sabor da pena", de uma forma pouco disciplinada.

Retrocedendo um pouco na resolução do problema de escrever a função `potência()`, é fundamental, pelo que se viu, começar por especificar o problema sem ambiguidades. Para isso formalizam-se a pré-condição e a condição objectivo da função. Para simplificar, suponha-se que a função só deve garantir bom funcionamento para expoentes não-negativos:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    int i = 0;
    double r = 1.0;
    while(i < n) {
        r *= x;
        ++i;
    }
    return r;
}
```

É importante verificar agora o que acontece se o programador consumidor da função se enganar e, violando o contrato expresso pela pré-condição e pela condição objectivo, a invocar com um expoente negativo. Como se viu, a função simplesmente devolve um valor errado: 1. Isso acontece porque, sendo `n` negativo e `i` inicializado com 0, a guarda é inicialmente falsa, não sendo o passo do ciclo executado nenhuma vez. É possível enfraquecer a guarda, de modo a que seja falsa em menos circunstâncias e de tal forma que o contrato da função não se modifique: basta alterar a guarda de $i < n$ para $i \neq n$. É óbvio que para valores do expoente não-negativos as duas guardas são equivalentes. Mas para expoentes negativos a nova guarda leva a um ciclo infinito! O contador `i` vai crescendo a partir de zero, afastando-se irremediavelmente do valor negativo de `n`.

Qual das guardas será preferível? A primeira, que em caso de engano por parte do programador consumidor da função devolve um valor errado, ou a segunda, que nesse caso entra num ciclo infinito, não chegando a terminar? A verdade é que é preferível a segunda, pois o programador consumidor mais facilmente se apercebe do erro e o corrige em tempo útil[15]. Com a primeira guarda o problema pode só ser detectado demasiado tarde, quando os resultados errados já causaram danos irremediáveis. Assim, a nova versão da função é

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    int i = 0;
    double r = 1.0;
    while(i != n) {
        r *= x;
        ++i;
    }
    return r;
}
```

onde a guarda é consideravelmente mais fraca do que anteriormente.

Claro está que o ideal é explicitar também a verificação da validade da pré-condição:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    while(i != n) {
        r *= x;
        ++i;
    }
    return r;
}
```

[15] Note-se que na realidade o ciclo não é infinito, pois os `int` são limitados e incrementações sucessivas levarão o valor do contador ao limite superior dos `int`. O que acontece depois depende do compilador, sistema operativo e máquina em que o programa foi compilado e executado. Normalmente o que acontece é que uma incrementação feita ao contador quando o seu valor já atingiu o limite superior dos `int` leva o valor a "dar a volta" aos inteiros, passando para o limite inferior dos `int` (ver Secção 2.3). Incrementações posteriores levarão o contador até zero, pelo que em rigor o ciclo não é infinito... Mas demora muito tempo a executar, pelo menos, o que mantém a validade do argumento usado para justificar a fraqueza das guardas.

Isto resolve o primeiro problema, pois agora o problema está bem especificado através de uma pré-condição e uma condição objectivo. E o segundo problema, do desenvolvimento indisciplinado?

É possível certamente desenvolver código "ao sabor da pena", mas é importante que seja desenvolvido correctamente. Significa isto que, para ciclos mais simples ou conhecidos, o programador desenvolve-os rapidamente, sem grandes preocupações. Mas para ciclos mais complicados o programador deve ter especial atenção à sua correcção. Das duas uma, ou os desenvolve primeiro e depois demonstra a sua correcção, ou usa uma metodologia de desenvolvimento que garanta a sua correcção[16]. As próximas secções lidam com estes dois problemas: o da demonstração de correcção de ciclos e o de metodologias de desenvolvimento de ciclos. Antes de avançar, porém, é fundamental apresentar a noção de invariante um ciclo.

4.7.1 Noção de invariante

Considerando de novo a função desenvolvida, assinalem-se com números todos as transições entre instruções da função:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
```

1: Depois da inicialização do ciclo, i.e., depois da inicialização das variáveis nele envolvidas.

```
    while(i != n) {
```

2: Depois de se verificar que a guarda é verdadeira.

```
        r *= x;
```

3: Depois de acumular mais uma multiplicação de `x` em `r`.

```
        ++i;
```

4: Depois de incrementar o contador do número de `x` já incluídos no produto `r`.

```
    }
```

[16]Em alguns casos a utilização desta metodologia não é prática, uma vez que requer um arsenal considerável de modelos: é o caso de ciclos que envolvam leituras do teclado e/ou escritas no ecrã. Nesses casos a metodologia continua aplicável, como é óbvio, embora seja vulgar que as asserções sejam escritas com menos formalidade.

5: Depois de se verificar que a guarda é falsa, imediatamente antes do retorno.

```
        return r;
    }
```

Suponha-se que a função é invocada com os argumentos 3 e 4. Isto é, suponha-se que quando a função começa a ser executada as constantes `x` e `n` têm os valores 3 e 4. Faça-se um traçado da execução da função anotando o valor das suas variáveis em cada uma das transições assinaladas. Obtém-se a seguinte tabela:

Transição	`i`	`r`	**Comentários**
1	0	1	Como $0 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	0	1	
3	0	3	
4	1	3	Como $1 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	1	3	
3	1	9	
4	2	9	Como $2 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	2	9	
3	2	27	
4	3	27	Como $3 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	3	27	
3	3	81	
4	4	81	Como $4 = 4$, o ciclo termina, passando-se à transição 5.
5	4	81	

É fácil verificar que há uma condição relacionando `x`, `r`, `n` e `i` que se verifica em todos as transições do ciclo com excepção da transição 3, i.e., que se verifica *depois da inicialização*, *antes do passo*, *depois do passo*, e no *final do ciclo*. A relação é $\mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$. Esta relação diz algo razoavelmente óbvio: em cada instante (excepto no Ponto 3) o acumulador possui a potência `i` do valor em `x`, tendo `i` um valor entre 0 e `n`. Esta condição, por ser verdadeira ao longo de todo o ciclo (excepto a meio do passo, no Ponto 3), diz-se uma *invariante* do ciclo.

Representando o ciclo na forma de um diagrama de actividade e colocando as asserções que se sabe serem verdadeiras em cada transição do diagrama, é mais fácil compreender a noção de invariante, como se vê na Figura 4.10.

As condições invariantes são centrais na demonstração da correcção de ciclos e durante o desenvolvimento disciplinado de ciclos, como se verá nas próximas secções.

Em geral, para um ciclo da forma

```
inic
while(G)
    passo
```

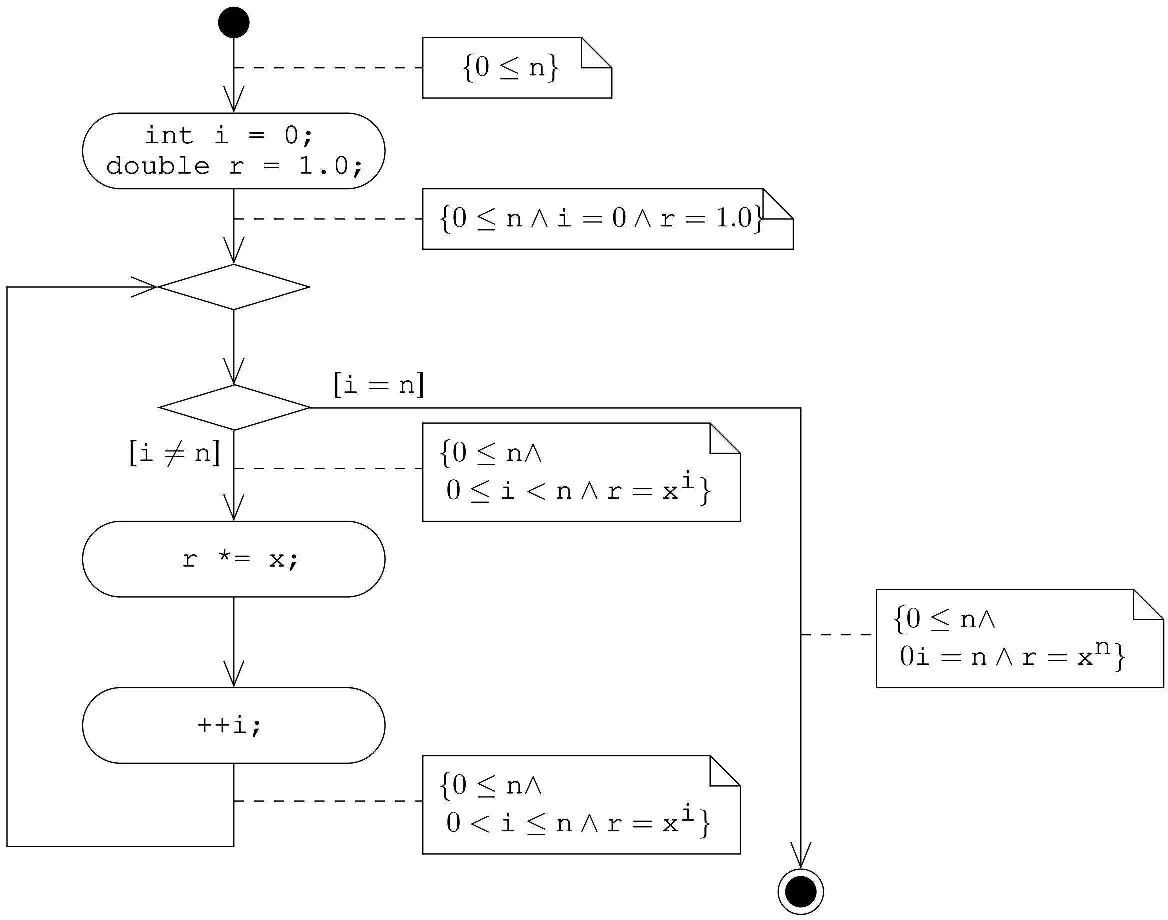

Figura 4.10: Diagrama de actividade do ciclo para cálculo da potência mostrando as asserções nas transições entre instruções (actividades).

uma condição diz-se invariante (CI) se

1. for verdadeira logo após a inicialização do ciclo (`inic`),
2. for verdadeira imediatamente antes do passo (`passo`),
3. for verdadeira imediatamente após o passo e
4. for verdadeira depois de terminado o ciclo.

O diagrama de actividade de um ciclo genérico pode-se ver na Figura 4.11.

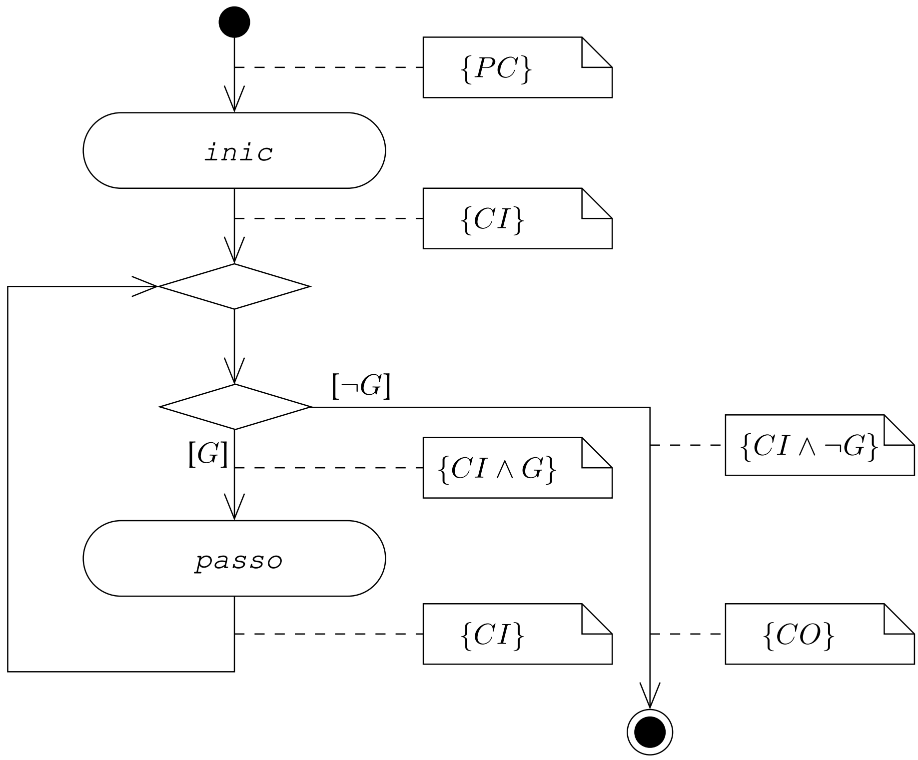

Figura 4.11: Diagrama de actividade de um ciclo genérico. Só as instruções podem alterar o estado do programa e portanto validar ou invalidar asserções.

Incluíram-se algumas asserções adicionais no diagrama. Antes da inicialização assume-se que a pré-condição (PC) do ciclo se verifica, e no final do ciclo pretende-se que se verifique a sua condição objectivo (CO). Além disso, depois da decisão do ciclo, em que se verifica a veracidade da guarda, sabe-se que a guarda (G) é verdadeira antes de executar o passo e falsa (ou seja, verdadeira a sua negação $\neg G$) depois de terminado o ciclo. Ou seja, antes do passo sabe-se que $CI \wedge G$ é uma asserção verdadeira e no final do ciclo sabe-se que $CI \wedge \neg G$ é também uma asserção verdadeira.

No caso da função `potência()`, a condição objectivo do ciclo é diferente da condição objectivo da função, pois refere-se à variável `r`, que será posteriormente devolvida pela função. I.e.:

```
/** Devolve a potência n de x.
    @pre 0 ≤ n.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ 0 ≤ n.
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
    while(i != n) {
        r *= x;
        ++i;
    }
    // CO ≡ r = x^n.
    return r;
}
```

onde se aproveitou para documentar a condição invariante do ciclo.

Demonstração de invariância

Um passo fundamental na demonstração da correcção de um ciclo é a demonstração de invariância de uma dada asserção CI.

Para que a asserção CI possa ser invariante tem de ser verdadeira depois da inicialização (ver Figura 4.11). Assumindo que a pré-condição do ciclo se verifica, então a inicialização *inic* tem de conduzir forçosamente à veracidade da asserção CI. É necessário portanto demonstrar que:

```
// PC
inic
// CI
```

Para o exemplo do cálculo da potência isso é uma tarefa simples. Nesse caso tem-se:

```
// PC ≡ 0 ≤ n.
int i = 0;
double r = 1.0;
// CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
```

A demonstração pode ser feita substituindo em CI os valores iniciais de `r` e `i`:

$$\begin{aligned} CI &\equiv \mathtt{r} = \mathtt{x}^{\mathtt{i}} \land 0 \leq \mathtt{i} \leq \mathtt{n} \\ &\equiv 1 = \mathtt{x}^0 \land 0 \leq 0 \leq \mathtt{n} \\ &\equiv \mathcal{V} \land 0 \leq \mathtt{n} \\ &\equiv 0 \leq \mathtt{n}, \end{aligned}$$

que é garantidamente verdadeira dada a PC, ou seja

$$CI \equiv \mathcal{V}$$

Neste momento demonstrou-se a veracidade da asserção CI depois da inicialização. Falta demonstrar que é verdadeira antes do passo, depois do passo, e no fim do ciclo. A demonstração pode ser feita por indução. Supõe-se que a asserção CI é verdadeira antes do passo numa qualquer iteração do ciclo e demonstra-se que é também verdadeira depois do passo nessa mesma iteração. Como antes do passo a guarda é forçosamente verdadeira, pois de outra forma o ciclo teria terminado, é necessário demonstrar que:

```
// CI ∧ G
passo
// CI
```

Se esta demonstração for possível, então por indução a asserção CI será verdadeira ao longo de todo o ciclo, i.e., antes e depois do passo em qualquer iteração do ciclo e no final do ciclo. Isso pode ser visto claramente observando o diagrama na Figura 4.11. Como se demonstrou que a asserção CI é verdadeira depois da inicialização, então também será verdadeira depois de verificada a guarda pela primeira vez, visto que a verificação da guarda não altera qualquer variável do programa[17]. Ou seja, se a guarda for verdadeira, a asserção CI será verdadeira antes do passo na primeira iteração do ciclo e, se a guarda for falsa, a asserção CI será verdadeira no final do ciclo. Se a guarda for verdadeira, como se demonstrou que a veracidade de CI antes do passo numa qualquer iteração implica a sua veracidade depois do passo na mesma iteração, conclui-se que, quando se for verificar de novo a guarda do ciclo (ver diagrama) a asserção CI é verdadeira. Pode-se repetir o argumento para a segunda iteração do ciclo e assim sucessivamente.

Para o caso dado, é necessário demonstrar que:

```
// CI ∧ G ≡ r = x^i ∧ 0 ≤ i ≤ n ∧ i ≠ n ≡ r = x^i ∧ 0 ≤ i < n.
r *= x;
++i;
// CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
```

A demonstração pode ser feita verificando qual a pré-condição mais fraca de cada uma das atribuições, do fim para o princípio (ver Secção 4.2.3). Obtém-se:

```
// r = x^(i+1) ∧ 0 ≤ i + 1 ≤ n, ou seja,
// r = x^i × x ∧ −1 ≤ i ≤ n − 1, ou seja,
// r = x^i × x ∧ −1 ≤ i < n.
++i;     // o mesmo que i = i + 1;
// CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
```

[17] Admite-se aqui que a guarda é uma expressão booleana *sem efeitos laterais*.

ou seja, para que a asserção CI se verifique depois da incrementação de `i` é necessário que se verifique a asserção

$$\mathtt{r} = \mathtt{x}^{\mathtt{i}} \times \mathtt{x} \wedge -1 \leq \mathtt{i} < \mathtt{n}$$

antes da incrementação. Aplicando a mesma técnica à atribuição anterior tem-se que:

```
// r × x = x^i × x ∧ −1 ≤ i < n.
r *= x; // o mesmo que r = r * x;
// r = x^i × x ∧ −1 ≤ i < n.
```

Falta portanto demonstrar que

```
// CI ∧ G ≡ r = x^i ∧ 0 ≤ i < n implica
// r × x = x^i × x ∧ −1 ≤ i < n.
```

mas isso verifica-se por mera observação, pois $0 \leq \mathtt{i} \Rightarrow -1 \leq \mathtt{i}$ e $\mathtt{r} = \mathtt{x}^{\mathtt{i}} \Rightarrow \mathtt{r} \times \mathtt{x} = \mathtt{x}^{\mathtt{i}} \times \mathtt{x}$

Em resumo, para provar a invariância de uma asserção CI é necessário:

1. Mostrar que, admitindo a veracidade da pré-condição antes da inicialização, a asserção CI é verdadeira depois da inicialização *inic*. Ou seja:

   ```
   // PC
   inic
   // CI
   ```

2. Mostrar que, se CI for verdadeira no início do passo numa qualquer iteração do ciclo, então também o será depois do passo nessa mesma iteração. Ou seja:

   ```
   // CI ∧ G
   passo
   // CI
   ```

   sendo a demonstração feita, comummente, do fim para o princípio, deduzindo as pré-condições mais fracas de cada instrução do passo.

4.7.2 Correcção de ciclos

Viu-se que a demonstração da correcção de ciclos é muito importante. Mas como fazê-la? Há que demonstrar dois factos: que o ciclo quando termina garante que a condição objectivo se verifica e que o ciclo termina sempre ao fim de um número finito de iterações. Diz-se que se demonstrou a correcção parcial de um ciclo se se demonstrou que a condição objectivo se verifica quando o ciclo termina, assumindo que a pré-condição se verifica no seu início. Diz-se que se demonstrou a correcção total de um ciclo se, para além disso, se demonstrou que o ciclo termina sempre ao fim de um número finito de iterações.

Correcção parcial

A determinação de uma condição invariante CI apropriada para a demonstração é muito importante. É que, como a CI é verdadeira no final do ciclo e a guarda G é falsa, para demonstrar a correcção parcial do ciclo basta mostrar que

$$CI \wedge \neg G \Rightarrow CO.$$

No caso da função `potência()` essa demonstração é simples:

$$\begin{aligned} CI \wedge \neg G &\equiv \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{n} \\ &\equiv \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge \mathtt{i} = \mathtt{n} \\ &\Rightarrow \mathtt{r} = \mathtt{x}^{\mathtt{n}} \equiv CO \end{aligned}$$

Em resumo, para provar a correcção parcial de um ciclo é necessário

1. encontrar uma asserção CI apropriada (a asserção $\mathcal{V}$ é trivialmente invariante de qualquer ciclo e não ajuda nada na demonstração de correcção parcial: é necessário que a CI seja "rica em informação"),
2. demonstrar que essa asserção CI é de facto uma invariante do ciclo (ver secção anterior) e
3. demonstrar que $CI \wedge \neg G \Rightarrow CO$.

Correcção total

A correcção parcial é insuficiente. Suponha-se a seguinte versão da função `potência()`:

```
/** Devolve a potência n de x.
    @pre 0 ≤ n.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ 0 ≤ n.
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
    while(i != n)
        ; // Instrução nula!
    // CO ≡ r = xⁿ.
    return r;
}
```

É fácil demonstrar a correcção parcial deste ciclo[18]! Mas este ciclo não termina nunca excepto quando `n` é 0. De acordo com a definição dada na Secção 1.3, este ciclo não implementa um algoritmo, pois não verifica a propriedade da finitude.

A demonstração formal de terminação de um ciclo ao fim de um número finito de iterações faz-se usando o conceito de *função de limitação* (*bound function*) [8], que não será abordado neste texto. Neste contexto será suficiente a demonstração informal desse facto. Por exemplo, na versão original da função `potência()`

```
/** Devolve a potência n de x.
    @pre 0 ≤ n.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ 0 ≤ n.
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
    while(i != n) {
        r *= x;
        ++i;
    }
    // CO ≡ r = xⁿ.
    return r;
}
```

é evidente que, como a variável `i` começa com o valor zero e é incrementada de uma unidade ao fim de cada passo, fatalmente tem de atingir o valor de `n` (que, pela pré-condição, é não-negativo). Em particular é fácil verificar que o número exacto de iterações necessário para isso acontecer é exactamente `n`.

O passo é, tipicamente, dividido em duas partes: a acção e o progresso. Os ciclos têm tipicamente a seguinte forma:

```
inic
while(G) {
    acção
    prog
}
```

onde o progresso *`prog`* corresponde ao conjunto de instruções que garante a terminação do ciclo e a acção *`acção`* corresponde ao conjunto de instruções que garante a invariância de CI apesar de se ter realizado o progresso.

No caso do ciclo da função `potência()` a acção e o progresso são:

[18] Porquê?

```
r *= x; // acção
++i;    // progresso
```

Resumo

Para demonstrar a correcção total de um ciclo:

```
// PC
inic
while(G) {
    acção
    prog
}
// CO
```

é necessário:

1. encontrar uma asserção CI apropriada;
2. demonstrar que essa asserção CI é de facto uma invariante do ciclo:
   (a) mostrar que, admitindo a veracidade de PC antes da inicialização, a asserção CI é verdadeira depois da inicialização *inic*, ou seja:
   ```
   // PC
   inic
   // CI
   ```
   (b) mostrar que, se CI for verdadeira no início do passo numa qualquer iteração do ciclo, então também o será depois do passo nessa mesma iteração, ou seja:
   ```
   // CI ∧ G
   passo
   // CI
   ```
3. demonstrar que $CI \wedge \neg G \Rightarrow CO$; e
4. demonstrar que o ciclo termina sempre ao fim de um número finito de iterações, para o que se raciocina normalmente em termos da inicialização *inic*, da guarda G e do progresso *prog*.

4.7.3 Melhorando a função `potência()`

Ao se especificar sem ambiguidades o problema que a função `potência()` deveria resolver fez-se uma simplificação: admitiu-se que se deveriam apenas considerar expoentes não-negativos. Como relaxar esta exigência? Acontece que, se o expoente tomar valores negativos, então a base da potência não pode ser nula, sob pena de o resultado ser infinitamente grande. Então, a nova especificação será

```
/** Devolve a potência n de x.
    @pre 0 ≤ n ∨ x ≠ 0.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    ...
}
```

É fácil verificar que a resolução do problema exige o tratamento de dois casos distintos, resolúveis através de uma instrução de selecção, ou, mais simplesmente, através do operador `? :`:

```
/** Devolve a potência n de x.
    @pre 0 ≤ n ∨ x ≠ 0.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    int const exp = n < 0 ? -n : n;
    // PC ≡ 0 ≤ exp.
    int i = 0;
    double r = 1.0;
    // CI ≡ r = x^i ∧ 0 ≤ i ≤ exp.
    while(i != exp) {
        r *= x;
        ++i;
    }
    // CO ≡ r = x^exp.
    return n < 0 ? 1.0 / r : r;
}
```

ou ainda, convertendo para a instrução de iteração `for` e eliminando os comentários,

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n ∨ x ≠ 0.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    int const exp = n < 0 ? -n : n;
    double r = 1.0;
    for(int i = 0; i != exp; ++i)
        r *= x;
    return n < 0 ? 1.0 / r : r;
}
```

Um outra alternativa é, sabendo que $x^n = \frac{1}{x^{-n}}$, usar recursividade:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n ∨ x ≠ 0.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    if(n < 0)
        return 1.0 / potência(x, -n);
    double r = 1.0;
    for(int i = 0; i != n; ++i)
        r *= x;
    return r;
}
```

4.7.4 Metodologia de Dijkstra

A programação deve ser uma actividade orientada pelos objectivos. A metodologia de desenvolvimento de ciclos de Dijkstra [8][5] tenta integrar o desenvolvimento do código com a demonstração da sua correcção, começando naturalmente por prescrever olhar com atenção para a condição objectivo do ciclo. O primeiro passo do desenvolvimento de um ciclo é a determinação de possíveis condições invariantes por observação da condição objectivo, seguindo-se-lhe a determinação da guarda, da inicialização e do passo, normalmente decomposto na acção e no progresso:

1. Especificar o problema sem margem para ambiguidades: definir a pré-condição PC e a condição objectivo CO.

2. Tentar perceber se um ciclo é, de facto, necessário. Este passo é muitas vezes descurado, com consequências infelizes, como se verá.

3. Olhando para a condição objectivo, determinar uma condição invariante CI interessante para o ciclo. A condição invariante escolhida é uma versão enfraquecida da condição objectivo CO.

4. Escolher uma guarda G tal que $CI \wedge \neg G \Rightarrow CO$.

5. Escolher uma inicialização `inic` de modo a garantir a veracidade da condição invariante logo no início do ciclo (assumindo, claro está, que a sua pré-condição PC se verifica).

6. Escolher o passo do ciclo de modo a que a condição invariante se mantenha verdadeira (i.e., de modo a garantir que é de facto invariante). É fundamental que a escolha do passo garanta a terminação do ciclo. Para isso o passo é usualmente dividido num progresso `prog` e numa acção `acção`, sendo o progresso que garante a terminação do ciclo e a acção que garante que, apesar do progresso, a condição invariante CI é de facto invariante.

As próximas secções detalham cada um destes passos para dois exemplos de funções a desenvolver.

Especificação do problema

A pré-condição PC e a condição objectivo CO devem indicar de um modo rigoroso e sem ambiguidade (tanto quanto possível) quais os possíveis estados do programa no início de um pedaço de código e quais os possíveis estados no seu final. Podem-se usar pré-condições e condições objectivo para qualquer pedaço de código, desde uma simples instrução, até um ciclo ou mesmo uma rotina. Pretende-se aqui exemplificar o desenvolvimento de ciclos através da escrita de duas funções. Estas funções têm cada uma uma pré-condição e um condição objectivo, que não são em geral iguais à pré-condição e à condição objectivo do ou dos ciclos que, presumivelmente, elas contêm, embora estejam naturalmente relacionadas.

Função `somaDosQuadrados()` Pretende-se escrever uma função `int somaDosQuadrados(int const n)` que devolva a soma dos quadrados dos primeiros `n` inteiros não-negativos. A sua estrutura pode ser:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = ...;
    ...
    // CO ≡ soma_dos_quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

Repare-se que existem duas condições objectivo diferentes. A primeira faz parte do contracto da função (ver Secção 3.2.4) e indica que valor a função deve devolver. A segunda, que é a que vai ser usada no desenvolvimento de ora em diante, indica que valor deve ter a variável `soma_dos_quadrados` antes da instrução de retorno.

Função `raizInteira()` Pretende-se escrever uma função `int raizInteira(int x)` que, assumindo que `x` é não-negativo, devolva o maior inteiro menor ou igual à raiz quadrada de `x`. Por exemplo, se `x` for 4 devolve 2, que é a raiz exacta de 4, se for 3 devolve 1, pois é o maior inteiro menor ou igual a $\sqrt{3} \approx 1{,}73$ e se for 5 devolve 2, pois é o maior inteiro menor ou igual a $\sqrt{5} \approx 2{,}24$. A estrutura da função pode ser:

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
```

```
    @pre    PC ≡ 0eqx.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)².  */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = ...;
    ...
    // CO ≡ 0 ≤ r ∧ r² ≤ x < (r + 1)².

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

Mais uma vez existem duas condições objectivo diferentes. A primeira faz parte do contracto da função e a segunda, que é a que vai ser usada no desenvolvimento de ora em diante, indica que valor deve ter a variável `r` antes da instrução de retorno. É a relação $\mathtt{x} < (\mathtt{r}+1)^2$ que garante que o valor devolvido é o *maior* inteiro menor ou igual a $\sqrt{\mathtt{x}}$.

Determinando se um ciclo é necessário

A discussão deste passo será feita mais tarde, por motivos que ficarão claros a seu tempo. Para já admite-se que sim, que ambos os problemas devem ser resolvidos usando ciclos.

Determinação da condição invariante e da guarda

A escolha da condição invariante CI do ciclo faz-se sempre olhando para a sua condição objectivo CO. Esta escolha é a fase mais difícil no desenvolvimento de um ciclo. Não existem panaceias para esta dificuldade: é necessário usar de intuição, arte, engenho, experiência, analogias com casos semelhantes, etc. Mas existe uma metodologia, desenvolvida por Edsger Dijkstra [5], que funciona bem para um grande número de casos. A ideia subjacente à metodologia é que a condição invariante se deve obter por enfraquecimento da condição objectivo, ou seja, $CO \Rightarrow CI$. A ideia é que, depois de terminado o ciclo, a condição invariante reforçada pela negação da guarda $\neg G$ (o ciclo termina forçosamente com a guarda falsa) garante que foi atingida a condição objectivo, ou seja, $CI \wedge \neg G \Rightarrow CO$.

Por enfraquecimento da condição objectivo CO entende-se a obtenção de uma condição invariante CI tal que, de entre todas as combinações de valores das variáveis que a tornam verdadeira, contenha todas as combinações de valores de variáveis que tornam verdadeira a CO. Ou seja, o conjunto dos estados do programa que verificam CI deve conter o conjunto dos estados do programa que verificam a CO, que é o mesmo que dizer $CO \Rightarrow CI$.

Apresentar-se-ão aqui apenas dois dos vários possíveis métodos para obter a condição invariante por enfraquecimento da condição objectivo:

1. Substituir uma das constantes presentes em CO por uma variável de programa com limites apropriados, obtendo-se assim a CI. A maior parte das vezes substitui-se uma constante que seja limite de um quantificador. A negação da guarda $\neg G$ é depois escolhida de modo a que $CI \wedge \neg G \Rightarrow CO$, o que normalmente é conseguido escolhendo para $\neg G$ o predicado que afirma que a nova variável tem exactamente o valor da constante que substituiu.

2. Se CO corresponder à conjunção de vários termos, escolher parte deles para constituírem a condição invariante CI e outra parte para constituírem a negação da guarda $\neg G$. Por exemplo, se $CO \equiv C_1 \wedge \cdots \wedge C_m$, então pode-se escolher por exemplo $CI \equiv C_1 \wedge \cdots \wedge C_{m-1}$ e $\neg G \equiv C_m$. Esta selecção conduz, por um lado, a uma condição invariante que é obviamente uma versão enfraquecida da condição objectivo, e por outro lado à verificação trivial da implicação $CI \wedge \neg G \Rightarrow CO$, pois neste caso $CI \wedge \neg G = CO$. A este método chama-se *factorização da condição objectivo*.

Muitas vezes só ao se desenvolver o passo do ciclo se verifica que a condição invariante escolhida não é apropriada. Nesse caso deve-se voltar atrás e procurar uma nova condição invariante mais apropriada.

Substituição de uma constante por uma variável Muitas vezes é possível obter uma condição invariante para o ciclo substituindo uma constante presente na condição objectivo por uma variável de programa introduzida para o efeito. A "constante" pode corresponder a uma variável que não seja suposto ser alterada pelo ciclo, i.e., a uma variável que seja constante apenas do ponto de vista lógico.

Voltando à função `somaDosQuadrados()`, é evidente que o corpo da função pode consistir num ciclo cuja condição objectivo já foi apresentada:

$$CO \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{n}\, :\, j^2\right).$$

A condição invariante do ciclo pode ser obtida substituindo a constante `n` por uma nova variável de programa `i`, com limites apropriados, ficando portanto:

$$CI \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{i}\, :\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n}.$$

Como se escolheram os limites da nova variável? Simples: um dos limites (normalmente o inferior) é o primeiro valor de `i` para o qual o quantificador não tem nenhum termo (não há qualquer valor de j que verifique $0 \leq j < 0$) e o outro limite (normalmente o superior) é a constante substituída.

Esta condição invariante tem um significado claro: a variável `soma_dos_quadrados` contém desde o início ao fim do ciclo a soma dos primeiros `i` inteiros não-negativos e a variável `i` varia entre 0 e `n`.

Claro está que a nova variável tem de ser definida na função:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = ...;
    int i = ...;
    ...
    // CO ≡ soma_dos_quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

Em rigor, a condição invariante escolhida não corresponde simplesmente a uma versão enfraquecida da condição objectivo. Na verdade, a introdução de uma nova variável é feita em duas etapas, que normalmente se omitem.

A primeira etapa obriga a uma reformulação da condição objectivo de modo a reflectir a existência da nova variável, que aumenta a dimensão do espaço de estados do programa: substitui-se a constante pela nova variável, mas força-se também a nova variável a tomar o valor da constante que substituiu, acrescentando para tal uma conjunção à condição objectivo

$$CO' \equiv \texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j \,:\, 0 \leq j < \texttt{i} \,:\, j^2\right) \wedge \texttt{i} = \texttt{n}$$

de modo a garantir que $CO' \Rightarrow CO$.

A segunda etapa corresponde à obtenção da condição invariante por enfraquecimento de CO': o termo que fixa o valor da nova variável é relaxado. A condição invariante obtida é de facto mais fraca que a condição objectivo reformulada CO', pois aceita mais possíveis valores para a variável `soma_dos_quadrados` do que CO', que só aceita o resultado final pretendido[19]. Os valores aceites para essa variável pela condição invariante correspondem a valores intermédios durante a execução do ciclo. Neste caso correspondem a todas as possíveis somas de quadrados de inteiros positivos desde a soma com zero termos até à soma com os `n` termos pretendidos.

A escolha da guarda é simples: para negação da guarda $\neg G$ escolhe-se a conjunção que se acrescentou à condição objectivo para se obter CO'

$$\neg G \equiv \texttt{i} = \texttt{n},$$

[19]Na realidade o enfraquecimento pode ser feito usando a factorização! Para isso basta um pequeno truque na reformulação da condição objectivo de modo a incluir um termo extra:

$$CO' \equiv \texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j \,:\, 0 \leq j < \texttt{i} \,:\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{i} = \texttt{n}.$$

Este novo termo não faz qualquer diferença efectiva em CO', mas permite aplicar facilmente a factorização da condição objectivo:

$$CO' \equiv \overbrace{\texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j \,:\, 0 \leq j < \texttt{i} \,:\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n}}^{CI} \wedge \overbrace{\texttt{i} = \texttt{n}}^{\neg G}.$$

ou seja,

$$G \equiv \texttt{i} \neq \texttt{n}.$$

Dessa forma tem-se forçosamente que $(CI \wedge \neg G) = CO' \Rightarrow CO$, como pretendido. A função neste momento é

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = ...;
    int i = ...;
    // CI ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        passo
    }
    // CO ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

É importante que, pelas razões que se viram mais atrás, a guarda escolhida seja o mais fraca possível. Neste caso pode-se-ia reforçar a guarda relaxando a sua negação para $\neg G \equiv \texttt{n} \leq \texttt{i}$, que corresponde à guarda $G \equiv \texttt{i} < \texttt{n}$ muito mais forte e infelizmente tão comum...

A escolha da guarda pode também ser feita observando que no final do ciclo a guarda será forçosamente falsa e a condição invariante verdadeira (i.e., que $CI \wedge \neg G$), e que se pretende, nessa altura, que a condição objectivo do ciclo seja verdadeira. Ou seja, sabe-se que no final do ciclo

$$\texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j \,:\, 0 \leq j < \texttt{i} \,:\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n} \wedge \neg G$$

e pretende-se que

$$CO \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j \,:\, 0 \leq j < \texttt{n} \,:\, j^2\right).$$

A escolha mais simples da guarda que garante a verificação da condição objectivo é $\neg G \equiv \texttt{i} = \texttt{n}$, ou seja, $G \equiv \texttt{i} \neq \texttt{n}$. Ou seja, esta guarda quando for falsa garante que $\texttt{i} = \texttt{n}$, pelo que sendo a condição invariante verdadeira, também a condição objectivo o será.

Mais uma vez, só se pode confirmar se a escolha da condição invariante foi apropriada depois de completado o desenvolvimento do ciclo. Se o não tiver sido, há que voltar a este passo e tentar de novo. Isso não será necessário neste caso, como se verá.

Factorização da condição objectivo Quando a condição objectivo é composta pela conjunção de várias condições, pode-se muitas vezes utilizar parte delas como negação da guarda $\neg G$ e a parte restante como condição invariante CI.

Voltando à função `raizInteira()`, é evidente que o corpo da função pode consistir num ciclo cuja condição objectivo já foi apresentada:

$$\begin{aligned} CO &\equiv 0 \leq \mathtt{r} \wedge \mathtt{r}^2 \leq \mathtt{x} < (\mathtt{r}+1)^2 \\ &\equiv 0 \leq \mathtt{r} \wedge \mathtt{r}^2 \leq \mathtt{x} \wedge \mathtt{x} < (\mathtt{r}+1)^2 \end{aligned}$$

Escolhendo para negação da guarda o termo $\mathtt{x} < (\mathtt{r}+1)^2$, ou seja, escolhendo

$$G \equiv (\mathtt{r}+1)^2 \leq \mathtt{x}$$

obtém-se para a condição invariante os termos restantes

$$CI \equiv 0 \leq \mathtt{r} \wedge \mathtt{r}^2 \leq \mathtt{x}.$$

que é o mesmo que

$$CI \equiv 0 \leq \mathtt{r} \wedge \mathtt{r} \leq \sqrt{\mathtt{x}}.$$

Esta escolha faz com que $CI \wedge \neg G$ seja igual à condição objectivo CO, pelo que *se o ciclo terminar* termina com o valor correcto.

A condição invariante escolhida tem um significado claro: desde o início ao fim do ciclo que a variável `r` não excede a raiz quadrada de `x`. Além disso, a condição invariante é, mais uma vez, uma versão enfraquecida da condição objectivo, visto que admite um maior número de possíveis valores para a variável `r` do que a condição objectivo, que só admite o resultado final pretendido. A função neste momento é

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre   PC ≡ 0eqx.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = ...;
    // CI ≡ 0 ≤ r ∧ r² ≤ x.
    while((r + 1) * (r + 1) <= x) {
        passo
    }
    // CO ≡ 0 ≤ r ∧ r² ≤ x < (r + 1)².

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

Escolha da inicialização

A inicialização de um ciclo é feita normalmente de modo a, assumindo a veracidade da pré-condição PC, garantir a verificação da condição invariante CI da forma mais simples possível.

Função `somaDosQuadrados()` A condição invariante já determinada é

$$CI \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \le j < \texttt{i}\, :\, j^2\right) \wedge 0 \le \texttt{i} \le \texttt{n}.$$

pelo que a forma mais simples de se inicializar o ciclo é com as instruções

```
int soma_dos_quadrados = 0;
int i = 0;
```

pois nesse caso, por simples substituição,

$$\begin{aligned} CI &\equiv 0 = \left(\mathbf{S}\, j\, :\, 0 \le j < 0\, :\, j^2\right) \wedge 0 \le 0 \le \texttt{n} \\ &\equiv 0 = 0 \wedge 0 \le \texttt{n} \\ &\equiv 0 \le \texttt{n} \end{aligned}$$

que é verdadeira desde que a pré-condição $PC \equiv 0 \le \texttt{n}$ o seja. A função neste momento é

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = 0;
    int i = 0;
    // CI ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        passo
    }
    // CO ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

Função `raizInteira()` A condição invariante já determinada é

$$CI \equiv 0 \le \texttt{r} \wedge \texttt{r}^2 \le \texttt{x}.$$

pelo que a forma mais simples de se inicializar o ciclo é com a instrução

```
int r = 0;
```

pois nesse caso, por simples substituição,

$$\begin{aligned} CI &\equiv 0 \leq 0 \wedge 0 \leq \mathtt{x} \\ &\equiv 0 \leq \mathtt{x} \end{aligned}$$

que é verdadeira desde que a pré-condição $PC \equiv 0eq\mathtt{x}$ o seja. A função neste momento é

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre PC ≡ 0eqx.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= n);

    int r = 0;
    // CI ≡ 0 ≤ r ∧ r² ≤ x.
    while((r + 1) * (r + 1) <= x) {
        passo
    }
    // CO ≡ 0 ≤ r ∧ r² ≤ x < (r + 1)².

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

Determinação do passo: progresso e acção

A construção de um ciclo fica completa depois de determinado o passo, normalmente constituído pela acção *acção* e pelo progresso *prog*. O progresso é escolhido de modo a garantir que o ciclo termina, i.e., de modo a garantir que ao fim de um número finito de repetições do passo a guarda G se torna falsa. A acção é escolhida de modo a garantir a invariância de CI *apesar do progresso*.

Função `somaDosQuadrados()` A guarda já determinada é

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

pelo que o progresso mais simples é a simples incrementação de `i`. Este progresso garante que o ciclo termina (i.e., que a guarda se torna falsa) ao fim de no máximo `n` iterações, uma vez que `i` foi inicializada com 0. Ou seja, o passo do ciclo é

// $CI \wedge G \equiv$ `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n}$, ou seja,
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
`++i;`
// $CI \equiv$ `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

onde se assume a condição invariante como verdadeira antes da acção, se sabe que a guarda é verdadeira nesse mesmo lugar (de outra forma o ciclo teria terminado) e se pretende escolher uma acção de modo a que a condição invariante seja verdadeira também *depois do passo*, ou melhor, *apesar do progresso*. Usando a semântica da operação de atribuição, pode-se deduzir a condição mais fraca antes do progresso de modo a que condição invariante seja verdadeira no final do passo:

// $CI \wedge G \equiv$ `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n}$, ou seja,
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i}+1 \,:\, j^2\right) \wedge 0 \leq \mathtt{i}+1 \leq \mathtt{n}$, ou seja,
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i}+1 \,:\, j^2\right) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.
`++i;` // o mesmo que `i = i + 1;`
// $CI \equiv$ `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Falta pois determinar uma acção tal que

// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i}+1 \,:\, j^2\right) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.

Para simplificar a determinação da acção, pode-se fortalecer um pouco a sua condição objectivo forçando `i` a ser maior do que zero, o que permite extrair o último termo do somatório

// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i} \,:\, j^2\right) + \mathtt{i}^2 \wedge 0 \leq \mathtt{i} < \mathtt{n}$
// `soma_ dos_ quadrados` $= \left(\mathbf{S}\, j \,:\, 0 \leq j < \mathtt{i}+1 \,:\, j^2\right) \wedge -1 \leq \mathtt{i} < \mathtt{n}$

A acção mais simples limita-se a acrescentar $\mathtt{i}^2$ ao valor da variável `soma_dos_quadrados`:

```
soma_dos_quadrados += i * i;
```

pelo que o o ciclo e a função completos são

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = 0;
    int i = 0;
    // CI ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        soma_dos_quadrados += i * i;
        ++i;
    }
    return soma_dos_quadrados;
}
```

ou, substituindo pelo ciclo com a instrução for equivalente,

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = 0;
    // CI ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        soma_dos_quadrados += i * i;
    return soma_dos_quadrados;
}
```

Nas versões completas da função incluiu-se um comentário com a condição invariante CI. Este é um comentário normal, e não de documentação. Os comentários de documentação, após `///` ou entre `/** */`, servem para documentar a interface da função ou procedimento (ou outras entidades), i.e., para dizer claramente o que essas entidades fazem, para que servem ou como se comportam, sempre de um ponto de vista externo. Os comentários normais, pelo contrário, servem para que o leitor do saiba como o código, neste caso a função, funciona. Sendo a condição invariante a mais importante das asserções associadas a um ciclo, é naturalmente candidata a figurar num comentário na versão final das funções ou procedimentos.

Função `raizInteira()` A guarda já determinada é

$$G \equiv (\mathtt{r}+1)^2 \leq \mathtt{x}$$

pelo que o progresso mais simples é a simples incrementação de `r`. Este progresso garante que o ciclo termina (i.e., que a guarda se torna falsa) ao fim de um número finito de iterações (quantas?), uma vez que `r` foi inicializada com 0. Ou seja, o passo do ciclo é

```
// CI ∧ G ≡ 0 ≤ r ∧ r² ≤ x ∧ (r + 1)² ≤ x, ou seja,
// 0 ≤ r ∧ (r + 1)² ≤ x, porque r² ≤ (r + 1)² sempre que 0 ≤ r.
acção
++r;
// CI ≡ 0 ≤ r ∧ r² ≤ x.
```

onde se assume a condição invariante como verdadeira antes da acção, se sabe que a guarda é verdadeira nesse mesmo lugar (de outra forma o ciclo teria terminado) e se pretende escolher uma acção de modo a que a condição invariante seja verdadeira também *depois do passo*, ou melhor, *apesar do progresso*. Usando a semântica da operação de atribuição, pode-se deduzir a condição mais fraca antes do progresso de modo a que condição invariante seja verdadeira no final do passo:

```
// 0 ≤ r ∧ (r + 1)² ≤ x.
acção
// 0 ≤ r + 1 ∧ (r + 1)² ≤ x, ou seja, −1 ≤ r ∧ (r + 1)² ≤ x.
++r; // o mesmo que r = r + 1;
// CI ≡ 0 ≤ r ∧ r² ≤ x.
```

Como $0 \leq \mathtt{r} \Rightarrow -1 \leq \mathtt{r}$, então é evidente que neste caso não é necessária qualquer acção, pelo que o ciclo e a função completos são

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre   PC ≡ 0eqx.
    @post CO ≡ 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = 0;
    // CI ≡ 0 ≤ r ∧ r² ≤ x.
    while((r + 1) * (r + 1) <= x)
        ++r;

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

A guarda do ciclo pode ser simplificada se se adiantar a variável `r` de uma unidade

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre PC ≡ 0eqx.
    @post CO ≡ 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
              0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x <
(raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = 1;
    // CI ≡ 1 ≤ r ∧ (r − 1)² ≤ x.
    while(r * r <= x)
        ++r;

    assert(1 <= r and (r - 1) * (r - 1) <= x and x < r * r);

    return r - 1;
}
```

Determinando se um ciclo é necessário

Deixou-se para o fim a discussão deste passo, que em rigor deveria ter tido lugar logo após a especificação do problema. É que é essa a tendência natural do programador principiante... Considere-se de novo a função `int somaDosQuadrados(int const n)`. Será mesmo necessário um ciclo? Acontece que a soma dos quadrados dos primeiros n inteiros não-negativos, pode ser expressa simplesmente por[20]

$$\frac{n(n-1)(2n-1)}{6}.$$

Não é necessário qualquer ciclo na função, que pode ser implementada simplesmente como[21]:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);
```

[20] A demonstração faz-se utilizando a propriedade telescópica dos somatórios

$$\sum_{j=0}^{n-1} (f(j) - f(j-1)) = f(n-1) - f(-1)$$

com $f(j) = j^3$.

[21] Porquê devolver `n * (n - 1) / 2 * (2 * n - 1) / 3` e não `n * (n - 1) * (2 * n - 1) / 6`? Lembre-se das limitações dos inteiros em C++.

```
    return n * (n - 1) / 2 * (2 * n - 1) / 3;
}
```

4.7.5 Um exemplo

Pretende-se desenvolver uma função que devolva verdadeiro se o valor do seu argumento inteiro não-negativo `n` for primo e falso no caso contrário. Um número inteiro não-negativo é primo se for apenas é divisível por 1 e por si mesmo. O inteiro 1 é classificado como não-primo, o mesmo acontecendo com o inteiro 0.

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 V se n for um número primo e F no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    ...
}
```

Como abordar este problema? Em primeiro lugar, é conveniente verificar que os valores 0 e 1 são casos particulares. O primeiro porque é divisível por todos os inteiros positivos e portanto não pode ser primo. O segundo porque, apesar de só ser divisível por 1 e por si próprio, não é considerado, por convenção, um número primo. Estes casos particulares podem ser tratados com recurso a uma simples instrução condicional, pelo que se pode reescrever a função como

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  0 ≤ n.
    @post éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    // PC ≡ 2 ≤ n.
    bool é_primo = ...;
    ...
    // CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
    return é_primo;
}
```

uma vez que, se a guarda da instrução condicional for falsa, e admitindo que a pré-condição da função $0 \leq \mathtt{n}$ é verdadeira, então forçosamente $2 \leq \mathtt{n}$ depois da instrução condicional. Por outro lado, a condição objectivo antes da instrução de retorno no final da função pode ser simplificada dada a nova pré-condição, mais forte que a da função. Aproveitou-se para introduzir uma variável booleana que, no final da função, deverá conter o valor lógico apropriado a devolver pela função.

Assim, o problema resume-se a escrever o código que garante que CO se verifica sempre que PC se verificar. Um ciclo parece ser apropriado para resolver este problema, pois para verificar se um número n é primo pode-se ir verificando se é divisível por algum inteiro entre 2 e $n-1$.

A condição invariante do ciclo pode ser obtida substituindo o limite superior da conjunção (a constante `n`) por uma variável `i` criada para o efeito, e com limites apropriados. Obtém-se a seguinte estrutura para o ciclo

```
// PC ≡ 2 ≤ n.
bool é_primo = ...;
int i = ...;
// CI ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(G) {
    passo
}
// CO ≡ é_primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

O que significa a condição invariante? Simplesmente que a variável `é_primo` tem valor lógico verdadeiro se e só se não existirem divisores de `n` superiores a 1 e inferiores a `i`, variando `i` entre 2 e `n`. Ou melhor, significa que, num dado passo do ciclo, já se testaram todos os potenciais divisores de `n` entre 2 e `i` *exclusive*.

A escolha da guarda é muito simples: quando o ciclo terminar $CI \wedge \neg G$ deve implicar CO. Isso consegue-se simplesmente fazendo

$$\neg G \equiv \mathtt{i} = \mathtt{n},$$

ou seja,

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

Quanto à inicialização, também é simples, pois basta atribuir 2 a `i` para que a conjunção não tenha qualquer termo e portanto tenha valor lógico verdadeiro. Assim, a inicialização é:

```
bool é_primo = true;
int i = 2;
```

pelo que o ciclo fica

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
```

```
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(i != n) {
    passo
}
// CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Que progresso utilizar? A forma mais simples de garantir a terminação do ciclo é simplesmente incrementar `i`. Dessa forma, como `i` começa com valor 2 e $2 \leq \mathtt{n}$pela pré-condição, a guarda torna-se falsa, e o ciclo termina, ao fim de exactamente $\mathtt{n} - 2$ iterações do ciclo. Assim, o ciclo fica

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(i != n) {
    acção
    ++i;
}
// CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Finalmente, falta determinar a acção a executar para garantir a veracidade da condição invariante apesar do progresso realizado. No início do passo admite-se que a condição é verdadeira e sabe-se que a guarda o é também:

$$CI \wedge G \equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n},$$

ou seja,

$$CI \wedge G \equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}.$$

Por outro lado, no final do passo deseja-se que a condição invariante seja verdadeira. Logo, o passo com as respectivas asserções é:

```
// Aqui admite-se que CI∧G ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0)∧2 ≤ i < n.
acção
++i;
// Aqui pretende-se que CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Antes de determinar a acção, é conveniente verificar qual a pré-condição mais fraca do progresso que é necessário impor para que, depois dele, se verifique a condição invariante do ciclo. Usando a transformação de variáveis correspondente à atribuição `i = i + 1` (equivalente a `++i`), chega-se a:

```
// CI ∧ G ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n.
acção
// é_ primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 2 ≤ i + 1 ≤ n, ou seja,
// é_ primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Por outro lado, se $2 \leq \mathtt{i}$, pode-se extrair o último termo da conjunção. Assim, reforçando a pré-condição do progresso,

// $CI \wedge G \equiv$ é_ primo $= (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.
acção
// é_ primo $= ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.
// é_ primo $= (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i} + 1\, :\, \mathtt{n} \div j \neq 0) \wedge 1 \leq \mathtt{i} < \mathtt{n}$.
++i;
// $CI \equiv$ é_ primo $= (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}$.

Assim sendo, a acção deve ser tal que

// é_ primo $= (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.
acção
// é_ primo $= ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.

É evidente, então, que a acção pode ser simplesmente:

```
é_primo = (é_primo and n % i != 0);
```

onde os parênteses são dispensáveis dadas as regras de precedência dos operadores em C++ (ver Secção 2.7.7).

A correcção da acção determinada pode ser verificada facilmente calculando a respectiva pré-condição mais fraca

// (é_ primo $\wedge\, \mathtt{n} \div \mathtt{i} \neq 0) = ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.
é_primo = (é_primo and n % i != 0);
// é_ primo $= (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0 \wedge 2 \leq \mathtt{i} < \mathtt{n}$.

e observando que $CI \wedge G$ leva forçosamente à sua verificação:

$$
\begin{array}{ll}
 & \text{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \\
\Rightarrow & (\text{é\_ primo} \wedge \mathtt{n} \div \mathtt{i} \neq 0) = ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}.
\end{array}
$$

Escrevendo agora o ciclo completo tem-se:

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(i != n) {
    é_primo = é_primo and n % i != 0;
    ++i;
}
// CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Reforço da guarda

A observação atenta deste ciclo revela que a variável `é_primo`, se alguma vez se tornar falsa, nunca mais deixará de o ser. Tal deve-se a que $\mathcal{F}$ é o elemento neutro da conjunção. Assim, é evidente que o ciclo poderia terminar antecipadamente, logo que essa variável tomasse o valor $\mathcal{F}$: o ciclo só deveria continuar enquanto $G \equiv \texttt{é\_ primo} \wedge \texttt{i} \neq \texttt{n}$. Será que esta nova guarda, reforçada com uma conjunção, é mesmo apropriada?

Em primeiro lugar é necessário demonstrar que, quando o ciclo termina, se atinge a condição objectivo. Ou seja, será que $CI \wedge \neg G \Rightarrow CO$? Neste caso tem-se

$$CI \wedge \neg G \equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{i}\, :\, \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} \leq \texttt{n} \wedge (\neg\texttt{é\_ primo} \vee \texttt{i} = \texttt{n}).$$

Considere-se o último termo da conjunção, ou seja, a disjunção $\neg\texttt{é\_ primo} \vee \texttt{i} = \texttt{n}$. Quando o ciclo termina pelo menos um dos termos da disjunção é verdadeiro.

Suponha-se que $\neg\texttt{é\_ primo} = \mathcal{V}$. Então, como $\texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{i}\, :\, \texttt{n} \div j \neq 0)$ também é verdadeira no final do ciclo, tem-se que $(\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{i}\, :\, \texttt{n} \div j \neq 0)$ é falsa. Mas isso implica que $(\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{n}\, :\, \texttt{n} \div j \neq 0)$, pois se não é verdade que não há divisores de `n` entre 2 e `i` *exclusive*, então também não é verdade que não os haja entre 2 e `n` *exclusive*, visto que $\texttt{i} \leq \texttt{n}$. Ou seja, a condição objectivo

$$\begin{aligned} CO &\equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{n}\, :\, \texttt{n} \div j \neq 0) \\ CO &\equiv \mathcal{F} = \mathcal{F} \\ CO &\equiv \mathcal{V} \end{aligned}$$

é verdadeira!

O outro caso, se $\texttt{i} = \texttt{n}$, é idêntico ao caso sem reforço da guarda. Logo, a condição objectivo é de facto atingida em qualquer dos casos[22].

Quanto ao passo (acção e progresso), é evidente que o reforço da guarda não altera a sua validade. No entanto, é instrutivo determinar de novo a acção do passo. A acção deve garantir a veracidade da condição invariante apesar do progresso. No início do passo admite-se que a condição é verdadeira e sabe-se que a guarda o é também:

$$\begin{aligned} CI \wedge G &\equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{i}\, :\, \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{é\_ primo} \wedge \texttt{i} \neq \texttt{n} \\ &\equiv (\mathbf{Q}\, j\, :\, 2 \leq j < \texttt{i}\, :\, \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n} \wedge \texttt{é\_ primo} \end{aligned}$$

Por outro lado, no final do passo deseja-se que a condição invariante seja verdadeira. Logo, o passo com as respectivas asserções é:

```
// Aqui admite-se que CI∧G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0)∧2 ≤ i < n∧é_ primo.
acção
++i;
// Aqui pretende-se que CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

[22] Se não lhe pareceu claro lembre-se que $(A \vee B) \Rightarrow C$ é o mesmo que $A \Rightarrow C \wedge B \Rightarrow C$ e que $a \wedge (B \vee C)$ é o mesmo que $(A \wedge B) \vee (A \wedge C)$.

Antes de determinar a acção, é conveniente verificar qual a pré-condição mais fraca do progresso que é necessário impor para que, depois dele, se verifique a condição invariante do ciclo. Usando a transformação de variáveis correspondente à atribuição `i = i + 1` (equivalente a `++i`), chega-se a:

$$
\begin{array}{l}
\texttt{//}\ \ CI \wedge G \equiv (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n} \wedge \texttt{é\_ primo}.\\
\textit{acção}\\
\texttt{//}\ \ \texttt{é\_ primo} = (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i} + 1\ :\ \texttt{n} \div j \neq 0) \wedge 1 \leq \texttt{i} < \texttt{n}.\\
\texttt{++i;}\\
\texttt{//}\ \ CI \equiv \texttt{é\_ primo} = (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} \leq \texttt{n}.
\end{array}
$$

Por outro lado, se $2 \leq \texttt{i}$, pode-se extrair o último termo da conjunção. Assim, reforçando a pré-condição do progresso,

$$
\begin{array}{l}
\texttt{//}\ \ CI \wedge G \equiv (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n} \wedge \texttt{é\_ primo}.\\
\textit{acção}\\
\texttt{//}\ \ \texttt{é\_ primo} = ((\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge \texttt{n} \div \texttt{i} \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n}.\\
\texttt{//}\ \ \texttt{é\_ primo} = (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i} + 1\ :\ \texttt{n} \div j \neq 0) \wedge 1 \leq \texttt{i} < \texttt{n}.\\
\texttt{++i;}\\
\texttt{//}\ \ CI \equiv \texttt{é\_ primo} = (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} \leq \texttt{n}.
\end{array}
$$

Assim sendo, a acção deve ser tal que

$$
\begin{array}{l}
\texttt{//}\ \ (\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n} \wedge \texttt{é\_ primo}.\\
\textit{acção}\\
\texttt{//}\ \ \texttt{é\_ primo} = ((\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge \texttt{n} \div \texttt{i} \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n}.
\end{array}
$$

É evidente, então, que a acção pode ser simplesmente:

```
é_primo = n % i != 0;
```

A acção simplificou-se relativamente à acção no ciclo com a guarda original. A alteração da acção pode ser percebida observando que, sendo a guarda sempre verdadeira no início do passo do ciclo, a variável `é_primo` tem aí sempre o valor verdadeiro, pelo que a acção antiga tinha uma conjunção com a veracidade. Como $\mathcal{V} \wedge P$ é o mesmo que P, pois $\mathcal{V}$ é o elemento neutro da conjunção, a conjunção pode ser eliminada.

A correcção da acção determinada pode ser verificada facilmente calculando a respectiva pré-condição mais fraca:

$$
\begin{array}{l}
\texttt{//}\ \ (\texttt{n} \div \texttt{i} \neq 0) = ((\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge \texttt{n} \div \texttt{i} \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n}.\\
\texttt{é\_primo = n \% i != 0;}\\
\texttt{//}\ \ \texttt{é\_ primo} = ((\mathbf{Q}\,j\ :\ 2 \leq j < \texttt{i}\ :\ \texttt{n} \div j \neq 0) \wedge \texttt{n} \div \texttt{i} \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n}.
\end{array}
$$

e observando que $CI \wedge G$ leva forçosamente à sua verificação:

$$\begin{array}{ll} & (\mathbf{Q}\, j\,:\, 2 \leq j < \mathtt{i}\,:\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \wedge \texttt{é\_ primo} \\ \Rightarrow & (\mathtt{n} \div \mathtt{i} \neq 0) = ((\mathbf{Q}\, j\,:\, 2 \leq j < \mathtt{i}\,:\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}. \end{array}$$

Escrevendo agora o ciclo completo tem-se:

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(é_primo and i != n) {
    é_primo = n % i != 0;
    ++i;
}
// CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Versão final

Convertendo para a instrução `for` e inserindo o ciclo na função tem-se:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    bool é_primo = true;
    // CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    for(int i = 2; é_primo and i != n; ++i)
        é_primo = n % i != 0;

    return é_primo;
}
```

A função inclui desde o início uma instrução de asserção que verifica a veracidade da pré-condição. E quanto à condição objectivo? A condição objectivo envolve um quantificador, pelo que não é possível exprimi-la na forma de uma expressão booleana em C++, excepto recorrendo a funções auxiliares. Como resolver o problema? Uma hipótese passa por reflectir na condição objectivo apenas os termos da condição objectivo que não envolvem quantificadores:

```
assert(not é_primo or 2 <= n);
```

Recorde-se que $A \Rightarrow B$ é o mesmo que $\neg A \vee B$. A expressão da asserção verifica portanto se $\texttt{é\_ primo} \Rightarrow 2 \leq \texttt{n}$.

Esta asserção é pouco útil, pois deixa passar como primos ou não primos todos os números não primos superiores a 2... Pode-se fazer melhor. Todos os inteiros positivos podem ser escritos na forma $6k - l$, com $l = 0, \ldots, 5$ e k inteiro positivo. É imediato que os inteiros das formas $6k$, $6k - 2$, $6k - 3$ e $6k - 4$ não podem ser primos (com excepção de 2 e 3). Ou seja, os números primos ou são o 2 ou o 3, ou são sempre de uma das formas $6k - 1$ ou $6k - 5$. Assim, pode-se reforçar um pouco a asserção final para:

```
assert(((n != 2 and n != 3) or é_primo) and
       (not é_primo or n == 2 or n == 3 or
        (2 <= n and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));
```

que, expressa em notação matemática fica

$$((\texttt{n} = 2 \vee \texttt{n} = 3) \Rightarrow \texttt{é\_ primo}) \wedge$$
$$(\texttt{é\_ primo} \Rightarrow (\texttt{n} = 2\texttt{n} = 3 \vee (2 \leq \texttt{n} \wedge ((\texttt{n} + 1) \div 6 = 0 \vee (\texttt{n} + 5) \div 6 = 0))))$$

A função fica então:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    bool é_primo = true;
    // CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    for(int i = 2; é_primo and i != n; ++i)
        é_primo = n % i != 0;

    assert(n == 2 or n == 3 or
           (5 <= n and ((n - 1) % 6 == 0 or (n - 5) % 6 == 0)));

    assert(((n != 2 and n != 3) or é_primo) and
           (not é_primo or n == 2 or n == 3 or
            (2 <= n and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));

    return é_primo;
}
```

Desta forma continuam a não se detectar muitos possíveis erros, mas passaram a detectar-se bastante mais do que inicialmente.

Finalmente, uma observação atenta da função revela que ainda pode ser simplificada (do ponto de vista da sua escrita) para

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    // CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    for(int i = 2; i != n; ++i)
        if(n % i != 0)
            return false;

    return true;
}
```

Este último formato é muito comum em programas escritos em C ou C++. A demonstração da correcção de ciclos incluindo saídas antecipadas no seu passo é um pouco mais complicada e será abordada mais tarde.

Outra abordagem

Como se viu, o valor $2 \leq \mathtt{n}$ é primo se nenhum inteiro entre 2 e n *exclusive* o dividir. Mas pode-se pensar neste problema de outra forma mais interessante. Considere-se o conjunto $\mathbf{D}$ de todos os possíveis divisores de n. Claramente o próprio n é sempre membro deste conjunto, $\mathtt{n} \in \mathbf{D}$. Se n for primo, o conjunto tem apenas como elemento o próprio n, i.e., $\mathbf{C} = \{\mathtt{n}\}$. Se n não for primo existirão outros divisores no conjunto, forçosamente inferiores ao próprio n. Considere-se o mínimo desse conjunto. Se n for primo, o mínimo é o próprio n. Se n não for primo, o mínimo é forçosamente diferente de n. Ou seja, a afirmação "n é primo" tem o mesmo valor lógico que "o mais pequeno dos divisores não-unitários de n é o n".

Regresse-se à estrutura da função:

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre  0 ≤ n.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
```

```
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    // PC ≡ 2 ≤ n.
    ...
    return ...;
    // CO ≡ éPrimo = min {2 ≤ j ≤ n : n ÷ j = 0} = n.
}
```

Introduza-se uma variável `i` que se assume conter o mais pequeno dos divisores não-unitários de `n` no final da função. Nesse caso a função deverá devolver o valor lógico de $\mathtt{i} = \mathtt{n}$. Então pode-se reescrever a função:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  0 ≤ n.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);


    if(n == 0 || n == 1)
        return false;

    // PC ≡ 2 ≤ n.
    int i = ...;
    ...
    // CO ≡ i = min {2 ≤ j ≤ n : n ÷ j = 0}.
    return i == n;
    // éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n).
}
```

O problema reduz-se pois a escrever um ciclo que, dada a pré-condição PC, garanta que no seu final se verifique a nova CO. A condição objectivo pode ser reescrita numa forma mais simpática. Se `i` for o menor dos divisores de `n` entre 2 e `n` *inclusive*, então

1. `i` está entre 2 e `n` *inclusive*.
2. `i` tem de ser divisor de `n`, i.e., $\mathtt{n} \div \mathtt{i} = 0$, e
3. nenhum outro inteiro inferior a `i` e superior ou igual a 2 é divisor de `n`, i.e., $(\mathbf{Q}\, j\ :\ 2 \leq j < \mathtt{i}\ :\ \mathtt{n} \div j \neq 0)$.

Traduzindo para notação matemática:

$$CO \equiv \mathtt{n} \div \mathtt{i} = 0 \wedge (\mathbf{Q}\, j\ :\ 2 \leq j < \mathtt{i}\ :\ \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}.$$

Como a nova condição objectivo é uma conjunção, pode-se tentar obter a condição invariante e a negação da guarda do ciclo por factorização. A escolha mais evidente faz do primeiro termo a guarda e do segundo a condição invariante

$$CO \equiv \overbrace{\mathtt{n} \div \mathtt{i} = 0}^{\neg G} \wedge \overbrace{(\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}}^{CI},$$

conduzindo ao seguinte ciclo

```
// PC ≡ 2 ≤ n.
int i = ...;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0) {
    passo
}
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

A escolha da inicialização é simples. Como a conjunção de zero termos é verdadeira, basta fazer

```
int i = 2;
```

pelo que o ciclo fica

```
// PC ≡ 2 ≤ n.
int i = 2;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0) {
    passo
}
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Mais uma vez o progresso mais simples é a incrementação de `i`

```
// PC ≡ 2 ≤ n.
int i = 2;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0) {
    acção
    ++i;
}
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Com este progresso o ciclo termina na pior das hipóteses com $\mathtt{i} = \mathtt{n}$. Que acção usar? Antes da acção sabe-se que a guarda é verdadeira e admite-se que a condição invariante o é também. Depois do progresso pretende-se que a condição invariante seja verdadeira:

```
// CI ∧ G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n ∧ n ÷ i ≠ 0.
acção
++i;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Como habitualmente, começa por se determinar a pré-condição mais fraca do progresso que garante a verificação da condição invariante no final do passo:

```
// CI ∧ G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n ∧ n ÷ i ≠ 0,
// que, como n ÷ i ≠ 0 implica que i ≠ n, é o mesmo que
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n.
acção
// (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 2 ≤ i + 1 ≤ n, ou seja,
// (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Fortalecendo a pré-condição do progresso de modo garantir que $2 \leq$ `i`, pode-se extrair o último termo da conjunção:

```
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n.
acção
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n, ou seja,
// (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

pelo que a acção pode ser a instrução nula! O ciclo fica pois

```
// PC ≡ 2 ≤ n.
int i = 2;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0)
    ++i;
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

e a função completa é

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre 0 ≤ n.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
```

```
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    int i = 2;
    // CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    while(n % i != 0)
        ++i;

    assert(((n != 2 and n != 3) or (i == n)) and
           (i != n or n == 2 or n == 3 or
            (2 <= n and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));

    return i == n;
}
```

A instrução de asserção para verificação da condição objectivo foi obtida por simples adaptação da obtida na secção anterior.

Discussão

Há inúmeras soluções para cada problema. Neste caso começou-se por uma solução simples mas ineficiente, aumentou-se a eficiência recorrendo ao reforço da guarda e finalmente, usando uma forma diferente de expressar a condição objectivo, obteve-se um ciclo mais eficiente que os iniciais, visto que durante o ciclo é necessário fazer apenas uma comparação e uma incrementação.

Mas há muitas outras soluções para este mesmo problema, e mais eficientes. Recomenda-se que o leitor tente resolver este problema depois de aprender sobre matrizes, no Capítulo 5. Experimente procurar informação sobre um velho algoritmo chamado a "peneira de Eratóstenes".

4.7.6 Outro exemplo

Suponha-se que se pretende desenvolver um procedimento que, dados dois inteiros como argumento, um o dividendo e outro o divisor, sendo o dividendo não-negativo e o divisor positivo, calcule o quociente e o resto da divisão inteira do dividendo pelo divisor e os guarde em variáveis externas ao procedimento (usando passagem de argumentos por referência). Para que o problema tenha algum interesse, não se pode recorrer aos operadores `*`, `/` e `%` do C++, nem tão pouco aos operadores de deslocamento *bit*-a-*bit*. A estrutura do procedimento é (ver Secção 4.6.6)

```
/** Coloca em q e r respectivamente o quociente e o resto da divisão
    inteira de dividendo por divisor.
```

@pre $PC \equiv 0 \leq \texttt{dividendo} \wedge 0 < \texttt{divisor}$.
@post $CO \equiv 0 \leq \texttt{r} < \texttt{divisor} \wedge \texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r}$.

```
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    assert(0 <= dividendo and 0 < divisor);

    ...

    assert(0 <= r and r < divisor and
           dividendo = q * divisor + r);
}
```

A condição objectivo pode ser vista como uma definição da divisão inteira. Não só o quociente `q` multiplicado pelo divisor e somado do resto `r` tem de resultar no dividendo, como o resto tem de ser não-negativo e menor que o divisor (senão não estaria completa a divisão!), existindo apenas uma solução nestas circunstâncias.

Como é evidente a divisão tem de ser conseguida através de um ciclo. Qual será a sua condição invariante? Neste caso, como a condição objectivo é a conjunção de três termos

$$CO \equiv 0 \leq \texttt{r} \wedge \texttt{r} < \texttt{divisor} \wedge \texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r},$$

a solução passa por obter a condição invariante e a negação da guarda por factorização da condição objectivo.

Mas quais das proposições usar para $\neg G$ e quais usar para CI? Um pouco de experimentação e alguns falhanços levam a que se perceba que a negação da guarda deve corresponder ao segundo termo da conjunção, ou seja, reordenando os termos da conjunção,

$$CO \equiv \overbrace{\texttt{r} < \texttt{divisor}}^{\neg G} \wedge \overbrace{\texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge 0 \leq \texttt{r}}^{CI},$$

Dada esta escolha, a forma mais simples de inicializar o ciclo é fazendo

```
r = dividendo;
q = 0;
```

pois substituindo os valores na condição invariante obtém-se

$$CI \equiv \texttt{dividendo} = 0 \times \texttt{divisor} + \texttt{dividendo} \wedge 0 \leq \texttt{dividendo},$$

que é verdadeira dado que a pré-condição garante que dividendo é não-negativo.

O ciclo terá portanto a seguinte forma:

```
r = dividendo;
q = 0;
// CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
while(divisor <= r) {
    passo
}
```

O progresso deve ser tal que a guarda se torne falsa ao fim de um número finito de iterações do ciclo. Neste caso é claro que basta ir reduzindo sempre o valor do resto para que isso aconteça. A forma mais simples de o fazer é decrementá-lo:

```
--r;
```

Para que CI seja de facto invariante, há que escolher uma acção tal que:

```
// Aqui admite-se que:
// CI ∧ G ≡ dividendo = q × divisor + r ∧ 0 ≤ r ∧ divisor ≤ r, ou seja,
// dividendo = q × divisor + r ∧ divisor ≤ r.
acção
--r;
// Aqui pretende-se que CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
```

Antes de determinar a acção, verifica-se qual a pré-condição mais fraca do progresso que leva à veracidade da condição invariante no final do passo:

```
// dividendo = q × divisor + r − 1 ∧ 0 ≤ r − 1.
--r; // o mesmo que r = r - 1;
// CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
```

A acção deve portanto ser tal que

```
// dividendo = q × divisor + r ∧ divisor ≤ r.
acção
// dividendo = q × divisor + r − 1 ∧ 1 ≤ r.
```

As únicas variáveis livres no procedimento são `r` e `q`, pelo que se o progresso fez evoluir o valor de `r`, então a acção deverá actuar sobre `q`. Deverá pois ter o formato:

```
q = expressão;
```

Calculando a pré-condição mais fraca da acção:

```
// dividendo = expressão × divisor + r − 1 ∧ 1 ≤ r.
q = expressão;
// dividendo = q × divisor + r − 1 ∧ 1 ≤ r.
```

a expressão deve ser tal que

```
// dividendo = q × divisor + r ∧ divisor ≤ r.
// dividendo = expressão × divisor + r − 1 ∧ 1 ≤ r.
```

É evidente que a primeira das asserções só implica a segunda se `divisor` $= 1$. Como se pretende que o ciclo funcione para qualquer divisor positivo, houve algo que falhou. Uma observação mais cuidada do passo leva a compreender que o progresso não pode fazer o resto decrescer de 1 em 1, mas de `divisor` em `divisor`! Se assim for, o passo é

```
// CI ∧ G ≡ dividendo = q × divisor + r ∧ divisor ≤ r.
acção
r -= divisor;
// CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
```

e calculando de novo a pré-condição mais fraca do progresso

```
// dividendo = q × divisor + r − divisor ∧ 0 ≤ r − divisor, ou seja,
// dividendo = (q − 1) × divisor + r ∧ divisor ≤ r.
r -= divisor; // o mesmo que r = r - divisor;
// CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
```

A acção deve portanto ser tal que

```
// dividendo = q × divisor + r ∧ divisor ≤ r.
acção
// dividendo = (q − 1) × divisor + r ∧ divisor ≤ r.
```

É evidente então que a acção pode ser simplesmente:

```
++q;
```

Tal pode ser confirmado determinando a pré-condição mais fraca da acção

```
// dividendo = (q + 1 − 1) × divisor + r ∧ divisor ≤ r, ou seja,
// dividendo = q × divisor + r ∧ divisor ≤ r.
++q; // o mesmo que q = q + 1;
// dividendo = (q − 1) × divisor + r ∧ divisor ≤ r.
```

e observando que $CI \wedge G$ implica essa pré-condição. A implicação é trivial neste caso, pois as duas asserções são idênticas!

Transformando o ciclo num ciclo for e colocando no procedimento:

```
/** Coloca em q e r respectivamente o quociente e o resto da divisão
    inteira de dividendo por divisor.
    @pre  PC ≡ 0 ≤ dividendo ∧ 0 < divisor.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    assert(0 <= dividendo and 0 < divisor);

    r = dividendo;
    q = 0;
    // CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
    while(divisor <= r) {
        ++q;
        r -= divisor;
    }

    assert(0 <= r and r < divisor and
           dividendo = q * divisor + r);
}
```

que também é comum ver escrito em C++ como

```
/** Coloca em q e r respectivamente o quociente e o resto da divisão
    inteira de dividendo por divisor.
    @pre  PC ≡ 0 ≤ dividendo ∧ 0 < divisor.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    // CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
    for(r = dividendo, q = 0; divisor <= r; ++q, r -= divisor)
        ;

    assert(0 <= r and r < divisor and
           dividendo = q * divisor + r);
}
```

onde o progresso da instrução `for` contém o passo completo. Esta é uma expressão idiomática do C++ pouco clara, e por isso pouco recomendável, mas que ilustra a utilização de um novo operador: a vírgula usada para separar o progresso e a acção do ciclo é o operador de sequenciamento do C++. Esse operador garante que o primeiro operando é calculado antes do segundo operando, e tem como resultado o valor do segundo operando. Por exemplo, as instruções

```
int n = 0;
n = (1, 2, 3, 4);
```

colocam o valor 4 na variável `n`.

Podem-se desenvolver algoritmos mais eficientes para a divisão inteira se se considerarem progressos que façam diminuir o valor do resto mais depressa. Uma ideia interessante é subtrair ao resto não apenas o divisor, mas o divisor multiplicado por uma potência tão grande quanto possível de 2.