Documentação

"Programming is best regarded as the process of creating *works of literature*, which are meant to be read."
— Donald E. Knuth, *Literate Programming*

"Computer science is no more about computers than astronomy is about telescopes."
— Edsger W. Dijkstra

"Comments are, at best, a necessary evil.
The proper use of comments is to compensate for our failure to express ourself in code [...]
Truth can only be found in one place: the code."
— Robert C. Martin, *Clean Code*

"Use definite, specific, concrete language."
"Write with nouns and verbs."
"Put the emphatic words at the end."
"Omit needless words."
— W. Strunk, Jr. & E.B. White, *The Elements of Style*

Há quem diga que *documentar* um programa é o mesmo que escrever muitos comentários de mistura com o código. Isso está errado! Uma boa documentação não suja o código com comentários. Uma boa documentação limita-se a

> explicar *o que* cada função do programa faz.

Uma boa documentação não perde tempo tentando explicar *como* a função faz o que faz, porque o leitor interessado nessa questão pode ler o código.

A distinção entre *o que* e *como* é a mesma que existe entre a interface (arquivo `.h`) e a implementação (arquivo `.c`) de uma biblioteca C. A seguinte analogia pode tornar mais clara a diferença. Uma empresa de entregas promete apanhar o seu pacote em São Paulo e entregá-lo em Manaus. Isso é *o que* a empresa faz. *Como* o serviço será feito — se o transporte será terrestre, aéreo ou marítimo, por exemplo — é assunto interno da empresa.

Em suma, a documentação de uma função é um *minimanual* que dá instruções completas sobre o uso correto da função. (Portanto, o conceito de documentação se confunde com a ideia de API.) Esse minimanual deve dizer o que a função recebe e o que devolve. Em seguida, deve dizer, de maneira exata, que efeitos a função produz, ou seja, qual a relação entre o que a função recebe e o que devolve.

Uma documentação correta é uma questão de honestidade intelectual, pois coloca nas mãos do leitor/usuário o poder de constatar e provar que a função está errada (quanto esse for o caso).

Sumário:

Exemplo

Em um dos capítulos deste sítio há uma função que encontra o valor de um elemento máximo de um vetor. Vamos repetir aqui o código daquela função juntamente com uma documentação perfeita:

```
// A seguinte função recebe um número n >= 1
// e um vetor v e devolve o valor de um
// elemento máximo de v[0..n-1].

int max (int n, int v[]) {
   int x = v[0];
   for (int j = 1; j < n; j += 1)
      if (x < v[j])
         x = v[j];
   return x;
}
```

Veja como a documentação é simples, exata e completa. A documentação diz *o que* a função faz mas não perde tempo tentando explicar *como* a função faz o que faz (por exemplo, se a função é recursiva ou iterativa, se percorre o vetor da esquerda para a direita ou vice-versa, etc.). Observe também que não há comentários inúteis (como "`o índice j vai percorrer o vetor`" ou "`x é o máximo provisório`") sujando o código.

Veja a seguir alguns exemplos de má documentação da função. Dizer apenas que

```
a função devolve o valor de um elemento máximo
de um vetor
```

é *indecentemente vago*, pois nem sequer menciona os parâmetros (`n` e `v`) da função! Dizer que

```
a função devolve o valor de um elemento máximo
do vetor v
```

é um pouquinho melhor, mas ainda *muito vago*: o leitor fica sem saber qual o papel do parâmetro `n`. Dizer que

```
a função devolve o valor de um elemento máximo
de um vetor v que tem n elementos
```

é melhor, mas ainda está vago: não se sabe se o vetor é `v[0..n-1]` ou `v[1..n]`. Dizer que

```
a função devolve o valor de um elemento máximo
de v[0..n-1]
```

já está quase bom, mas sonega a informação de que a função só faz sentido se $n \geq 1$.

Outro exemplo

Em um dos capítulos deste sítio há uma função que decide se um número x é igual a algum elemento de um vetor v. Repetimos aqui o código daquela função juntamente com uma documentação perfeita:

```
// Recebe um número x, um vetor v,
// e um índice n >= 0. Devolve 1 se
// x está em v[0..n-1] e devolve 0
// em caso contrário.

int busca (int x, int n, int v[]) {
   int j = 0;
   while (j < n && v[j] != x)
      j += 1;
   if (j < n) return 1;
   else return 0;
}
```

A documentação diz, de maneira exata e completa, *o que* a função faz. Ela não perde tempo tentando explicar *como* a função faz o serviço. Para contrastar, veja alguns exemplos de má documentação. Dizer

```
a função decide se x está em v[0..n-1]
```

é um pouco vago, pois o leitor precisa adivinhar o que a função devolve. Dizer

```
a função decide se x está no vetor v
```

é muito vago, pois não explica o papel do parâmetro n. Dizer

```
a função decide se um número está em um vetor
```

é absurdamente vago, pois nem sequer menciona os parâmetros da função!

Mais um exemplo

Eis uma função acompanhada de documentação perfeita:

```
// Recebe x, v e n >= 0 e devolve j
// tal que 0 <= j < n e v[j] == x.
// Se tal j não existe, devolve n.

int onde (int x, int v[], int n) {
   int j = 0;
   while (j < n && v[j] != x)
      j += 1;
   return j;
}
```

Exercícios 1

1. Considere a seguinte documentação de uma função: "Esta função recebe números inteiros p,

`q, r, s` e devolve a média aritmética de `p, q, r`." O que há de errado?

2. Considere a seguinte documentação de uma função: "Esta função recebe números inteiros `p, q, r` tais que `p <= q <= r` e devolve a média aritmética de `p, q, r`." O que há de errado?

Observações

1. A documentação de uma função tem o importante papel de separar as responsabilidades do programador das do usuário. Cabe ao programador dizer, na documentação, quais os valores válidos de cada parâmetro da função; cabe ao usuário a tarefa de verificar, antes de invocar a função, se os valores dos argumentos são válidos. Com esse acordo, o programador *fica dispensado de verificar a validade dos argumentos* e pode dedicar toda a sua atenção à solução do problema que a função deve resolver.

2. Nos demais capítulos deste sítio, por conveniência tipográfica, a documentação de muitas funções não foi integrada ao código (na forma de `// comentário`) *mas escrita no texto que precede o código*.

3. Existem excelente ferramentas para integrar código com documentação. Veja, por exemplo, o sistema CWEB de D.E. Knuth. Para ilustrar, preparei dois programas em CWEB: mdp e isort.

Invariantes de iterações

Há uma situação em que comentários misturados com código são úteis. O corpo de muitas funções consiste em um processo iterativo (controlado por um `for` ou um `while`). Nesses casos, depois de dizer *o que* a função faz, você pode enriquecer a documentação dizendo quais os *invariantes* do processo iterativo. Um *invariante* é uma *relação entre os valores das variáveis*

que vale no início de cada iteração

e não se altera de uma iteração para outra. Essas relações invariantes explicam o funcionamento do processo iterativo e permitem *provar*, por indução, que ele tem o efeito desejado.

EXEMPLO 1. A função `max` calcula o valor de um elemento máximo de `v[0..n-1]`. O comentário embutido no código dá o invariante do processo iterativo:

```
int max (int n, int v[]) {
   int x = v[0];
   for (int j = 1; j < n; ++j)
      // neste ponto, x é um
      // elemento máximo de v[0..j-1]
      if (x < v[j])
         x = v[j];
   return x;
}
```

EXEMPLO 2. Digamos que um segmento `v[i..j]` de um vetor `v[0..n-1]` é *constante* se todos os seus elementos têm o mesmo valor. A função `scmax` abaixo recebe um vetor `v[0..n-1]`,

com `n > 0`, e devolve o comprimento de um segmento constante máximo. O comentário embutido no código dá o invariante do processo iterativo:

```
int scmax (int v[], int n) {
   int i = 0, max = 0;
   while (/*A*/ i < n) { // no ponto A,
      // 1. max é o comprimento de um segmento
      //    constante máximo de v[0..i-1] e
      // 2. i == 0 ou v[i-1] != v[i] ou i == n
      int j = i+1;
      while (j < n && v[j] == v[i]) ++j;
      if (max < j-i) max = j-i;
      i = j;
   }
   return max;
}
```

[Carlos A. Estombelo-Montesco encontrou um erro na versão anterior do código.]

Invariantes são essenciais para entender *por que* uma função ou um algoritmo estão corretos. Vários exemplos de prova de correção de algoritmos baseada em invariantes aparecem nos capítulos dedicadas à busca binária, à ordenação, ao mergesort, ao heapsort, e ao quicksort.

Invariantes podem ajudar a escolher entre duas versões equivalentes de um trecho de código: a versão mais elegante tem invariantes mais simples.

Exercícios 2

1. Um programador inexperiente afirma que a seguinte proposição é um invariante da função `max` acima: "`x` é o maior elemento da parte do vetor `v` vista até agora." Critique essa afirmação.
2. A seguinte documentação da função `scmax` está correta?

```
// a função recebe um vetor crescente
// v[0..n-1] com n >= 1 e devolve o
// comprimento de um segmento constante
// máximo do vetor
```

Perguntas e respostas

- PERGUNTA: Num dos exemplos acima aparece a expressão `while (j < n && v[j] != x)`. Eu não deveria escrever `while ((j < n) && (v[j] != x))`?

  RESPOSTA: Não. Os parênteses adicionais são supérfluos porque os operadores `<` e `!=` têm precedência sobre `&&`.

- PERGUNTA: Como devo traduzir o "`return`" que aparece em todos os programas em C?

  RESPOSTA: Em inglês, o verbo *to return* tem dois significados diferentes: *retornar* (ou seja, *voltar*) e *devolver*. Não é bom confundir esses dois significados. Na grande maioria das vezes, um `return` num programa em C tem o sentido de *devolver* e não o de *retornar*.

Veja o verbete *Software documentation* na Wikipedia.

Veja o verbete *Application programming interface* (interface para programação de aplicações) na

Wikipedia.

Veja o capítulo *Leiaute*.

Atualizado em 2019-01-01
https://www.ime.usp.br/~pf/algoritmos/

DCC-IME-USP