

Alocação Dinâmica e Valgrind

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).

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

Figura 1: Exemplo de execução do programa.

Observações importantes:

- Use a função **rand** para gerar números “aleatórios” (<http://www.cplusplus.com/reference/cstdlib/rand/>);
- Para este exercício é obrigatório que a memória para armazenamento dos arranjos seja alocada dinamicamente;
- 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.
- Agora, execute 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 **nomeDoExecutavel** 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 **numeroProcesso** 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.
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 0 0 1
1 1 1 1
```

Figura 2: Exemplo de execução do programa.

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 (dinamicamente) 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
```

Figura 3: Exemplo de funcionamento do programa.

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 importante!: implemente seu exercício completando o código abaixo:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Imagem {
5     int **pixels; //matriz com os pixels da imagem
6     int nrows; //numero de linhas na imagem (altura)
7     int ncolums; //numero de colunas na imagem (largura)
8 };
9
10 //Insira seu codigo aqui....
11
12 //
13
14 int main() {
15     Imagem im;
16     leImagem(im);
17     inverteCorImagem(im);
18     imprimeImagem(im);
19     deletaImagem(im);
20     return 0;
21 }
```

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

Considerações!

- Passo inicial: Leitura do tutorial disponível em <https://www.ic.unicamp.br/~rafael/cursos/2s2017/mc202/valgrind.html>. Esse segundo 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>. (Sugestão: para fins de aprendizado e assimilação do conteúdo, fazer todos os exercícios dessa página! Mas não é necessário entregar!);
- 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;

Dicas!

- 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
```

- Para depuração dos códigos utilize o GDB.
- Neste exercício não é necessário entregar nenhum código!