EA871 – Laboratório de Programação Básica de Sistemas Digitais

Atividade 02 – 2º semestre de 2019

1. Objetivos

- Familiarização com a linguagem C e algoritmos essenciais
- Utilizar ponteiros, vetores e funções

2. Resumo da Atividade

Nesta atividade, aprenderemos a desenvolver e analisar programas em C usando ponteiros, vetores e funções.

3. Roteiro da Aula

Exercício 1

Leia o código-fonte abaixo, correspondente a um trecho do programa 07-funcoes.c. Ele foi escrito em linguagem C.

a) Encontre todas as palavras-chave que você não conhece.
b) Faça uma hipótese sobre o que o programa faz.
c) Compile e execute o programa. Ele faz o que você imaginava?
d) O que significa void()?
e) Como a função procedimentoN() "sabe" que deve imprimir o valor 3 na tela?
f) Modifique o programa para que a função procedimentoN() imprima o valor 95, ao invés de 3.

```
#include <stdio.h>

void procedimento() {
	printf("Ola, mundo! Eu sou um procedimento!\n");
}

void procedimentoN(int N) {
	printf("Ola, mundo! Eu sou o procedimento %d!\n", N);
}

int main() {
	int i, j;
	printf("Procedimentos:\n");
	procedimento();
	procedimentoN(2);
	i = 3;
	procedimentoN(i);
	return 0;
}
```

Exercício 2

Leia o código-fonte abaixo, correspondente a outro trecho do programa 07-funcoes.c. Ele foi escrito em linguagem C.

a) Encontre todas as palavras-chave que você não conhece.
b) Faça uma hipótese sobre o que o programa faz.
c) Compile e execute o programa. Ele faz o que você imaginava?
d) O rótulo *i* definido dentro da função procedimentoEscopo() se refere à mesma variável *i* definida na função main()? Como é possível demonstrar isso?

```
#include <stdio.h>

void procedimentoN(int N) {
```

```
    printf("Ola, mundo! Eu sou o procedimento %d!\n", N);
}

void procedimentoEscopo(int N) {
    int i;
    i = N + 1;
    printf("Sucessor de %d: %d\n", N, i);
}

int main() {
    int i;
    i = 3;
    procedimentoN(i);
    printf("----\n");
    printf("Escopo:\n");
    printf("i=%d (main)\n", i);
    procedimentoEscopo(i);
    printf("i=%d (main)\n", i);
}
```

Exercício 3

Ao declarar uma variável, estamos pedindo ao compilador que reserve um espaço de memória correspondente ao tipo (int, char, float) e então associe esse espaço ao rótulo que escolhemos. É possível descobrir o endereço de memória que foi alocado para nossa variável usando o operador **&**:

```
#include <stdio.h>

int main() {
    int i;
    printf("A variavel i tem valor %d e esta na posicao 0x%x\n", i, &i);
    return 0;
}
```

Também, podemos criar uma variável do tipo "ponteiro", que contém um endereço de memória. Para isso, usamos o modificador *, na forma:

```
#include <stdio.h>

int main() {
    int i;
    int *j; /* j eh um ponteiro para um int */
    i = 10;
    j = &i; /* j recebe o endereco de i */
    *j = 50; /* o endereco apontado por j recebe 50 */
    printf("i=%d\n", i);
    return 0;
}
```

O programa abaixo contém uma função que recebe ponteiros como parâmetros. Ele foi extraído de 07-funcoes.c.

a) Encontre todas as palavras-chave que você não conhece.

b) Faça uma hipótese sobre o que o programa faz.

c) Compile e execute o programa. Ele faz o que você imaginava?

```
#include <stdio.h>

void procedimento_ref(int *N, int M) {
    *N = *N + 1;
    M = M + 1;
}
```

```
int main() {
	int i, j;
	i = 0;
	j = 0;
	procedimento_ref(&i, j);
	printf("i=%d, j=%d\n", i, j);
	return 0;
}
```

Exercício 4

Colchetes [] são usados em C para definir e acessar vetores. Ao declarar um vetor, como em:

```
int i [10];
```

estamos pedindo ao compilador que reserve espaço suficiente para um certo número (neste caso, 10) de variáveis do tipo definido (neste caso, int).

Podemos então acessar cada posição usando:

```
printf(“%d\n”, i[0]);
printf(“%d\n”, i[3]);
printf(“%d\n”, i[4]);
```

Lembre-se que a posição inicial tem índice 0 e que, portanto, a posição final do vetor tem índice 9 (neste caso, já que definimos um vetor de 10 posições).

O programa abaixo corresponde ao código-fonte 09-vetores.c.
a) Encontre todas as palavras-chave que você não conhece.
b) Faça uma hipótese sobre o que o programa faz.
c) Compile e execute o programa. Ele faz o que você imaginava?

```
#include <stdio.h>

#define TAMANHO_VETOR 5

void imprimir_vetor(int vetor[], int N) {
	int i;
	for (i=0; i<N; i++) {
		printf("%d\t", vetor[i]);
	}
	printf("\n");
}


void ler_vetor(int vetor[], int N) {
	int i;
	int j;
	for (i=0; i<N; i++) {
		scanf("%d", &j);
		vetor[i] = j;
	}
}


int produto_interno(int vetor1[], int vetor2[], int N) {
	int p;
	int i;
	p = 0;
```

```
    for (i=0; i<N; i++)
        p = p + (vetor1[i] * vetor2[i]);

    return p;
}

int main() {
    int vetor_base[] = {1, 2, 6, 3, 5};
    int meu_vetor[TAMANHO_VETOR];
    ler_vetor(meu_vetor, TAMANHO_VETOR);
    printf("Vetor lido! Imprimindo...\n");
    imprimir_vetor(meu_vetor, TAMANHO_VETOR);
    printf("Produto interno com vetor-base: %d\n", produto_interno(vetor_base, meu_vetor, TAMANHO_VETOR));
    return 0;
}
```

Exercício 5

No programa abaixo,
a) Faça uma hipótese sobre o que o programa faz.
b) Compile e execute o programa. Ele fez o que você imaginava?

```
#include <stdio.h>


int main() {
    char string[] = {'a', 'b', 'c', 'd', 'e'};
    char *p1 = &(string[0]);
    char *p2 = string;
    for (int i=0; i<5; i++) {
        printf("*p1=%c\t", *p1);
        p1++;
        printf("*p2=%c\n", *p2);
        p2++;
    }
    return 0;
}
```

4. Exercício computacional para casa (individual)

Buffer circular

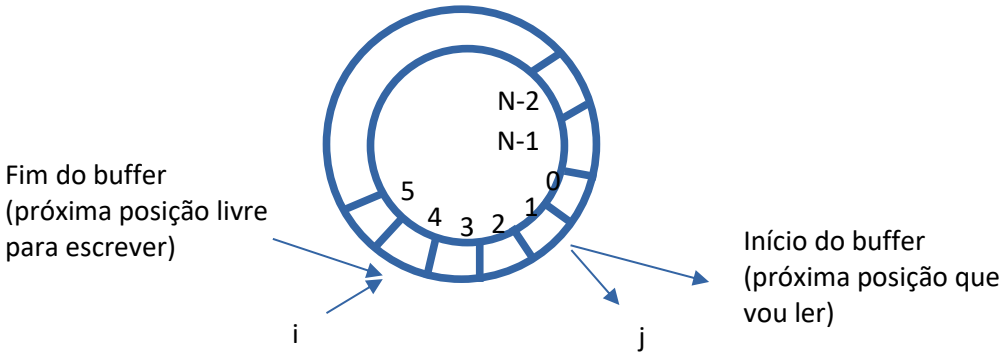
Vetores podem ser usados de muitas maneiras em programas de computador. Uma dessas maneiras se chama **buffer circular**. O buffer circular é uma estrutura de dados que é usada para manejar filas de dados que são recebidos de algum lugar no seu programa. Em sistemas microcontrolados, buffers circulares são usados para armazenar dados recebidos até que eles possam ser processados.

O buffer circular parte da ideia de que vamos chamar uma função "adicionar_buffer()" que adiciona um elemento no nosso vetor. Para isso, precisamos de um índice i, que aponta para a próxima posição livre do vetor, e que é incrementado a cada chamada de "adicionar_buffer()".

Outra possibilidade de ação sobre o buffer circular é "remover um elemento". Nesse caso, usamos outra função, "remover_buffer()", que retorna o primeiro elemento da fila – isto é, o elemento mais antigo que foi adicionado. Isso significa que precisamos também de um índice j, que aponta para o elemento mais antigo do buffer que ainda não foi processado. Este índice deve ser atualizado sempre que a função remover_buffer() for chamada.

Quando um elemento é removido do buffer, o espaço em que ele estava fica livre, isto é, pode ser usado para adicionar um novo elemento. Então, a função adicionar_buffer() deve ser capaz de manejar o índice i de forma a voltar ao começo do vetor e verificar se aquela posição está livre.

Uma possível maneira de pensar o buffer circular é imaginar que o vetor de dados está posicionado sobre um círculo. Para verificar se você entendeu bem, identifique na figura abaixo as posições do buffer circular que contém dados válidos, isto é, dados que foram recebidos, mas ainda não processados:



Descrição da Atividade

Esta tarefa consiste em programar as funções relacionadas a um buffer circular. No caso, vamos construir e manipular um buffer circular com tamanho igual a cinco (isto é, cabem 5 elementos nele).

O programa receberá uma cadeia de caracteres, continuamente. Se o caractere recebido for um dígito (0, 1, 2, 3, 4, 5, 6, 7, 8, ou 9), ele deverá ser adicionado ao buffer. Se o caractere for uma letra (a-z ou A-Z), o elemento mais antigo do buffer deverá ser removido. Se for um fim de linha (\n - tecla ENTER), o programa deve encerrar.

Depois de qualquer adição ou remoção de elemento do buffer, seu conteúdo deverá ser impresso inteiramente, do elemento mais antigo para o mais recente, com espaço entre os elementos e um fim de linha (\n) ao final.

Tentativas de inserção em buffer cheio ou remoções em buffer vazio devem ser ignoradas, mas a rotina de impressão deve ser chamada normalmente. Isso significa que uma tentativa de remover elementos de um buffer vazio deve levar a imprimir somente \n, e a tentativa de inserir em um buffer cheio leva a imprimir o conteúdo do buffer inteiro, que não deve ter sido alterado (seguido de \n).

Exemplo:

Sequência de entrada:
a123456bZf789\n

Saída:
\n
1\n
1 2\n
1 2 3\n
1 2 3 4\n
1 2 3 4 5\n
1 2 3 4 5\n
2 3 4 5\n
3 4 5\n
4 5\n
4 5 7\n
4 5 7 8\n

4 5 7 8 9\n

Importante: esse código será usado em atividades posteriores durante o semestre. Por isso, lembre-se de respeitar os cabeçalhos que foram pré-estabelecidos!

Instruções para a submissão do trabalho

1) Baixe o *template* da atividade 2 do Google Classroom.
2) Modifique o arquivo src/main.c para completar seu laboratório.
3) Verifique se o seu programa responde corretamente aos testes mostrados na tabela a seguir.
4) Quando terminar, crie um arquivo no formato .zip (Aviso: não use .tar.gz nem .rar) cujo nome é seu_ra.zip (Exemplo: 025304.zip) com toda a estrutura de diretórios que você baixou.
5) Faça o *upload* da sua solução da atividade 2 no Google Classroom.

OBS: No Linux, o comando **make** pode ser usado em uma janela do terminal aberta no diretório raiz para compilar seu código. O comando **make test** testa o funcionamento de seu programa.

Tabela de testes	
Entrada	Saída (sequência impressa)
1234567890	1\n 1 2\n 1 2 3\n 1 2 3 4\n 1 2 3 4 5\n 1 2 3 4 5\n 1 2 3 4 5\n 1 2 3 4 5\n 1 2 3 4 5\n 1 2 3 4 5\n
a1b2c3	\n 1\n \n 2\n \n 3\n
12345abcde	1\n 1 2\n 1 2 3\n 1 2 3 4\n 1 2 3 4 5\n 2 3 4 5\n 3 4 5\n 4 5\n 5\n \n
123abcdef456	1\n 1 2\n 1 2 3\n 2 3\n 3\n \n \n \n \n 4\n 4 5\n 4 5 6\n