INF 112 – Prática 2, Alocação Dinâmica – 2019/2

Exercício 0 Registre-se no Piazza como aluno! O link a ser seguido para registro está no PVAnet.

Exercício 1 Implemente um "simulador de par ou ímpar". Seu programa deverá funcionar da seguinte forma: ao executá-lo, ele deverá perguntar ao usuário quantas jogadas (seja *n* esse número) haverá na simulação. A seguir, ele deverá alocar dois arranjos *a* e *b* de tamanho *n* e, então, preencher tais arranjos com números aleatórios entre 0 e 9. Finalmente, ele deve varrer os dois arranjos e contar a quantidade de jogadas com valor par e ímpar (a jogada *i* é considerada par se *a[i]+b[i]* for par e ela é ímpar caso contrário). Seu programa deverá imprimir os dois arranjos gerados e também a quantidade de jogadas par e ímpar obtidas na simulação. Após o término da simulação, ele deverá voltar ao "menu" de simulação, pedindo para o usuário digitar novamente o valor de *n* (o programa deverá parar quando o *n* digitado for negativo).

Veja um exemplo de tela abaixo:

```
Digite a quantidade de jogadas a simular: 5
6, 2, 5, 1, 1,
7, 8, 1, 2, 0,
Par: 2
Impar: 3
Digite a quantidade de jogadas a simular: 3
1, 8, 0,
7, 8, 3,
Par: 2
Impar: 1
Digite a quantidade de jogadas a simular: -1
```

Observações:

- Use a função `rand` para gerar números "aleatórios" (http://www.cplusplus.com/reference/cstdlib/rand/).

- Seu programa deverá ter (pelo menos) as seguintes funções implementadas:
    - Uma função `preencheAleatorios` que recebe como parâmetro um arranjo de inteiros e um número *n* e, então, preenche o arranjo com *n* números aleatórios entre 0 e 9.
    - Uma função `imprime` que recebe como parâmetro um arranjo de inteiros e um número *n* e, então, imprime esse arranjo na tela.
    - Uma função `contaParImpar` que recebe como parâmetro dois arranjos *a*, *b*, o tamanho *n* desses arranjos, um inteiro *par* e um inteiro *ímpar* e, então, conta a quantidade de números par/ímpar que há nas *n* jogadas.

- Faça o seguinte experimento:
    - Remova a desalocação de memória do seu programa (espero que você tenha se lembrado de colocá-la :)).
    - Execute o programa (mas não simule nada por enquanto).
    - Abra um novo terminal, digite `ps -aux | grep nomeDoExecutavel`, onde `nomeDoExecutável` é o nome do executável do seu programa. Com esse comando, descubra o número do processo do seu programa.
    - Digite (no novo terminal) o comando `top -p numeroProcesso`, onde `númeroProcesso` é o número do processo de seu programa (descoberto no passo anterior).
    - Anote o uso de memória (coluna `res` do top).
    - Simule a execução do seu programa com os seguintes valores de *n*: 1, 1000000, 4000000 e 5 e veja como o uso de memória evolui.
    - Repita os procedimentos anteriores, mas deixando seu código com a desalocação de memória.

Exercício 2 Um formato de imagem bastante simples de trabalhar é o formato “pbm”. O formato pbm é utilizado para armazenar imagens em formato “preto e branco” e pode ser codificado utilizando arquivos de texto. A seguir, temos um exemplo de imagem “pbm”:

```
P1
4 8
1 1 1 1
1 0 0 1
1 0 0 1
1 0 0 1
1 0 0 1
1 0 0 1
1 0 0 1
1 1 1 1
```

Cole o texto do arquivo de exemplo no `gedit` (ou qualquer outro editor), salve como um arquivo “imagem.pbm” e visualize-a utilizando um visualizador de imagens.

Nesse exemplo, o “P1” é um código “mágico” (utilizado internamente pelo formato pbm você pode supor que todo arquivo pbm em formato texto terá esse código). A seguir, temos os números 4 e 8 que indicam, respectivamente, o número de colunas e linhas na imagem. Finalmente, temos um conjunto de 0's e 1's representando os pixels da imagem (o pixel 0 é preto e o 1 é branco). O formato “pbm” também permite a adição de comentários utilizando o caractere “#” mas, por simplicidade, vamos trabalhar com imagens que não contêm comentários.

Faça um programa que lê uma imagem no formato pbm (como no exemplo) a partir do “cin”, armazena a imagem em um registro, inverte as cores da imagem e, finalmente, “imprime” a imagem no “cout”.

Exemplo de funcionamento:

```
$ ./exercicio2
P1
4 8
1 1 1 1
1 0 0 1
1 0 0 1
1 0 0 1
1 0 0 1
1 0 0 1
1 0 0 1
1 1 1 1
P1
4 8
0 0 0 0
0 1 1 0
0 1 1 0
0 1 1 0
0 1 1 0
0 1 1 0
0 1 1 0
0 0 0 0
```

No linux, a entrada/saída dos programas pode ser redirecionada de/para arquivos. Por exemplo, se digitarmos o comando abaixo:

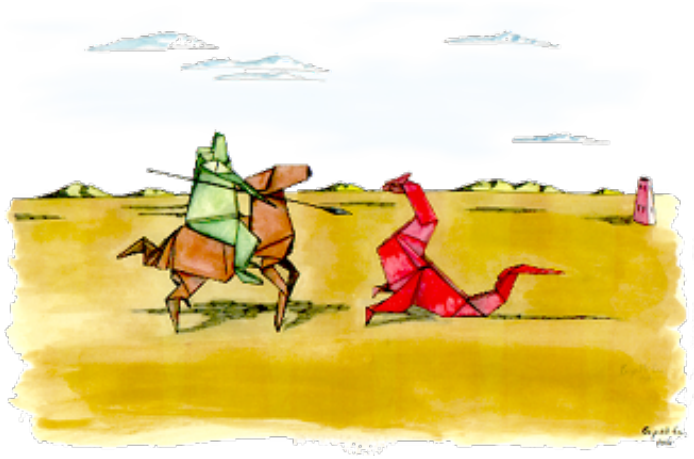
```
./exercicio2 < imagem.pbm > imagem2.pbm
```

O programa “exercício2” irá ler os dados de entrada (pelo cin) a partir do arquivo imagem.pbm (os dados serão lidos na mesma ordem em que aparecem no arquivo; é como se o usuário digitasse o texto que está no arquivo) e toda a saída (do cout) será gravada no arquivo imagem2.pbm. Use essa técnica para testar seu programa em algumas imagens pbm.

Observação: implemente seu exercício completando o código abaixo:

```cpp
#include <iostream>
using namespace std;

struct Imagem {
    int **pixels; //matriz com os pixels da imagem
    int nrows; //numero de linhas na imagem (altura)
    int ncolumns; //numero de colunas na imagem (largura)
};

//Insira seu codigo aqui....


//

int main() {
    Imagem im;
    leImagem(im);
    inverteCorImagem(im);
    imprimeImagem(im);
    deletaImagem(im);
    return 0;
}
```

Exercício 3 O objetivo desse exercício é introduzir o uso da ferramenta de depuração *Valgrind* (http://valgrind.org).



Comece com a leitura desse tutorial https://www.ic.unicamp.br/~rafael/cursos/2s2017/mc202/valgrind.html
Esse tutorial sobre alocação pode lhe ser útil, se você não conhecer as funções `malloc` e `free`
https://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html (**para casa:** faça todos os exercícios dessa página!)

Parte 1: Repare que todos os códigos do exemplo estão em C. Execute cada um desses exemplos utilizando o compilador *gcc*. Utilize o Valgrind para verificar o que há (ou não) de errado em cada caso.

Parte 2: Porte cada um dos exemplos para C++. Novamente, utilize Valgrind em cada um deles.

Parte 3: No início da aula 2, vimos 4 exemplos de código em C++ com problemas de acesso a endereços inválidos de memória. Use a ferramenta `valgrind` para identificar o problema consertar `03.cpp` (disponível na Aula 02, no PVAnet) utilizando alocação dinâmica na função `le_nome`. Não se esqueça de desalocar a memória...

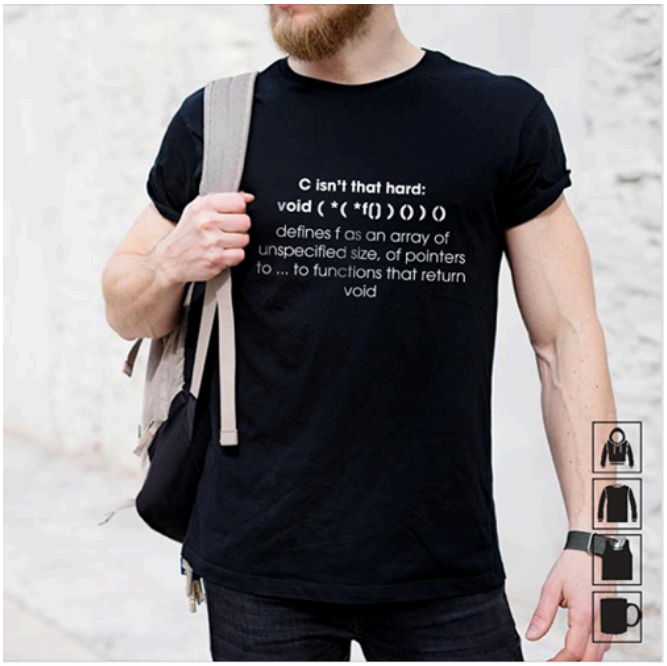
Observação: compile os programas utilizando a *flag -g*. Isso fará com que a saída do Valgrind seja legível por um humano. Exemplo:

```
gcc -o test -g test.c ou g++ -o test -g test.cpp
```

Observação 2: será de enorme ajuda para vocês, tanto em Programação II quanto em Estrutura de Dados, ter um bom relacionamento com a ferramenta Valgrind.

Exercício 4 (para casa) Leia os Capítulos 6 e 10 da apostila de C, disponível no PVAnet. Faça todos os exercícios destes dois capítulos.

Exercício 5 (para casa) Essa questão é significativamente mais complicada do que será cobrado nas provas e trabalhos. Trata-se de um desafio. Recentemente, o Facebook recomendou ao professor de INF 112 a seguinte camiseta:



Sim! Não obstante com ponteiros para variáveis, C e C++ têm ponteiros para funções. Nós veremos ponteiros para funções em breve, mas essa camisa vai muito além. Segundo ela,
`void (*(*f[])())()` define um arranjo, de tamanho não especificado, de ponteiros para funções que retornam `void`. Faça uma busca sobre ponteiros para funções e vetores de ponteiros para funções na Web. Você concorda com o que está escrito na camisa? Apresente um código (pequeno e simples) que corrobore sua resposta.

Observação: aos interessados, vou discutir essa questão no Piazza em breve.