Lab 3 – Alocação de Memória Estática, Dinâmica e Variáveis locais

Neste trabalho de laboratório pretende-se ver as diferentes formas de utilizar a memória, salientando as diferenças entre alocação estática, dinâmica e variáveis locais, e também abordar problemas que podem surgir.

1. Memória estática, dinâmica e a pilha

Abra o ficheiro **memoria.c** e descreva (editando os comentários) o efeito em termos de memória de cada uma das declarações e instruções assinaladas, respondendo às seguintes questões:

1. Quais as zonas de memória implicadas: armazenamento estático, dinâmico (*heap*) e/ou pilha?
2. Alguma memória é reservada ou libertada? Se sim, a que zona de endereçamento corresponde essa memória?
3. A instrução em análise pode levar a uma fuga de memória?

Pode recorrer ao **gdb** ou colocar **printf()**'s no seu programa para o auxiliar na sua investigação sobre o comportamento da memória.

2. Tabela de tamanho variável

O programa **vector.c**, disponibiliza uma estrutura para a implementação de uma tabela de tamanho variável (em Java e C++, estas tabelas de tamanho variável correspondem à classe *Vector*). O programa recorre a estruturas de dados *struct* e à gestão de memória em C para implementar as funcionalidades da tabela dinâmica.

Os comentários no código indicam o funcionamento pretendido das diversas funções a implementar. Analisando o código já implementado poderá ver como devem ser utilizadas as estruturas de dados definidas. Não esquecer que o `malloc()` não coloca a zero a zona de memória que aloca, e que neste caso pretende-se que os valores iniciais dos vetores sejam todos 0.

2.1) Analise o ficheiro Makefile para compreender as novidades em relação ao Makefile do trabalho anterior.

2.2) Escreva o código que falta nas zonas assinaladas em **vector.c**, nomeadamente:

a) Implemente a função `vector_print()` para imprimir para o ecrã todos os valores da lista ligada separados por `' -> '`.

b) Implemente a função **`vector_set()`** para adicionar um elemento a uma lista ligada numa posição determinado por um argumento de entrada.

c) Implemente a função **`vector_delete()`** para libertar toda a memória alocada dinamicamente pelo vetor. Não se esqueça de libertar toda a memória alocada e não apenas a correspondente à estrutura do vetor, mas também a memória alocada na lista ligada.

2.3) Teste o código implementado

Certifique-se que tem os ficheiros **vector.c**, **vector.h**, **test.c** e **Makefile** na diretoria corrente, e teste fazendo **make vector_test**.

Se o teste falhar, tente descobrir os *bugs* utilizando o **gdb** no **vector.exe** criado pelo make, ou com **`printf()`**'s no programa tendo que recompilar fazendo novamente **make vector_test**. Se tiver tempo, acrescente mais testes em **test.c**.

3. "Debugging" para craques em alocação de memória (Pontos Extra)

Vai encontrar em anexo um ficheiro **transforma.c** que utiliza as rotinas de vetores que programou no ponto 2. Comece por compilar e correr o programa. Vai constatar que o resultado não é bem o esperado. Explique porque é que a função **ZeroVector()** não coloca a zeros as entradas do vetor de entrada.

Para compilar pode usar **make transforma**.