Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

GUIA DE CODIFICAÇÃO

Manual de Boas Práticas de Programação

SCC0202 -Algoritmos e Estruturas de Dados I

Professora: Sandra Maria Aluísio

PAE: Arineiza Cristina Pinheiro

São Carlos, 2010.

Guia de Codificação

Manual de Boas Práticas de Programação

Sumário *pág*

Autoria: Arineiza Cristina Pinheiro

1. Padronização de idioma

Definir o idioma a ser adotado durante toda a implementação do programa (inglês ou português). Uma vez definido, todo o programa deve refletir a escolha adotada, sendo válida para: nome de variáveis, funções, constantes, comentário etc.

Muitas vezes opta-se por usar os nomes em inglês para TADs clássicos como Pilha, Fila, Lista para ficarem similares aos nomes usados na literatura clássica (veja o caso do TAD pilhas dado em sala). Entretanto, a prática de padronização de idioma, com a utilização do português, poderia ser adotada, inclusive para apoiar uma política de soberania de língua .

2. Nomenclatura de Variáveis

O nome das variáveis deve seguir o modelo: “iNomeVariavel”.

- A primeira letra (em minúsculo) do nome da variável corresponde a primeira letra do seu tipo. No exemplo, a variavél é do tipo ***int***. Para os demais tipos, vide Tabela 1.

Tabela 1 – Letras que definem o tipo das variáveis

Tipo	Letra
int	i
float	f
double	d
char	c

- Quando houver modificadores, concatenar a primeira letra (em minúsculo) do modificador antes da letra que define o tipo. Por exemplo, “uiNomeVariavel” seria do tipo ***unsigned int***. As letras para cada modificador são definidas na Tabela 2.

Tabela 2 – Letras que definem o modificador aplicado ao tipo da variável.

Modificadores	Letra
unsigned	u
short	s
long	l

- Se a variável for um ponteiro, adicionar a letra (em minúsculo) “p” antes das letras que definem seu tipo. Por exemplo, “puiNomeVariavel” é um ponteiro para uma variável de tipo ***unsigned***

int.

- As palavras que definem nome propriamente dito da variável devem ser escritas com a primeira letra em maiúsculo e as demais em minúsculo. Por exemplo, “puiTamanhoVetor”.
- O nome das variáveis não deve conter caracteres especiais, “ç” e acentos, para evitar problemas com diversos editores.
- O nome das variáveis sempre deve ser SIGNIFICATIVO. Evitar nome de variáveis como: “aux” ou “i”. Prefira “iIterador” para listas e vetores, “iLinha” e “iColuna” para matrizes.
- Quando o tipo de uma variável for um novo tipo criado pelo comando `typedef`, não é necessário aplicar os modificadores. Apenas aplicar a letra minúscula “p” caso ela seja um ponteiro.
    - Ex.: Matriz* pMatriz

3. Nomenclatura de Funções

O nome das funções deve seguir o modelo: NomeBiblioteca_NomeFuncao().

- A primeira parte do nome da função é definida pelo nome do arquivo .h em que está inserida, sendo o padrão de primeira letra em maiúsculo e as demais em minúsculo, para cada palavra que compõe o nome da função.
    - Essa prática auxilia o programador e o usuário da biblioteca que utilizam ambientes como Eclipse e Code-Blocks, uma vez que disponibilizam a função “auto-completar” para nome de variáveis e funções, além de deixar claro a qual biblioteca pertence cada função.
- Separar o nome da biblioteca do nome da função com um “_” (*underline*).
- O nome da função deve seguir o mesmo padrão adotado para o nome da biblioteca, visando ser auto-explicativo sobre qual a funcionalidade implementada.
- Evitar caracteres especiais, “ç” e acentos, pelo mesmo motivo descrito anteriormente no tópico sobre Nomenclatura de Variáveis.
    - Ex.: Matriz.h → Matriz_CriaMatrizVazia

4. Nomenclatura de Constantes

O nome das constantes deve ser apresentado todo em maiúsculo, podendo ser abreviado.

- Por exemplo: `#define TAM 100 //tamanho`

O nome das constantes **de erro** deve seguir o modelo: ERRO_NOME (Ver tópico 8 - Definição de Biblioteca de Erros).

- A palavra “erro” deve aparecer em maiúsculo no início do nome da constante, seguida de “_”(*underline*) e o nome que descreve o erro, também em maiúsculo.
  - Por exemplo: `#define ERRO_PONTEIRO_NULO 1`

5. Comentários

- Criar comentários para expliar trechos pouco claros do código ou com alta complexidade.
  - Na dúvida? Comente!
- Comentários devem ser feitos no idioma definido no início do projeto, de modo a manter a uniformidade do texto.

6. Cabeçalho de Funções

Definir cabeçalhos para cada função da biblioteca, que deveram ser reproduzidos no .h e também no .c . No cabeçalho deve conter as informações:

- @Nome: copiar o nome da função
- @Descricao: em que é descrito rapidamente qual a funcionalidade implementada
- @Parametro: descreve quais os parâmetros da função
  - Para efeito de esclarecimento, sinalizar quais são de entrada, saída, ou entrada/saída, colocando os indicadores _E_ (_IN_), _S_ (_OUT_) e _ES_ (_IO_) respectivamente, antes do nome dos parâmetros (opcional).
- @Retorno: descrevendo qual o tipo de retorno da função, quando for diferente de *void*.
- @Erro: quando há flags de erro, descrever quais os possíveis erros a serem retornados.

```
/* ********************************************************************************* */
/* @Nome: Matriz_ZeraElemento(Matriz *matriz, int iLinha, int iColuna, int* piFlagErro)*/
/* @Descricao: função que zera o elemento (iLinha, iColuna) da matriz                  */
/* @Parametro: _E_ *matriz  -  matriz a ser manipulada                                 */
/*             _E_ iLinha  -  corresponde ao número da linha do elemento               */
/*             _E_ iColuna  -  corresponde ao número da coluna do elemento             */
/*             _S_ iFlagErro  -  retorna o estado final da função                      */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                                */
/*        ERRO_ENDERECO_INVALIDO  -  elemento fora do limite da matriz                 */
/* ********************************************************************************* */
```

- Essa prática auxilia no entendimento da biblioteca e reforça a documentação do projeto, deixando claro ao usuário o contrato de uso da biblioteca.

7. Criação de Construtor/Destrutor

Toda biblioteca necessariamente deverá possuir duas funções padrão: um Construtor e um Destrutor.

- No Construtor, toda inicialização de variável da biblioteca (alocação de memória) deve ser concentrada. Esse procedimento facilita a destruição (desalocação) de variáveis, que deve ser realizada na função Destrutor.
  - Para cada chamada do Construtor, necessariamente ao final do uso da variável ou final do programa deverá ter uma chamada do Destrutor.
- O nome das funções Construtor e Destrutor pode ser adaptado ao contexto do programa, devendo apenas seguir o padrão já descrito no tópico 3 – Nomenclatura de Funções.
  - Por exemplo: Matriz_CriaMatrizVazia
- No cabeçalho da função, no item "@Descrição" descrever que a função corresponde ao Construtor/Destrutor da biblioteca (ver tópico 5 – Cabeçalho de Funções).
  - Por exemplo:

```
/* @Nome: Matriz_CriaMatrizVazia(int iLinha, int iColuna, int* piFlagErro)          */
/* @Descrição: Construtor -  cria uma matriz vazia de dimensão (iLinha x iColuna)   */
```

8. Definição de Biblioteca de Erros

Definir uma biblioteca exclusiva para a difinição de erros para seu programa.

- Alguns códigos de erros (ou estado) são recorrentes na maioria dos programas, como: sucesso, acesso a ponteiro nulo, espaço de memória insuficiente.
- Criar um .h geral para todos tipos de erro, e sempre incluir na biblioteca a ser disponibilizada.
  - Facilita o entendimento do código por parte do desenvolvedor da biblioteca.
  - Disponibilizado o .h ao usuário, é possível consultá-lo como parte da documentação.
- Por exemplo:

```
#ifndef BIBLIOERRO_H_INCLUDED
#define BIBLIOERRO_H_INCLUDED

/*Biblioteca que define os tipos de erros possíveis*/
/*Disponível ao usuário - parte da documentação*/

#define ERRO_SUCESSO 0
#define ERRO_PONTEIRO_NULO 1
#define ERRO_ENDERECO_INVALIDO 2
#define ERRO_MEMORIA_INSUFICIENTE 3


#endif // BIBLIOERRO_H_INCLUDED
```

9. Verificação de Ponteiros

Toda variável ponteiro deve ser verificada antes de ser acessada.

- Exceto no Construtor, quando a primeira operação for uma alocação de memória (malloc, calloc...). Porém, verificar após a alocação se o ponteiro é diferente de nulo.
  - Caso seja nulo, retornar o erro ERRO_MEMORIA_INSUFICIENTE.
- Todo parâmetro passado à biblioteca deve ser verificado se é diferente de NULL, isso evita que o usuário utilize errado as funcionalidades disponibilizadas e garante maior robustez à biblioteca.
  - Caso o usuário passe um ponteiro nulo, retornar o erro ERRO_PONTEIRO_NULO.

10. Abstração e Encapsulamento

Na linguagem C, uma forma de se fazer abstração e encapsulamento de dados é a utilização de ponteiros opacos para tipos abstratos de dados (TADs). Um ponteiro opaco é um caso especial de tipo de dados opaco, um tipo de dados que é declarado como um ponteiro para um registro ou estrutura de dados de algum tipo não especificado.

Ponteiros opacos representam uma forma de esconder os detalhes da implementação de uma interface de clientes normais, de modo que a lógica da biblioteca poderá ser alterada sem a necessidade de recompilar os módulos que a utilizam. Esta característica é importante por fornecer compatibilidade, uma vez que diferentes versões de um biblioteca compartilhada fornecem uma mesma interface ao usuário final.

Para implementar um ponteiro opaco, define-se a interface da biblioteca em um arquivo “.h” que será desponibilizada ao usuário. Por exemplo:

```c
/* Matriz.h */

typedef struct Matriz *tMatriz;

/* ********************************************************************************** */
/* @Nome: Matriz_CriaMatrizVazia(int iLinha, int iColuna, int* piFlagErro)            */
/* @Descricao: Construtor -  cria uma matriz vazia de dimensão (iLinha x iColuna)     */
/* @Parametro: _E_ iLinha  -  corresponde ao número da linha do elemento              */
/*             _E_ iColuna  -  corresponde ao número da coluna do elemento            */
/*             _S_ piFlagErro  -  retorna o estado final da função                    */
/* @Retorno: *tMatriz  -  matriz a ser manipulada                                     */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                               */
/*        ERRO_MEMORIA_INSUFICIENTE  -  sem memória para alocar a matriz              */
/* ********************************************************************************** */
*tMatriz Matriz_CriaMatrizVazia(int iLinha, int iColuna, int* piFlagErro);

/* ********************************************************************************** */
/* @Nome: Matriz_Destroi(Matriz *pMatriz, int* piFlagErro)                            */
/* @Descricao: Destrutor - função que destroi a estrutura da matriz                   */
/* @Parametro: _E_ pMatriz  -  matriz a ser destruida                                 */
/*             _S_ piFlagErro  -  retorna o estado final da função                    */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                               */
/*        ERRO_PONTEIRO_NULO  -  ponteiro pMatriz nulo                                */
/* ********************************************************************************** */
void Matriz_Destroi(tMatriz *pMatriz);

/* ********************************************************************************** */
/* @Nome: Matriz_ZeraElemento(Matriz *pMatriz, int iLinha, int iColuna, int* piFlagErro)*/
/* @Descricao: função que zera o elemento (iLinha, iColuna) da matriz                 */
/* @Parametro: _E_ pMatriz  -  matriz a ser manipulada                                */
/*             _E_ iLinha  -  corresponde ao número da linha do elemento              */
/*             _E_ iColuna  -  corresponde ao número da coluna do elemento            */
/*             _S_ piFlagErro  -  retorna o estado final da função                    */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                               */
/*        ERRO_ENDERECO_INVALIDO  -  elemento fora do limite da matriz                */
/*        ERRO_PONTEIRO_NULO  -  ponteiro pMatriz nulo                                */
/* ********************************************************************************** */
void Matriz_ZeraElemento(tMatriz *pMatriz, int iLinha, int iColuna, int* piFlagErro);
```

A lógica e a definição explícita do tipo devem ser implementadas no arquivo “.c”:

```
/* Matriz.c */
#include <stdlib.h>
#include “BiblioErro.h”
#include “Matriz.h”

struct Matriz {
          int iLin;
          int iCol;
          float* fVet;
};

/* ********************************************************************************** */
/* @Nome: Matriz_CriaMatrizVazia(int iLinha, int iColuna, int* piFlagErro)            */
/* @Descricao: Construtor -  cria uma matriz vazia de dimensão (iLinha x iColuna)     */
/* @Parametro: _E_ iLinha  -  corresponde ao número da linha do elemento              */
/*             _E_ iColuna  -  corresponde ao número da coluna do elemento            */
/*             _S_ piFlagErro  -  retorna o estado final da função                    */
/* @Retorno: *tMatriz  -  matriz a ser manipulada                                     */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                               */
/*        ERRO_MEMORIA_INSUFICIENTE  -  sem memória para alocar a matriz              */
/* ********************************************************************************** */
*tMatriz Matriz_CriaMatrizVazia(int iLinha, int iColuna, int* piFlagErro){
    Matriz* pMatriz = (tMatriz*) malloc(sizeof(tMatriz));
    if (pMatriz == NULL) {
          *piFlagErro = ERRO_MEMORIA_INSUFICIENTE;
          return NULL;
    }
    pMatriz->iLin = iLinha;
    pMatriz->iCol = iColuna;
    pMatriz->fVet = (float*) malloc(iLinha*iColuna*sizeof(float));
    if(pMatriz->fVet == NULL){
     /*Se não há memória suficiente para matriz,
     desalocar a memória anteriormente alocada*/
        free(pMatriz);
        *piFlagErro = ERRO_MEMORIA_INSUFICIENTE;
        return  NULL;
    }
    *piFlagErro = ERRO_SUCESSO;
    return pMatriz;
}

/* ********************************************************************************** */
/* @Nome: Matriz_Destroi(Matriz *pMatriz, int* piFlagErro)                            */
/* @Descricao: Destrutor - função que destroi a estrutura da matriz                   */
/* @Parametro: _E_ pMatriz  -  matriz a ser destruida                                 */
/*             _S_ piFlagErro  -  retorna o estado final da função                    */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                               */
/*        ERRO_PONTEIRO_NULO  -  ponteiro pMatriz nulo                                */
/* ********************************************************************************** */
void Matriz_Destroi(tMatriz *pMatriz, int* piFlagErro){
    if((pMatriz->fVet != NULL) && (pMatriz != NULL)){
        free(pMatriz->fVet);
        free(pMatriz);
        *piFlagErro = ERRO_SUCESSO;
    }
    else{
         *piFlagErro = ERRO_PONTEIRO_NULO;
    }
 }
```

```c
/* ************************************************************************** */
/* @Nome: Matriz_ZeraElemento(Matriz *pMatriz, int iLinha, int iColuna, int* piFlagErro)*/
/* @Descricao: função que zera o elemento (iLinha, iColuna) da matriz                 */
/* @Parametro: _E_ pMatriz  -  matriz a ser manipulada                                */
/*             _E_ iLinha  -  corresponde ao número da linha do elemento              */
/*             _E_ iColuna  -  corresponde ao número da coluna do elemento            */
/*             _S_ iFlagErro  -  retorna o estado final da função                     */
/* @Erro: ERRO_SUCESSO  -  função executada com sucesso                               */
/*        ERRO_ENDERECO_INVALIDO  -  elemento fora do limite da matriz                */
/*        ERRO_PONTEIRO_NULO  -  ponteiro pMatriz nulo                                */
/* ************************************************************************** */
 void Matriz_ZeraElemento(tMatriz *pMatriz, int iLinha, int iColuna, int* piFlagErro){
     int iIndice; /* índice do elemento no vetor */
     if(pMatriz != NULL){
         if (iLinha<0 || iLinha>=pMatriz->iLin || iColuna<0 || iColuna>=pMatriz->iCol) {
             *piFlagErro = ERRO_ENDERECO_INVALIDO;
         }
        else{
             iIndice = (iLinha - 1) * pMatriz->iCol + iColuna;
             pMatriz->fVet[iIndice] = 0;
             *piFlagErro = ERRO_SUCESSO;
         }
     }
     else{
         *piFlagErro = ERRO_PONTEIRO_NULO;
     }
 }
```

Desta forma, o usuário receberá a biblioteca compilada (arquivo “.o”) e terá acesso apenas ao arquivo “.h” onde foi definido o ponteiro opaco para o TAD Matriz. Essa prática proporciona abstração e encapsulamento de dados e, portanto, deve ser adotada.

11. Identação

Identar o código com espaços (e não tab) preferencialmente.

- Alguns editores identam de forma ruim quando utiliza-se a tecla tab.
- Preferir utilizar quatro espaços em branco (sugestão).
- Evitar muito níveis de identação. No máximo 5, contando do nível inicial da função.
    - Verificar se não é possível quebrar em mais de uma rotina (função).