Fundamentos de Programação

Filipe Francisco

19 de abril de 2018

Problemas

- Vetores estão muito ligados a loops
  - leitra dos elementos: loop
  - trabalhar com os elementos do vetor: loop
  - imprimir os elementos: loop
- Sugestão: crie um loop apenas para ler os elementos do vetor!
- Coloque as demais ações em outros loops, separados da leitura!
- Loop para preencher um vetor V de n inteiros:
  - **for**(i=0;i<n;i++) {
    scanf("%d",&V[i]);
    }
- Lembrete: um loop para percorrer o vetor deve trabalhar sobre V[i], e não sobre V[n]!
  - n é o tamanho do vetor
  - as posições deste vetor, portanto, são 0,1,...,n-1
  - ou seja, a posição V[n] não existe, está fora do vetor!

Problemas

- Soldados pequenos e grandes:
  - caso de teste: 3 soldados de alturas 1.70, 1.80 e 1.60
  - problema: usando **float**, a média é 1.70, mas o primeiro soldado é considerado grande!
    - ou seja, o compilador diz que a altura 1.70 é maior que a média 1.70!
  - este problema acontece devido à precisão do float
    - o float tem precisão apenas até a 7a casa decimal
  - para resolver este problema, em vez de float, basta utilizarmos **double**!
- Variável **double**:
  - tipo de variável para números reais com maior precisão que o float
  - precisão até a 16a casa decimal!
- Lendo e imprimindo um double:
  - scanf("%lf",&val);
  - printf("%lf",val);
- Lembra do **long long int**, que aumentava o tamanho do inteiro?
- A ideia é a mesma, mas aplicada a números reais

Problemas

- Tudo em ordem?
  - para i de 0 a $n-1$, comparar $V[i]$ com $V[i+1]$
  - a ideia é correta, mas tem um problema: quando $i = n-1$, estamos de fato comparando $V[n-1]$ com $V[n]$!
    - ou seja, comparando o último elemento a uma posição fora do vetor!
  - para resolver isso, basta eliminarmos esta última comparação
  - ou seja, fazemos um loop para i de 0 a $n-2$
- Um dos exercícios trata de um problema muito conhecido: ordenar um vetor
  - vetor ordenado: cada elemento é menor ou igual ao elemento seguinte
  - este problema é muito conhecido por ser um problema com muitas aplicações
    - ex: fila de espera por prioridade
    - ex: ordenar livros em ordem alfabética
  - a questão recomenda fazer o Bubble Sort, mas saibam que há vários métodos para ordenar um vetor

Funções

- Considerem um problema em que temos que aplicar repetidas vezes um mesmo método que já montamos
  - ex: ordenar três vetores individualmente
  - ex: calcular o número binomial a seguir (três fatoriais):

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Pergunta: vamos ter que criar cópias do código para fazer isso várias vezes?
- A resposta é NÃO!
- Temos uma maneira de montarmos um código apenas uma vez, e aplicar este código quantas vezes quisermos
- Para isso, utilizamos **funções**!
  - inclusive, vale dizer que já usamos função a algum tempo!

Funções

- Também chamada de subrotina ou procedimento
- O que é uma função?
  - é uma unidade de código definido pelo usuário para resolver uma tarefa específica
    - ex: ordenar um vetor, calcular um fatorial
  - esta função é chamada durante a execução do código
  - uma vez chamada, a função executa seu código e produz um resultado
- Uma função muitas vezes recebe algumas informações para trabalhar
- Uma função também pode retornar um resultado
  - este resultado é algum tipo de variável, como int, char, ...

Funções

- Como definir uma função?

- Uma função é definida por quatro informações:
  - tipo de retorno: tipo de informação retornada pela função
    - faz referência a algum tipo de variável
    - também chamado de saída da função
  - identificador: "nome" da função
    - as chamadas para a função utilizarão este nome
  - argumentos: valores/dados passados para a função trabalhar sobre eles
    - também chamado de entrada da função
  - código: código a ser executado pela função ao ser chamada

Funções

- Como criar uma função?

- Pseudocódigo:
  - função *nome* (*argumentos*) {
    [*código*]
    }
    - *nome*: identificador da função
    - *argumentos*: entrada da função
    - *código*: código da função
    - obs: em pseudocódigo, não é necessário colocarmos o tipo de retorno!

Funções

- Como criar uma função?

- C:
  - *tipo nome* (*argumentos*) {
    [*código*]
    }
    - *tipo*: tipo de retorno da função
    - *nome*: identificador da função
    - *argumentos*: entrada da função
    - *código*: código da função

Funções

- Como criar uma função?

- Exemplo: função para calcular o fatorial de um número
- A utilidade da função pode ser apresentada por sua entrada e saída
  - entrada: valor inteiro *n*
  - saída: valor do fatorial de *n*
- A partir disto, podemos montar uma função fat
- Pseudocódigo:
  - função fat (inteiro n) {
    inteiros i, f=1
    para *i* de 1 a *n* passo 1 faça {
    f=f*i
    }
    retorne f;
    }

Funções

- Como criar uma função?

- Exemplo: função para calcular o fatorial de um número
- A utilidade da função pode ser apresentada por sua entrada e saída
  - entrada: valor inteiro *n*
  - saída: valor do fatorial de *n*
- C:
  - ```
    int fat (int n) {
        int i, f=1;
        for(i=1;i<=n;i++) {
            f=f*i;
        }
        return f;
    }
    ```
    ```

Funções

- Uma vez que temos a função fat, podemos utilizá-las quantas vezes quisermos!
  - ex: a=fat(5);
  - ex: b=fat(7);
  - ex: c=fat(x);
- Restrições:
  - o tipo de retorno da função fat é inteiro, logo a função deve retornar uma variável inteira (ou um valor inteiro)
    - se retornarmos algum outro tipo, pode haver um erro!
  - nos exemplos acima, as variáveis a, b e c devem ser do mesmo tipo do retorno da função fat
    - ou seja, neste caso, a, b e c devem ser inteiros!
  - os argumentos passados para a função devem ser do mesmo tipo, e estarem na mesma ordem, que os argumentos da função fat
    - se passarmos um valor char como argumento, pode haver um erro!
  - o nome da função não pode ser uma palavra reservada da linguagem
    - o ideal é ter um nome diferente de todas as variáveis utilizadas

Funções

- Uma vez que temos a função fat, podemos utilizá-las quantas vezes quisermos!
  - ex: a=fat(5);
  - ex: b=fat(7);
  - ex: c=fat(x);
- Restrições:
  - o tipo de retorno da função fat é inteiro, logo a função deve retornar uma variável inteira (ou um valor inteiro)
    - se retornarmos algum outro tipo, pode haver um erro!
  - nos exemplos acima, as variáveis a, b e c devem ser do mesmo tipo do retorno da função fat
    - ou seja, neste caso, a, b e c devem ser inteiros!
  - os argumentos passados para a função devem ser do mesmo tipo, e estarem na mesma ordem, que os argumentos da função fat
    - se passarmos um valor char como argumento, pode haver um erro!
  - o nome da função não pode ser uma palavra reservada da linguagem
    - o ideal é ter um nome diferente de todas as variáveis utilizadas

Funções

- Por que dissemos que já usávamos funções?
  - porque a "**int** main" é uma função!
- Uma função também pode chamar a outra função
  - o exemplo mais básico disto é criarmos uma função e chamarmos a esta função dentro da main
- Podemos ter três funções f1, f2 e f3, com:
  - f1 chamando f2
  - f2 chamando f3
- Para isto, em C, devemos definir primeiro f3, depois f2, e por último f1
  - uma função só pode ser chamada se já estiver definida
- Pergunta: como fazer f1 chamar f2, e f2 chamar f1?

Funções

- Resposta: usando protótipo!
- Um protótipo é um esquema da função
  - este esquema contém tipo de retorno, nome da função, e argumentos recebidos
- O protótipo é colocado ao início do algoritmo para indicar ao compilador que vai existir uma função com aquelas informações
  - por este motivo, não se inclui o código
- Exemplo: protótipo da função fat (que já montamos)
  - **int** fat (**int** n);
    - isto indica ao compilador que haverá uma função fat que recebe um inteiro e retorna um inteiro, que terá seu código feito posteriormente

Funções

- Suponha que você passa uma certa variável *num* para a função
- Quando enviamos algo como argumento de uma função, o compilador "cria" uma variável para usarmos dentro da função e armazena o valor passado nesta variável criada
  - se a função for "**int** fat(**int** n)", a variável a ser utilizada dentro da função é n
  - se a função for "**int** fat(**int** numero)", a variável a ser utilizada dentro da função é numero
- Portanto, se alterarmos o valor desta variável dentro da função, alteramos seu valor apenas dentro da função, e não fora!
- Além disso, em C, uma função não pode acessar variáveis que estão fora dela mesma!
  - por exemplo, suponha que f1 tenha variáveis *a* e *b*, e f2 tenha variáveis *c* e *d*
  - portanto, em f1 não podemos acessar as variáveis *c* e *d*, e em f2 não podemos acessar as variáveis *a* e *b*

Funções

- Esta ideia de variáveis "dentro" e "fora" da função é a definição de escopo da variável
  - escopo: "área" onde a variável pode ser acessada
- Em C, ao criarmor uma variável dentro de uma função, esta variável pode ser acessada somente por esta função
  - dizemos que esta é uma variável local
  - o escopo da variável é a própria função onde ela foi criada
- O escopo nos abre a possibilidade de termos variáveis de mesmo nome em funções diferentes
  - podemos ter uma variável *num* em f1, outra em f2, outra na main, ...
- E apesar de cada variável ter seu escopo, também podemos fazer com que uma variável possa ser acessada por qualquer função
- Como fazer isso?

Funções

- Resposta: variável global!
- Uma variável global é uma variável declarada fora de qualquer função
  - como ela não está dentro de nenhuma função, o escopo dela não é uma função em específico, mas sim o algoritmo inteiro!
- Criar uma variável global é simples!
  - basta criar uma variável como já fazemos, mas fora de funções!
- Deste modo, ela pode ser acessada por todas as funções que criarmos
  - main, f1, f2, ...
- E como todas as funções podem acessar esta variável, todas as funções também podem alterar seu valor!
- Obs: se tivermos uma variável local com o mesmo nome de uma variável global, a função utilizará a variável local!
  - para acessar uma variável, a função busca uma variável local, depois uma variável global
  - se não encontrar nenhuma, retorna um erro