Pseudo algoritmo

Algoritmos, Estruturas de Dados, Programação Imperativa e C

Sérgio Soares
scbs@cin.ufpe.br

Tipos

- Tipos
  - `int`
    `1, 16, -234`
  - `real`
    `34.45, -87.92`
  - `bool`
    `V, F`
  - `char`
    `'a', 'b'`
  - `string`
    `"casa", "livro"`

Tipos

- Definindo tipos
  `tipo nome :: lista_valores;`
- Exemplo
  `tipo vogais :: 'a', 'e', 'i', 'o', 'u';`

Registros

- Definição
  `reg` nome {lista_declarações};
- Exemplo

```
reg pessoa {int rg, string nome};
pessoa p;
p.rg <- 1765;
p.nome <- "Maria";
```

Operações

- `int`
  `+, -, *, /, div, mod`
- `real`
  `+, -, *, /`
- Relações
  `=, ≠, <, ≤, >, ≥`
- `bool`
  `e, ou, não`
- `char`
  `=, ≠`
- `string`
  `+`

Funções

- `div` e `mod`
  `div(10/3) = 3`
  `mod(10/3) = 1`
- `trunc`
  `trunc(1.8) = 1`
- `arred`
  `arred(1.4) = 1`
  `arred(1.5) = 2`
- `tier`
  `tier(9) = 9.0`
- `ord`
  `ord('a') = 65`
  `ord('b') = 66`
  `ord('c') = 67`

Programa

- Definição

  `programa` nome

  declarações_variáveis

  comandos

Variáveis

- Declaração

  tipo nome;

  tipo nome1, nome2, ..., nomeN;

  tipo nome <- valor;

```
int x, y;
string aluno <- "Luis";
```

Comandos

- Atribuição

  variável <- expressão;

- Entrada

  `leia`(lista_variáveis);

- Saída

  `escreva`(lista_variáveis);

Comandos

- Condicional simples

  `se` condição `então`

  comando

```
int x;
leia(x);
se mod(x,2)=0 então
    escreva("A entrada é par");
```

Comandos

- Condicional simples

  `se` condição `então`

  comando

  `senão`

  comando

```
se mod(x,2)=0 então
    escreva("A entrada é par");
senão
    escreva("É impar");
```

Exercícios

1 - Escreva um programa que lê dois inteiros e imprime o maior.

2 - Escreva um programa que lê três inteiros e os imprime em ordem crescente.

Comandos

- Iteração (repetição) com teste no início
  `enquanto` condição `faça`
    comando

```
int i <- 1;
enquanto i < 10 faça
    escreva(i);
    i <- i + 1;
```

Comandos

- Iteração (repetição) com teste no início
  `para` variável `de` valor_inicial `incr` valor_incremento `até` valor_final `faça`
    comando

```
int i;
para i de 1 incr 1 até 10 faça
    escreva(i);
```

Comandos

- Iteração (repetição) com teste no fim
  `repita`
    comando
  `até (que)` condição

```
int i;
repita
    escreva(i);
    i <- i + 1;
até que i < 10 faça
```

Comandos

- Escape
  `saia`
  `escape`

```
int i;
repita
    escreva(i);
    i <- i + 1;
    se i=4 então
        saia;
até que i < 10 faça
```

Comandos

- Seleção
  `conforme` variável `faça`
    lista_dos_casos

```
imprima(“Entre com um valor”);
leia(x);
imprima(“Escolha 1 (dobro) ou 2 (metade)”);
leia(i);
conforme i faça
    1: x <- 2*x;
    2: x <- div(x,2);
escreva(“O resultado é ”, x);
```

Exercícios

3 - Escreva um programa que lê dois inteiros e imprime os número no intervalo formado pelos mesmos. Os valores lidos tem de expressar um intervalo crescente.

4 - Altere o programa para que o mesmo aceite quaisquer intervalos imprimido na ordem crescente ou decrescente, conforme os valores dados. (obrigatório)

Procedimentos

- Sem retorno (rotinas)

  `proc` nome`(`lista_parâmetros`)`

  declarações_variáveis

  comandos

```
proc par(int a, bool resposta)
    se mod(a,2)=0 então
        resposta <- T;
    senão
        resposta <- F;
```

Procedimentos

- Execução de rotina

  `execute` nome_rotina`(`lista_argumentos`);`

```
int a;
bool x;
leia(a);
execute par(a, x);
se x então
     imprima(“É par”);
senão
     imprima(“É ímpar”);
```

Procedimentos

- Com retorno (funções)

  `proc` tipo nome`(`lista_parâmetros`)`

  declarações_variáveis

  comandos

```
proc bool par(int a)
    se mod(a,2)=0 então
        retorne T;
    senão
        retorne F;
```

Procedimentos

- Execução de função

  nome_função`(`lista_argumentos`);`

```
int a;
leia(a);
se par(a) então
    imprima(“É par”);
senão
    imprima(“É ímpar”);
```

Programa

- Definição completa

  `programa` nome

  declarações_variáveis

  comandos

  procedimentos

Exercício

5 - Escreva um programa que lê três inteiros e imprime o menor e o maior. O programa deve definir e utilizar os procedimentos `menor` e `maior` que dado dois números retornam, respectivamente, o menor e o maior deles.

Exercício

6 - Escreva uma aplicação bancária. O programa deve exibir um menu com as opções: creditar, debitar, saldo, e sair. O programa deve manter uma variável `conta` do tipo `real` que é alterada segundo a opção escolhida. Caso a opção seja creditar ou debitar, o programa deve ler o valor a ser adicionado ou removido. (obrigatório)

Arrays

Algoritmos, Estruturas de Dados, Programação Imperativa e C

Sérgio Soares
scbs@cin.ufpe.br

Arrays

- Armazenam elementos de um determinado tipo em seqüência.
- Também chamado de vetor
- Declarando um *array*
  `int[tamanho] arr;`
- Acessando o i-ésimo elemento do *array*
  `arr[i];`

Arrays

- O primeiro índice é `0` e o último é tamanho`-1`

```
int[4] arr;
int i <- 0;
enquanto i<5 faça
    arr[i] <- (2*i);
    i <- i +1
```

arr[0]	[1]	[2]	[3]	[4]
0	2	4	6	8

Exercício

1 - Escreva um programa que lê uma seqüência de 10 números e os armazena em um *array*. Em seguida o programa imprime o produto e a soma destes números.

Arrays

- Recebendo um *array* como parâmetro em um procedimento
  - A dimensão não precisa ser informada

```
proc tipo_procedimento nome(tipo_array[] variável)
    declarações
    comandos
```

Exercício

2 – Escreva uma rotina que recebe um *array* de inteiros ordenado e um número inteiro. A rotina deve criar um novo *array*, copiar os valores do *array* recebido para o novo, inserindo o número recebido na posição correta de modo a manter o *array* resultante ordenado, e retornar o *array* criado.

Exercício

3 - Altere o exercício 6 da aula anterior para que o mesmo utilize um *array* de reais de tamanho 10 que simbolizam contas do banco. Os números das contas serão os índices do *array* (0..9). Ao iniciar o programa todas as contas iniciam a execução com saldo `0.0`. (obrigatório)

Arrays multidimensionais

- Também chamados de matrizes
- Declaração
  - int[2][2] matriz;
- Inicialização

```
matriz[0][0] <- -10;
matriz[0][1] <- 30;
matriz[1][0] <- 2;
matriz[1][1] <- -15;
```

-10	**30**
2	**-15**

Exercício

4 – Escreva um programa que lê valores de uma matriz 2x2 de inteiros. Em seguida calcule a soma das diagonais da matriz.

5 – Escreva um programa que soma as diagonais de matrizes quadráticas de qualquer dimensão. O programa deve no início pedir a dimensão da matriz e ler os valores.

Exercício

6 – Escreva um programa que verifica se uma matriz quadrática é simétrica.

7 – Escreva um programa que lê valores de duas matrizes quadradas (de mesma dimensão) e retorna o produto das mesmas. (obrigatório)

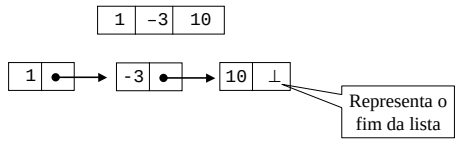
Listas

Algoritmos, Estruturas de Dados, Programação Imperativa e C

Sérgio Soares
scbs@cin.ufpe.br

Listas

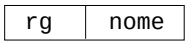
- Armazenam um conjunto de dados com uma ordem relativa entre si
- Semelhante a um *array*/vetor



Registros

- Definição
  `reg` nome {lista_declarações};
- Exemplo

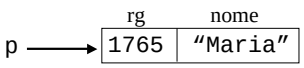
```
reg pessoa {int rg, string nome};
```



Registros

- Alocação dinâmica de memória
  - Comando `aloque`

```
reg pessoa {int rg, string nome};
pessoa p;
aloque(p);
p.rg <- 1765;
p.nome <- "Maria";
```



Exercício

1 – Escreva um programa que define um registro `cliente` que contém as informações de um cliente de uma loja (nome, cpf, endereço e renda mensal). O programa deve definir um *array* de `cliente` com um tamanho fornecido pelo usuário e carregar os elementos (`cliente`) no *array* com os dados fornecidos pelo usuário.

Listas com registros

```
reg listaInt {int dado, listaInt prox};
int i <- 0;
listaInt cabeca, aux;
aloque(cabeca);
aux <- cabeca;
enquanto i<10 faça
    aux.dado <- 2 * i;
    aloque(aux.prox);
    aux <- aux.prox;
    i <- i + 1;
aux.prox <- ⊥;
...
```

Exercícios

2 – Crie um programa com uma lista de número inteiros e que insere números fornecidos pelo usuário na mesma até que o número 0 seja lido, indicando o fim do programa.

3 – Modifique o exercício 1 para que o mesmo não utilize um *array* de registros (`cliente`), mas uma lista de `cliente`.

Exercício

4 – Crie dois procedimentos. O primeiro recebe um número inteiro e uma lista ordenada com números inteiros. O procedimento deve inserir o número recebido na lista mantendo-a ordenada. O segundo procedimento recebe os mesmos parâmetros do primeiro mas remove da lista o número recebido.

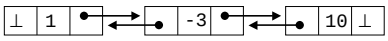
Exercício

5 – Modifique o exercício de aplicação bancária para que uma conta seja um registro de numero (`int`) e saldo (`real`). Altere a aplicação bancária para trabalhar com uma lista de conta! (obrigatório)

Listas duplamente encadeadas

- Listas que podem ser percorridas nos dois sentidos
- Exemplo

```
reg exemplo {exemplo ante,
             int dado,
             exemplo prox};
```




```
reg lstInt {lstInt ante, int dado, lstInt prox};
int i <- 0;
lstInt cabeca , auxA , auxB;
aloque(cabeca);
cabeca.ante <- ⊥
auxA <- cabeca;
enquanto i<10 faça
    auxA.dado <- 2 * i;
    aloque(auxB);
    auxA.prox <- auxB;
    auxB.ante <- auxA;
    auxA <- auxB;
    i <- i + 1;
auxA.prox <- ⊥;
...
```

Exercícios

6 – Modifique o exercício 2 para trabalhar com uma lista duplamente encadeada.

7 – Modifique o exercício 3 para trabalhar com uma lista duplamente encadeada.

8 – Modifique o exercício 4 para trabalhar com uma lista duplamente encadeada.

Fila e Pilha

Algoritmos, Estruturas de Dados, Programação Imperativa e C

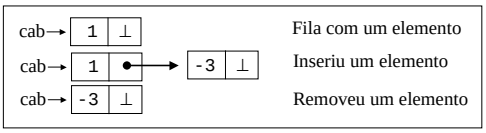
Sérgio Soares
scbs@cin.ufpe.br

Fila e Pilha

- Estruturas de dados com disciplina de acesso
  - Fila
    - FIFO (*first in first out)* - O primeiro elemento inserido é o primeiro a ser retirado
  - Pilha
    - LIFO (*last in first out)* - O último elemento inserido é o primeiro a ser retirado

Fila

- Uma lista onde:
  - os elementos são inseridos sempre no final
  - os elementos são acessados e retirados sempre da cabeça




```
reg pessoa {string nome, string cpf};
reg fila_pessoas {pessoa dado, fila_pessoas prox};

proc inserir(fila_pessoas fila, pessoa p)
    fila_pessoas aux, e;
    aloque(e);
    e.dado <- p;
    e.prox <- ⊥;
    se fila = ⊥ então
        fila <- e;
    senão
        aux <- fila;
        enquanto aux.prox ≠ ⊥ faça
            aux <- aux.prox;
        aux.prox <- e;
```

```
reg pessoa {string nome, string cpf};
reg fila_pessoas {pessoa dado, fila_pessoas prox};

proc remover(fila_pessoas fila)
    se fila ≠ ⊥ então
        fila <- fila.prox;

proc pessoa consultar(fila_pessoas fila)
    se fila ≠ ⊥ então
       retorne fila.dado;
```

Aplicações de Fila

- Armazenar procedimentos que esperam para executar
- Armazenar *jobs* de impressão
- Armazenar pacientes a serem atendidos

Exercícios

1 – Crie um programa que define uma fila de inteiros e insere números fornecidos pelo usuário na mesma até que o número 0 seja lido, indicando o fim do programa. Em seguida o programa imprime os elementos da fila.

Exercício

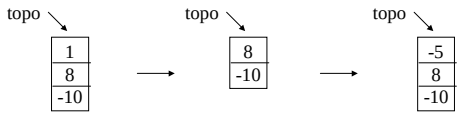
2 – Escreva um programa para consultórios que define um registro `paciente` que contém as informações de um paciente de uma clínica (nome, cpf, endereço). O programa deve definir uma fila de `paciente` para serem atendidos e as seguintes operações:

- inserir um paciente
- verificar o próximo a ser atendido
- atender um paciente

Obrigatório

Pilhas

- Estruturas com o sentido contrário de filas, ou seja, listas onde:
  - os elementos são inseridos, acessados, e retirados sempre da cabeça (topo)




```
reg pessoa {string nome, string cpf};
reg pilha_pessoas {pessoa dado, pilha_pessoas prox};

proc inserir(pilha_pessoas pilha, pessoa p)
    pilha_pessoas e;
    aloque(e);
    e.dado <- p;
    e.prox <- pilha;

proc remover(pilha_pessoas pilha)
    se fila ≠ ⊥ então
        pilha <- pilha.prox;

proc pessoa consultar(pilha_pessoas pilha)
    se fila ≠ ⊥ então
       retorne pilha.dado;
```

Exercícios

3 – Crie um programa que define uma pilha de inteiros e insere números fornecidos pelo usuário na mesma até que o número 0 seja lido, indicando o fim do programa. Em seguida o programa imprime os elementos da pilha.

Recursividade

Algoritmos, Estruturas de Dados,
Programação Imperativa e C

Sérgio Soares

scbs@cin.ufpe.br

Recursividade

- Capacidade de um procedimento chamar a si próprio

Fatorial sem recursividade

```
fatorial(1) = 1
fatorial(n) = n * fatorial(n-1)
```

```
proc int fatorial(int valor)
    int resposta;
    enquanto n > 0 faça
        resposta <- resposta * valor;
        valor <- valor - 1;
    retorne resposta;
```

Fatorial com recursividade

```
fatorial(1) = 1
fatorial(n) = n * fatorial(n-1)
```

```
proc int fatorial(int valor)
    se valor = 1 então
        retorne 1;
    senão
        retorne valor * fatorial(valor-1)
```

Exercício

1 – Crie o procedimento `fib` que calcula o fibonnaci de um número.

```
fib(2) = 1
fib(n) = fib(n-1) * fib(n-2)
```

Pesquisando se um elemento está em uma lista recursivamente

```
proc bool pesquisar(Lista_Int lista, int valor)
    se lista = ⊥ então
        retorne F;
    senão
        se lista.dado = valor então
            retorne V;
        senão
            retorne pesquisar(lista.prox, valor);
```

Exercícios

2 – Crie o procedimento `tam` que calcula recursivamente o tamanho de uma lista.

3 – Crie o procedimento `soma` que calcula recursivamente a soma dos elementos de uma lista.

Exercícios

4 – Crie o procedimento recursivo `raiz` que calcula a raiz quadrada de um número

5 – Crie o procedimento recursivo `inserir` que insere um elemento no final de uma lista. obrigatório