UNEB – Universidade do Estado da Bahia
Departamento de Ciências Exatas e da Terra
Colegiado de Sistemas de Informação
Linguagem de Programação – I
Prof. Cláudio Amorim

Exemplos de Código-Fonte em Linguagem C
Parte 1: Funções numéricas.
(V 1.3 – EM CONSTRUÇÃO)

Prezado(a) Estudante:

Aqui você encontra exemplos de funções numéricas, em código-fonte amplamente comentado, a fim de estudar estruturas e técnicas de programação.

Todas rotinas foram testadas. Contudo, apesar do esforço em mantê-las portáveis, não é possível garantir que funcionarão, na forma em que se encontram, em qualquer compilador C. Por isso, fique atento a eventuais mensagens de erro ou avisos (*warnings*) durante a compilação e linkedição.
O tipo `long long`, em especial, pode não ser nativo no seu ambiente. Se for o caso, substitua-o pelo tipo `long`. Também não é impossível que haja algum erro de transcrição, da IDE para o editor de texto, o que demandaria alguma pequena correção.

O aprendizado de C é importante para maior intimidade com o funcionamento dos computadores, proporcionando fundamentos para a programação avançada em quaisquer outras linguagens. Por outro lado, na prática, a programação de interfaces, no *front-end* é feita usando bibliotecas próprias, ou outras linguagens, sobre os navegadores WEB ou ambientes de janelas, voltados a eventos. Logo, não faz sentido incluirmos, nos nossos exemplos, longas salvaguardas para possíveis erros de entradas de dados.

Um programa de computador não é um segredo pessoal nem um código mirabolante de um grupo fechado, mas um patrimônio profissional. Por isso, precisa ser legível e acessível a intervenções posteriores, pelo autor ou autora inicial, mas também por outras pessoas que venham a fazer parte da equipe de desenvolvimento.

Código sinuoso e pouco legível acaba gerando erros e, consequentemente, aumento de tempo e custos de desenvolvimento. No limite, pode queimar a imagem de uma empresa, pois baixa a confiabilidade dos produtos e atrasa as correções. Sabe-se, há décadas, que quanto mais tarde um erro é descoberto, no ciclo de desenvolvimento, mais difícil e mais caro se torna corrigi-lo. Por isso, importa programar acertando de primeira, por meio de técnicas e padrões próprios, nunca – mas nunca, mesmo! – por tentativa e erro.

Um programa também deve ser eficiente. Não se justifica a elaboração de código ineficiente, a pretexto de que os computadores dispõem hoje de processadores poderosos e muita memória RAM. Ao contrário: o tamanho e a complexidade dos problemas que queremos resolver por meios computacionais coloca desafios novos todos os dias, em termos de desempenho do código.

Nenhum programa é perfeito: quase sempre há espaço para melhorias, que se evidenciam sob outros olhares, que não o do programador original. Então, fica o convite para que você fique atento à possibilidade de aprimorar as rotinas aqui apresentadas.

```
//  Nesse programa são definidas e testadas algumas funções
//  numéricas
//
//  Prof. Cláudio Amorim
//  UNEB - Campus I / DCET, março de 2019
//  Atualizado em julho de 2019



#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <math.h>
#include "NumerosPrimos.h"

short ehIgualAprox(double x, double y, double aprox);

void somaInteiros(char n1[], char n2[], char soma[]);
void somaInteiros2(char n1[], char n2[], char soma[]);

unsigned long long fatorialRec(unsigned long n);
unsigned long long fatorialIter(unsigned long n);

unsigned char ehPrimo(const long long p);
unsigned char ehPerfeito(unsigned const long long n);
void sequenciaPrimos(long long vPrimos[], long long maxPrimo);
void sequenciaPrimosE(long long vPrimos[], long long maxPrimo);

unsigned char ehPotenciaDe2Iter(const long long n);
unsigned char ehPotenciaDe2Bitw(const long long n);


void fatorar(unsigned long long n, long long fat[]);
void fatores(unsigned long long n, long long fat[]);

float maior(float x[], size_t comp);
float maiorAbs(float x[], size_t comp);
```

```c
void binarioParaString(unsigned int numBin, char strBin[]);
short strBinParaInt(char strBin[], u_int32_t *numDec);
void numIntParaStrHex(u_int32_t nInt, char strHex[]);

void bubbleSortInt(int m[], size_t comprimento);

unsigned short contaIguais(int n1[], int n2[], const size_t L1,
const size_t L2);



#define TESTE 10

int main(int argc, const char * argv[]) {

    unsigned long long n = 1;

    do {
        printf("Digite um número inteiro positivo: ");
        scanf("%llu", &n);

        if (n == 0) {
            printf("\nFim do programa\n");
            break;
        }

#if TESTE == 1 //FATORIAL

        printf("fatorial de %llu = %llu\n\n", n, fatorialIter(n));

#elif TESTE == 2 //PRIMO

        printf("%llu %s \n\n", n, ehPrimo(n)? "é primo" :
               "NÃO é primo");

#elif TESTE == 3 //FATORACAO

        long long fator[40] = {0}; //Fatores repetidos.
        long long fatorRep[20] = {0}; //Fatores nas posições pares
                  e quantidade de repetições nas posições ímpares.

        fatorar(n, fator);

        printf("Os fatores de %llu são: ", n);
        for(size_t i = 0; fator[i]>0; i++)
            printf("%llu ", fator[i]);
        printf("\n\n");

        fatores(n, fatorRep);

        printf("%llu = ", n);
```

```
        for(size_t i = 0; fatorRep[i]>0; i += 2)
            printf("%llu^%llu %s ", fatorRep[i], fatorRep[i+1],
            fatorRep[i+2] == 0? "" : "* ");
        printf("\n\n");

#elif TESTE == 4 //POTÊNCIA DE 2

        if (ehPotenciaDe2Iter(n)) printf("%llu é potência de 2\n
                                          n", n);
        else printf("%llu NÃO É potência de 2\n\n", n);

#elif TESTE == 5 //Binário

        char strEmBinario[33] = "";
        binarioParaString(n, strEmBinario);
        printf("O equivalente binário é  %s\n\n", strEmBinario);

#elif TESTE == 6 //Bubblesort

        const size_t COMPRIMENTO = 1000;
        int m[COMPRIMENTO] = {0};
        for (size_t i = 0; i < COMPRIMENTO; i++)
            m[i] = rand() % 10000;

        bubbleSortInt(m, COMPRIMENTO);

        for (size_t i = 0; i < COMPRIMENTO; i++)
            printf("%4d, %c", m[i], (i % 15 ? ' ' : '\n'));

#elif TESTE == 7

        char sBinario[33], sHex[9];
        u_int32_t n = 0;

        printf("\nDigite o binário sem sinal de até 32 bits:\n");
        scanf("%s",sBinario);
        if (strBinParaInt(sBinario,&n)) {
            printf("O equivalente decimal é %d\n", n);
            numIntParaStrHex(n, sHex);
            printf("O equivalente hexadecimal é %s\n\n", sHex);
        }
        else printf("A entrada não é um número binário.\n");

#elif TESTE == 8

        long long vPrimos[1000000] = {0};
        sequenciaPrimosE(vPrimos, n);
        size_t i = 0;

        while(vPrimos[i]) printf((i % 5)? "%8llu " : "%8llu\n ",
              vPrimos[i++]);
        printf("\n\nNumero de primos no intervalo: %lu\n\n", i);
```

```
#elif TESTE == 9

        char na[50] = "", nb[50] = "", s[51] = "";
        printf("Digite o primeiro operando: ");
        scanf("%s", na);
        printf("Digite o segundo operando: ");
        scanf("%s", nb);
        somaInteiros2(na, nb, s);
        printf("\n");
        printf("Soma = %s\n\n", s);

#elif TESTE == 10

        unsigned long long n;
        printf("Digite o número, para saber se é perfeito: ");
        scanf("%llu", &n);
        if (ehPerfeito(n))
           printf("%llu é um número perfeito.\n n", n);
        else printf("%llu NÃO é um número perfeito.\n\n", n);

#endif

    } while (1);

    return 0;
}


unsigned long long fatorialRec(unsigned long n) {
//Calcula o fatorial de n recursivamente.

    if ((n == 0) || (n == 1)) return 1;
    else return (n * fatorialRec(n-1));
}


unsigned long long fatorialIter(unsigned long n) {
//Calcula o fatorial de n iterativamente.

    unsigned long long fatorial = 2;

    if ((n == 0) || (n == 1)) return 1;
    if (n == 2) return 2;
    else do fatorial *= n--; while(n > 2);

    return fatorial;
}
```

```
void fatorar(unsigned long long n, long long fat[]) {
    // Coloca os fatores do número n no vetor fat, com repetições.

    long long fator = 2;
    size_t i = 0;

    while (n % 2 == 0) fat[i++] = 2;
    fator = 3;

    while(n > 1) {
        if(n % fator == 0) {
            fat[i++] = fator;
            n /= fator;
        }
        else fator += 2;
    }
}


void fatores(unsigned long long n, long long fat[]) {
// Coloca, em posições sucessivas do vetor fat, cada fator do
// número n, seguido do número de vezes em que o fator ocorre.

    long long fator = 2, repet = 0;
    size_t i = 0;

    while(n > 1) {
        while(n % fator != 0) fator++;

        while(n % fator == 0) {
           repet++;
           n /= fator;
        }

        //Coloca o fator em uma posição par do array.
        fat[i++] = fator;
        //Coloca a quantidade de repetições na posição ímpar
        //subsequente.
        fat[i++] = repet;

        repet = 0;
    }
}
```

```c
unsigned char ehPotenciaDe2Iter(const long long n) {
    //Verifica se n é uma potência de dois por processo
    //iterativo, em tempo proporcional a log(n).

    unsigned long long m = 1;

    while(m < n) m <<= 1;
    if (m == n) return 1;
    else return 0;

}


unsigned char ehPotenciaDe2Bitw(const long long n) {
    //Verifica n é uma potência de dois por meio de uma subtração
    //e uma operação bitwise, em tempo constante.

    if (n == 0) return 0;
    else if (n & (n - 1)) return 0;
            //n tem mais de um bit = 1, logo não é potência de 2.
    else return 1; //n só tem um bit = 1, logo é potência de 2.
}


float maior(float x[], size_t comp) {
    //Retorna o maior número presente no array x, sendo
    //comp o comprimento do array.

    float maior = x[0];

    for(size_t i = 1; i < comp; i++)
        if (x[i] > maior) maior = x[i];

    return maior;
}


float maiorAbs(float x[], size_t comp) {
    //Retorna o maior módulo presente no array x, sendo
    //comp o comprimento do array.

    float maior = x[0], menor  = x[0];

    for(size_t i = 1; i < comp; i++) {
        if (x[i] > maior) maior = x[i];
        if (x[i] < menor) menor = x[i];
    }

    if (maior > -menor) return maior;
    else return -menor;

    return maior;
}
```

```c
void binarioParaString(unsigned int numBin, char strBin[]) {

    unsigned int mascara = 0x80000000;

    for (unsigned short i = 0; i < 32; i++) {
        strBin[i] = (numBin & mascara) ? '1' : '0';
        mascara >>= 1;
    }
}


void bubbleSortInt(int m[], size_t comprimento) {

    size_t i = 0,
           j = comprimento - 1;
           //Última posição a comparar com a posição seguinte

    int temp = 0;

    unsigned char troca = 1;

    while (troca) {

        troca = 0;

        for (i = 0; i < j; i++) {
            if (m[i] > m[i + 1]) {
               temp = m[i]; m[i] = m[i + 1]; m[i + 1] = temp;
               troca = 1;
            }
        }
        --j;
    }
}


void numIntParaStrHex(u_int32_t nInt, char strHex[]) {

    char tabHex[] = "0123456789ABCDEF";

    size_t i = 7;

    for (; nInt > 0; nInt /= 16) {
        strHex[i--] = tabHex[nInt % 16];
        printf("%s", strHex);
    }
    for (size_t j = 0; j <= i; j++) strHex[j] = '0';
}
```

```
short strBinParaInt(char strBin[], u_int32_t *numDec) {
//A função recebe uma string que representa um número binário e
//devolve o valor correspondente em um inteiro de 32 bits sem
//sinal.
//O valor de retorno é 1, em caso de conversão bem-sucedida, e
zero, caso contrário.

    unsigned short potencia = 1;
    *numDec = 0;

    for(short i = (strlen(strBin) - 1); i >= 0; i--) {
        if (strBin[i] == '1') *numDec += potencia;

        else if(strBin[i] != '0') return 0;
             //Se o caractere não representa 0 ou 1, retorna erro.
        potencia *= 2;
    }
    return 1;
}


void sequenciaPrimos(long long vPrimos[], long long maxPrimo) {

    size_t i = 0;
    long long n;
    vPrimos[0] = 2;

    for(n = 3; n <= maxPrimo; n+=2) {
        i = 0;
        while(vPrimos[i] && (n % vPrimos[i]) != 0) i++;
        if (vPrimos[i] == 0) vPrimos[i] = n;
    }
}


void sequenciaPrimosE(long long vPrimos[], long long maxPrimo) {

    size_t i = 0, iProximo = 2;
    long long n;
    vPrimos[0] = 2; vPrimos[1] = 3;

    for(n = 5; n <= maxPrimo; n+=2) {
        i = 0;
        while((vPrimos[i] < sqrt(n)) && (n % vPrimos[i]) != 0)
             i++;
        if ((n % vPrimos[i]) != 0) vPrimos[iProximo++] = n;
    }
}
```

```c
void somaInteiros2(char n1[], char n2[], char soma[]) {

    const size_t MAXLEN = 50;
    char algarismo = 0, vaiUM = 0;
    size_t lenN1 = strlen(n1), lenN2 = strlen(n2);
    char op1[MAXLEN] = {0}, op2[MAXLEN] = {0};

    memset(soma, 48, MAXLEN);

    // Normalizar os operandos, preenchidos com 0 à esquerda.
    size_t j = 0;
    for(size_t i = (MAXLEN - lenN1); i < MAXLEN; i++) {
        op1[i] = n1[j] - 48;
        j++;
    }
    j = 0;
    for(size_t i = (MAXLEN - lenN2); i < MAXLEN; i++) {
        op2[i] = n2[j] - 48;
        j++;
    }


    // Somar.
    vaiUM = 0; size_t iSoma = MAXLEN - 1;
    do {
        algarismo = (vaiUM + op1[iSoma] + op2[iSoma]) % 10;
        //printf("Algarismo: %c; ", algarismo + 48);
        vaiUM = (vaiUM + op1[iSoma] + op2[iSoma]) / 10;
        //printf("VaiUm: %c; ", vaiUM + 48);
        soma[iSoma] = algarismo + 48;
        iSoma--;
    } while (algarismo || vaiUM);

    //algarismo = soma[0] - 48;

    // Formatar a string soma.
    j = 0;
    while(!(soma[j++] - 48)) {
    }

    int i = 0; j -= 1;
    while(j < MAXLEN) {
        soma[i++] = soma[j++];
    }
    soma[i] = '\0';
}
```

```c
unsigned char ehPerfeito(unsigned const long long n) {

    // tentar dividir até a raiz de n
    // a cada divisor encontrado, encontrar tambem seu par
    // incrementar soma dos divisores
    // se maior do que o número, sair retornando 0
    // se sair do laço sem excesso na soma, verificar soma
    // se for diferente de n, retornar 0; se for igual, retornar 1

    unsigned long long divisor = 2, divisor2 = 0, soma = 1,
                       raizInt = sqrt(n), divMax = raizInt;

    if(sqrt(n) == raizInt) {
        // Se a raiz é um valor inteiro, é divisor. Soma e elimina
        // do teste.
        soma += raizInt;
        divMax--;
    }
    // Print para debbuging;
    printf("RaizInt: %llu  divMax: %llu\nSoma: %llu\n",
           raizInt, divMax, soma);


    while((divisor <= divMax) && (soma < n)) {
        if((n % divisor) == 0) {
            divisor2 = n / divisor;
            soma += (divisor + divisor2);
            // Print para debbuging;
            printf("Soma: %llu  Divisores: %llu e %llu\n",
                    soma, divisor, divisor2);
        }
        divisor++;
    }
    if(soma == n) return 1;
    else return 0;

}
```

```
short ehIgualAprox(double x, double y, double aprox) {

    // Retorna verdadeiro se a diferença entre x e y é menor do
    // que o produto entre o menor módulo e a aproximação.
    // Assim, o critério de proximidade não é absoluto,
    // mas relativo.

    double menor = x, maior = y;
    if(x > y) { maior = x; menor = y; }

    double menorModulo = fabs(x), maiorModulo = fabs(y), temp;
    if (menorModulo > maiorModulo) {
        temp = maiorModulo; maiorModulo = menorModulo;
        menorModulo = temp;
    }

    if((maior - menor) < (aprox * menorModulo)) return (1);
    else return (0);
}


har mesmoConteudo(int n1[], int n2[], const size_t L1, const
size_t L2) {

    // Essa rotina usa um método simples (embora não seja
    // o mais eficiente)para saber se os vetores têm os mesmos
    // elementos: ordena cada um deles e compara-os, em seguida,
    // posição a posição.

    if(L1 != L2) return(0);
       // Vetores de tamanhos distintos não têm o mesmo conteúdo.

    else {
        bubbleSortInt(n1, L1);
        bubbleSortInt(n2, L2);

        for(size_t i = 0; i < L1; i++)
            if(n1[i] != n2[i]) return(0);
        }

    return(1);
}
```

```c
unsigned short contaIguais(int n1[], int n2[], const size_t L1,
const size_t L2) {

    bubbleSortInt(n1, L1);
    bubbleSortInt(n2, L2);

    // Busca tendo como base sempre o menor vetor.
    // Isso ilustra a flexibilidade proporcionada pelo uso de
    // apontadores, em C. Pode-se escolher o vetor de trabalho
    // praticamente sem overhead.

    size_t compBase = L1, compBusca = L2;
    int *pBase = n1, *pBusca = n2;

    if(L1 > L2) {
        compBase = L2; compBusca = L1;
        pBase = n2; pBusca = n1;
        // O menor vetor tem prioridade na busca.
    }

    size_t iBase = 0, iBusca = 0;
    short cont = 0;

    while((iBase < compBase) && (iBusca < compBusca)) {
        while((*pBusca < *pBase) && (iBusca < compBusca)) {
            iBusca++; pBusca++;
        }

        if(*pBusca == *pBase) {
            // Print para debbuging;
            printf("Elemento encontrado: %u\n", *pBusca);
            printf("iBase: %zu, iBusca: %zu\n", iBase, iBusca);
            cont++;
            iBusca++; pBusca++;
            // Prossegue a busca a partir do próximo,
            // para não contar repetições.
        }

        iBase++; pBase++;
    }

    return(cont);
}
```