1. Introdução ao uso de funções

Se você precisar de um *algoritmo* com um objetivo muito bem definido (e.g. um *algoritmo* para encontrar o menor valor dentro uma *lista* dada) é melhor **encapsulá-lo** em um bloco de modo a poder invocá-lo sempre que precisar desse *algoritmo*. Geralmente todas as linguagens de programação possibilita esse *encapsulamento* e isso é feito na forma de uma função ou de um procedimento (um *procedimentos* é análogo à uma função, entretanto nele **não** existe qualquer devolução). A grande vantagem dessa abordagem é deixar o seu código melhor organizado, além de permitir que a função seja utilizada em outros trechos do código sem a necessidade de codificá-la novamente!

Para ampliar as vantagens do uso de *funções* as *linguagens de programação* permitem o *aninhamento* de funções (funções dentro de funções) e a declaração de variáveis locais.
Se desejar examinar o aninhamento de funções em *C* siga essa URL.
Se desejar examinar o aninhamento de funções em *Python* siga essa URL.

2. Um paralelo entre função matemática e função em linguagens de programação

A ideia básica de uma função, implementada em alguma linguagem de programação, é **encapsular** um código que poderá ser invocado/chamado por qualquer outro trecho do programa. Seu significado e uso são muito parecidos com o de funções matemáticas, ou seja, existe um nome, uma definição e posterior invocação à função.

Vamos explicitar o conceito de **função matemática** para depois verificar que a contra-parte em computação é semelhante. Tomemos um exemplo simples, função *cúbica*: *f(x) = x^2* é sua definição (eventualmente explicitamos *f:IR → IR*) e exemplos de seu uso poderia ser *f(-1)* e *f(1/2)*, que devolvem, respectivamente, *1* e *1/4*.

Por outro lado, existem funções mais "complicadas", como a função trigométrica *cosseno(x)*, que não pode ser escrita a partir de operações somas e multiplicações (é uma função *transcendental*) e portanto é necessário algum "truque" matemático para obtê-la de modo aproximado. Isso pode ser feito via série de *Taylor*, nesse paso poderíamos definir: *cos:IR → IR*, $cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! + ...$
Para quem deseja saber mais: Esta é uma função que é impossível de ser implementada de **modo exato** em um computador digital, entretanto é viável obter boas aproximações utilizando o conceito de *série de Taylor*. A série é conseguida escrevendo um polinômio *p(x)* que tem a seguinte propriedade: *p(0) = cos(0)* e as derivadas em tordo no *0* de todas as ordens de *p(x)* coincide com mesmas derivadas do *cos(x)*. Fazendo os cálculos das derivadas de *p(x)* e de *cos(x)* no ponto *0* e forçando-se as igualdades, podemos encontrar os coeficientes para *p(x)* Do ponto de vista prático, a série que aproxima a função *cosseno*, para valores de x próximo à origem é:
(1) $cosseno(x) \approx 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! + ...$

Note que o *lado direito* da equação (1) é a definição da função e ela está escrita em termos do **parâmetro formal** *x*. Ao "chamarmos" a função com **parâmetros efetivos**, é sobre o valor desses parâmetros que computamos o lado direito da expressão, por exemplo, computar *cos(0.1)* ou de *cos(1.1)*.

3. Por que usar o conceito de função?

Assim, implementar códigos com objetivos específicos (como computar o *cosseno* de qualquer valor) apresentam três grandes vantagens:

1. Facilita o desenvolvimento (*desenvolvimento modular*): implementa-se uma *particular unidade*, um trecho menor, concentrando-se nele, até que ele esteja funcionando com alto grau de *confiabilidade*;
2. Organização: o código fica melhor organizado e portanto mais fácil de ser mantido;
3. Reaproveitamento: sempre que precisar aplicar o código encapsulado em qualquer outro trecho de código (ou noutro código), pode-se utilizar aquele que já foi implementado e é *confiável*.

4. Sobre a definição (seus parâmetros) e uso de funções em *Portugol*

Assim, agrupar trechos com objetivos específicos, implementando-os na forma de uma *função* ajuda bastante no desenvolvimento e na organização de códigos em programação. Vejamos isso inicialmente uma pseudo-linguagem de programação, em *Portugol*.

Do ponto de vista prático, a estrutura básica de uma função em uma linguagem de programação está representada abaixo, com a *declaração da função* e sua *lista de parâmetros formais*, seguido de sua invocação (quando providenciamos os *parâmetro efetivos*).

Declaração:
```
[eventual_tipo_de_retorno] nome_da_funcao
(lista_parametros_formais)
  comando1
  ...
  comandoN
  devolva EXP //# devolve o valor em EXP para quem chamou a
funcao
```

Uso:
```
var = nome_da_funcao(lista_parametros_efetivos)
```

Note que, assim como em matemática a função compucional deve **devolver algo**, o que é feito no comando *Portugol* `devolva`.
Neste texto consideraremos apenas funções que devolvem valores inteiros, mas poderia ser qualquer outro tipo válido para a linguagem em foco.
No esquema *Portugol* acima, o tipo a ser devolvido está indicado no `eventual_tipo_de_retorno`. Na linguagem *C* esse campo é obrigatório, enquanto em *Python* ele não é explicitado (o tipo a ser devolvido é automaticamente deduzido a partir da variável no `return`). Vide código 1 na seção 6.

Atenção ao **`devolva`** (geralmente nas linguagens práticas **`return`**), além dele estar associado com o "tipo da função", ele é um comando muito especial: se ele for alcançado (e.g., ele pode estar subordinado a um comando de seleção `if`), então a execução da função é **interrompida**, retornando-se ao ponto imediatamente após o ponto de chamada da função (com o valor de devolução).
No caso do comando ter uma expressão qualquer, como indicado no exemplo `return EXP`, o valor de `EXP` é devolvido ao "ponto de chamada", neste caso a chamada deve estar dentro de uma expressão (lógica ou artimética de acordo com o tipo de `EXP`) ou ser o lado direito de uma *atribuição*. O primeiro caso ocorre nos exemplos do código 1 e o segundo caso é ilustrado na figura 1 (atribuições para *a*, *b* e *c*.

A *lista de parâmetros* pode conter vários nomes de variáveis, geralmente, separadas por vírgula. Por exemplo, se houver necessidade de uma função que realiza vários cálculos com três variáveis pode-se usar como declaração da função algo como: `nome_da_funcao (var1, var2, var3)`.

4.1 Parâmetros formais e efetivos

De modo geral, a diferença entre os *parâmetros formais* e *efetivos* é que o primeiro corresponde ao nome da variável utilizada dentro da função, enquanto o segundo é o nome da variável que será usado para iniciar o parâmetro formal ao iniciar a execução da função.

Assim durante a execução, ao encontrar uma chamada à função `nome_da_funcao`, o fluxo de execução segue a partir do código da função. Mas antes de executar a primeira linha de código da função, os valores dos *parâmetros efetivos* servem para inicializar os parâmetros formais (que são também variáveis locais à função). Após esta inicialização, inicia-se a execução do código da função e geralmente ao final, encontra-se um comando do tipo "retorne devolvendo um valor" (*return*).

4.2 Ilustrando a execução de um trecho de programa com 3 chamadas à mesma função

Suponhamos que precisemos computar o valor da combinação de *N* tomado *k* a *k*, ou seja, C(N,k) = N! / (k! (N-k)!). Para isso percebe-se que é necessário implementar o cômputo de fatorial que será utilizado 3 vezes. Para facilitar a compreensão, podemos escrever um código com 3 variáveis auxiliares para armazenar, respectivamente, *N!*, *k!* e *(N-k)!*.

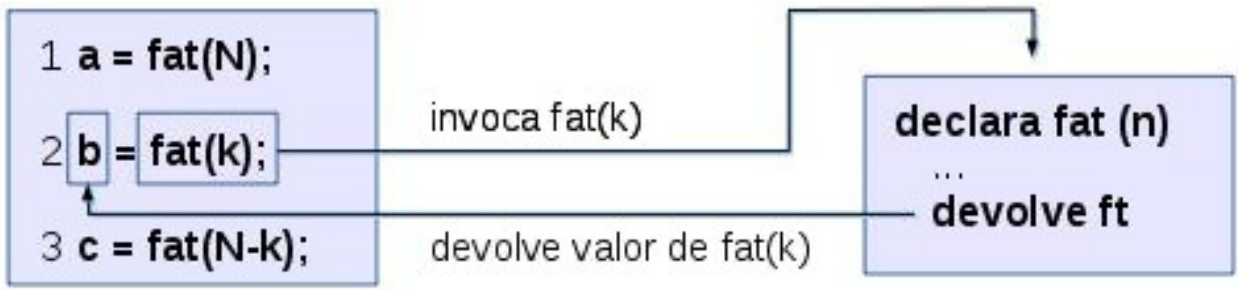

*Fig. 1. Ilustração do fluxo de execução ao **invocar** uma função para computar o **fatorial** de um natural.*

A figura 1 ilustra a execução do código para computar *C(N,k)*, com o retângulo à esquerda com a instrução que invoca a função *fat* e na direita o resumo da função *fat*. Para entender o fluxo de execução destacaremos a execução da instrução na linha 2. Como `b = fat(k);` é uma atribuição, primeiro computa-se o valor do *lado direito* da atribuição e só depois atribui-se à variável, no *lado esquerdo* da atribuição, o valor que ela passará a ter, ou seja,

1. primeiro executa-se o cômputo de `fat(k)`, para isso
2. pega-se o valor do *parâmetro efetivo* `k` e usa-o para iniciar o *parâmetro formal* `n` da função
3. então inicia-se a execução da função `fat`
4. ao final da função `fat`, o valor de *n!* (que está em `ft`) é devolvido para quem invocou `fat(.)`,
5. então, este valor é atribuído à variável `b`
6. seguindo-se para a execução da próxima linha (3).

Vale notar que a execução da instrução `c = fat(N-k);` produzirá um fluxo de execução análogo aos 6 passos acima, mas nesse caso, como o *parâmetro efetivo* é `N-k`, o *parâmetro formal* de `fat(.)` será iniciado com tal valor (i.é., na prática, no início de `fat(.)`, existe uma instrução ímplicita do tipo `n = N-k`).

Uma vez entendido como é executado uma chamada á função, podemos novamente comparar com o conceito usual de função matemática. Existe a declaração da função, com seu parâmetro formal

```
fat : IN -> IN  (nome 'fat', com domínio e imagem nos naturais)
fat(n) = { 1, se n=0; n x fat(n-1), se n>0 }
```

E existe o uso da função, como em

```
C(N,k) = fat(N) / (fat(k) x fat(N-k)
```

5. Usando o conceito de função para desenvolver um algoritmo: usar funções para "quebrar" a complexidade do algoritmo

Uma aplicação interessante das 3 vantagens acima citadas é possibilitar **quebrar** o problema, para isso deve-se tentar perceber alguma estrutura do problema que permita quebrá-lo em tarefas menores (e mais simples!). Esta abordagem é conhecida em Matemática como a técnica dividir para conquistar.

Essa técnica permite resolver problemas complexos como determinar a melhor maneira para mover os discos no problema da Torre de Hanói.

Para uma explicação mais completa da técnica de *divisão e conquista*, podemos desenhar um *algoritmo* de modo *indutivo* (que poderá resultar em um *algoritmo recursivo*) para resolver um problema de *contagem/frequência*: determinar quantas ocorrência de determinado valor *x* existem em um vetor *Vet*.
Para um iniciante na *arte de programar* o raciocínio abaixo (os 4 itens) soará um tanto quanto abstrato, então sugerimos que o estude de modo bastante atento, eventualmente revendo-o mais de uma vez. Se você já estudou no *Ensino Médio*, lembre-se do *processo de indução finita*, é ele que fundamenta o raciocínio.

A. **Supor um algoritmo que resolve o problema**.
   Podemos supor a existência de uma função de nome *CO* que conta as ocorrências de *x* no vetor *Vet*, com *N* elementos (desde `Vet[0]` até `Vet[N-1]`.
   Suporemos a função com 4 parâmetros, além de *x* e *Vet*, o índice inícial e final para contar, ou seja, `CO(x,ini,fim,Vet)` deve devolver o número de ocorrências de *x* entre `Vet[ini]` e `Vet[fim]`.
B. **Quebrando o problema**.
   Desse modo, para conseguir obter o total de cópias de *x* em *Vet*, poderíamos invocar *CO* para as 2 metades do vetor, isto é,
   supor *ce* receber o total de ocorrência na primeira parte (`ce = CO(x,0,N/2,Vet)` e
   supor *cd* receber o total de ocorrência na segunda parte (`cd = CO(x,N/2+1,N-1,Vet)`.
C. **Conquistando a solução global**.
   Então, podemos devolver *ce+cd* como o total de ocorrências de *x* em *Vet[0..N-1]*, ou seja `CO(x,0,N-1,Vet)` deve devolver `ce + ce`,
   pois `CO(x,0,N-1,Vet)` tem que coincidir com `CO(x,0,N/2,Vet) + CO(x,N/2+1,N-1,Vet)`! Essa é a resposta final.
D. **Detalhando a solução global**.
   Para conseguir codificar uma solução completa, bastaria cuidar dos detalhes de quando não é necessário dividir: quando *ini=fim*, a resposta é *1*, se *x = Vet[ini]*, e *0* em caso contrário.
   Ou seja, o algoritmo completo em *Portugol* poderia ser:
   inteiro CO (x, ini, fim, Vet) { inteiro meio; se (ini==fim) se (x == Vet[ini]) devolver 1; devolver 0; // Note que o algoritmo NAO chega aqui se ini==fim e a cada chamada |fim-ini| eh um numero menor! meio = (ini+fim)/2; // divisao inteira devolver CO(x, ini, meio, Vet) + CO(x, meio+1, fim, Vet); // chama a propria funcao CO => recursividade }

As funções que chamam a si própria recebem o nome recursivas, essa técnica de programação será explicado mais ao final do curso. Mas vale fazer nova analogia com matemática, pois existe uma função matemática, examinada no Ensino Médio, cuja definição é intrinsecamente recursiva, a **função fatorial**: fatorial de *n* é *1* se *n=0*, senão é *n* vezes o fatorial de *n-1*, ou seja,

*fat(n) = { 1, se n=0; n*fat(n-1), se n>0 }.*

Se você dispuser de mais um tempo para estudo, experimente implementar a função fatorial, na linguagem que está aprendendo, de modo *recursivo* (seguindo precisamente sua definição

acima).
A seguir veja como implementar a função fatorial de modo *iterativo*.

6. Exemplos concretos em *C* e em *Python* - cuidado com a organização de seu código

Uma questão importante para o uso de *funções* (ou *procedimentos*) é a **boa organização de código**:

1. Mais importante é reconhecer a repetição de código e organizá-lo como função/procedimento, desse modo seu código fica menor (e mais fácil de manter);
2. Depois deve-se procurar declarar as funções **por ordem de chamada**, ou seja, se seu código usualmente invoca primeiro `f1(.)`, depois `f2(.)` e por último `f3(.)`, então deveria declarar nessa ordem (primeiro `f1(.)` e assim por diante).
   Entretanto, existem códigos maiores e mais complexos, nesses casos tente **organizar por assunto**, deixando próximas as funções que tratam do mesmo *assunto*.

Para examinar exemplos concretos usando a linguagem *C* *clique* aqui.
Para examinar exemplos concretos usando a linguagem *Python* *clique* aqui.

Para ilustrar o uso e sintaxe de funções em *C* e em *Python*, examinemos um exemplo em que implementamos uma função para cômputo do fatorial de um natural *n*, mas fazendo-o via *algoritmo iterativo*. Vamos denominar a função por *fat* e é razoável que ela tenha um único *parâmetro (formal)*, ou seja, deverá ser invocada como `fat(0)` ou `fat(8)`.

C

```c
int fat (int n) { // define funcao com 1 parametro
  int ft = 1, i=1;
  while (i < n) {
    i = i + 1; // poderiamos tambem usar "i++"
    ft = ft * i; // poderiamos tambem usar "ft *= i"
    }
  return ft;
  } // chave indica final da funcao 'fat(...)'
...
// supondo existir neste contexto variaveis: N e k
printf("Combinacao = %f\n", (float)fat(N) / (fat(k) * fat(N-k)));
```

Python

```python
def fat (int n) : # define funcao com 1 parametro
  ft = 1; i=1
  while (i < n) :
    i = i + 1; # poderiamos tambem usar "i += 1"
    ft = ft * i; # poderiamos tambem usar "ft *= i"
  return ft;
  # Final da funcao 'fat(...)' - em Python,
  # basta a indentacao da proxima linha ser recuada
...
# supondo existir neste contexto variaveis: N e k
```

```
print("Combinacao = %f" %
(fat(N) / (fat(k) * fat(N-
k)))
```

Cód. 1. Exemplo de código para computar combinação de n, k-a-k, com uma função para fatorial.

6.1 Variáveis locais

Note na linha 2 do código acima que são declaradas duas novas variáveis, *ft* e *i*, dentro da fução *fat*. Isso significa que as variáveis *ft* e *i* são **variáveis locais** à função, ou seja, pode-se utilizar variáveis com os mesmos nomes em outras funções sendo que elas não terão qualquer relação. Em particular, na linha 11 dos dois exemplos (no código 1), poderia-se usar uma variável com nome *n*, *ft* ou *i* e ainda assim, está variável não teria qualquer relação com as correspondentes da função *fat*.

6.2 Para que serve função?

Suponhamos que o objetivo seja conseguir um código para computar a combinação de *n* elementos, tomados *k* a *k*, que representaremos por *C(n,k)*. Como sabemos do *Ensino Médio*, *C(n,k) = n! / (k! (n-k)!)*.

Nesse contexto, tente esquematizar uma implementação **sem** o uso de funções! O núcleo desse código deverá ter cerca de 12 linhas, pois precisariamos repetir as linhas 2 a 5 (ou 6) três vezes, uma para *fat(n)*, uma para *fat(k)* e outra para *fat(n-k)*. Por outro lado, usando função (no código 1) foram necessárias apenas 11 linhas no total! Mais ainda, o código 1 é certamento mais claro que a versão sem funções.

Portanto, o uso de função simplifica o código. Mas existe uma outra razão (não tão óbvia) para usar funções: a maior facilidade para escrever um programa (como explorado na seção 5). E isso se deve à vários fatores, mas principalmente à quebra de um problema maior em outros menores.

Portanto, desenvolver códigos quebrando-o em funções menores facilita o desenvolvimento e reduz a incidência de erros de programação, pois, pode-se implementar cada função separadamente, testando-a até que ela fique pronta e sem erros.

Resumidamente, usar funções: resulta em códigos mais fáceis de serem entendidos, mais fáceis de serem testados e, muito possivelmente viabilizando a obtenção do próprio algoritmo (que sem a "quebra" em funções menores e mais simples, seria muito mais difícil).

Leônidas de Oliveira Brandão
http://line.ime.usp.br

Alterações