Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Lectivo 2014/2015
Linguagens e Ambientes Programação – Teste 2

02 de Junho de 2015 às 17:00

Teste com consulta com 1 hora e 40 minutos de duração + 15 minutos de tolerância

Nome:	Num:

Notas: *Este enunciado é constituído por 4 grupos de perguntas. Responda no próprio enunciado, usando a frente e o verso.*
Pode definir funções auxiliares sempre que quiser.
Normalmente, respostas imperfeitas merecem alguma pontuação.
Fraude implica reprovação na cadeira.

1. [4 valores] Escolha múltipla. As respostas erradas não descontam. Indique as respostas mais correctas aqui:

A	B	C

A) Tempo de vida das ligações das variáveis em C.

a) As variáveis dinâmicas são criadas com malloc e depois persistem durante o resto da execução do programa.

b) As variáveis globais e static persistem durante toda a execução do programa; as variáveis locais persistem durante a execução do bloco ou função a que pertencem; as variáveis dinâmicas têm a sua persistência gerida manualmente.

c) As variáveis globais e static persistem durante toda a execução do programa; as variáveis locais persistem durante a execução do bloco ou função a que pertencem; as variáveis dinâmicas têm persistência indefinida, não controlável.

d) Todas as variáveis, independentemente das suas características, persistem durante toda a execução do programa.

B) Gestão automática de memória comparada com gestão manual de memória.

a) A primeira garante maior velocidade de execução dos programas.

b) A primeira garante ausência de memory leaks e de dangling pointers, e atua em tempo de compilação.

c) A primeira garante ausência de memory leaks e de dangling pointers, e atua em tempo de execução.

d) A primeira não permite alocação de memória manual nem libertação de memória manual.

C) O que nos garante um sistema de tipos sem falhas de proteção?

a) Que os programas validados correm sem problemas.

b) Que os programas validados correm sem problemas, desde que a execução termine.

c) Que os programas validados correm sem erros de tipo nem exceções de *null pointer*.

d) Que os programas validados correm sem erros de tipo.

2. [4 valores] Considere o seguinte programa escrito em GCC, uma variante do C que suporta aninhamento de funções:

```
#include <stdio.h>

#define N_NODES  4
typedef struct Node { int value ; struct Node *next ; } Node, *List ;

void G(List l) {     // global function
    for( ; l->next != NULL ; l = l->next );
    // Ψ
}

int main(void) {      // global function
    Node nodes[N_NODES];
    void F(int i) {   // local function
       nodes[i].value = i;
       if( i == N_NODES-1 ) {
          nodes[i].next = NULL;
          G(&nodes[0]);
       }
       else {
          nodes[i].next = &nodes[i+1];
          F(i+1);
       }
    }
    F(0);
    return 0 ;
}
```

Mostre qual o estado da pilha de execução no momento em que a execução do programa atinge o ponto marcado com a letra grega Ψ. Note que o array `nodes` ocupa 8 posições de memória. Não se esqueça dos registos de ativação das funções `start` e `main`.

Use as convenções habituais das aulas: Para efeito da criação do registo de activação inicial, imagine que cada programa em GCC está embebido numa função sem argumentos chamada `start`. Depois trate todas as entidades globais do programa como sendo locais à função imaginária `start`. Assuma também que a primeira célula da pilha de execução é identificada como posição 00, a segunda célula como posição 01, etc.

35	23	11
34	22	10
33	21	09
32	20	08
31	19	07
30	18	06
29	17	05
28	16	04
27	15	03
26	14	02
25	13	01
24	12	00

3. Neste problema vamos trabalhar **com listas de inteiros ordenadas crescentemente, onde podem ocorrer valores repetidos**. Para acelerar a remoção de valores, decidiu-se que **os nós removidos permanecem na lista, na mesma posição, mas com uma flag a indicar que estão inativos**. Para acelerar a insersão de valores, reaproveita-se o nó inativo nos casos em que por sorte esteja um nó inativo ponto de insersão.

Vamos usar o seguinte tipo ANSI-C. Cada nó da lista contém: uma flag que indica se o nó está ativo, o valor do nó (no caso de estar ativo), um apontador para o nó que se segue. O apontador NULL marca o final da lista. A função da direita pode ser usada para criar nós inicializados.

```
typedef struct Node {
    bool active ;
    int value ;
    struct Node *next ;
} Node, *List;
```

```
List newNode(int value, List next)
{
    List n = malloc(sizeof(Node));
    if( n == NULL ) return NULL;
    n->active = true;
    n->value = value;
    n->next = next;
    return n;
```

Importante: Pretendem-se soluções iterativas, portanto sem uso de recursão, e que a lista-argumento seja percorrida apenas uma vez.

a) [2 valores] Escreva em C uma função para determinar o comprimento duma lista. Para o comprimento, contam só os nós ativos.

```
int length(List l) {
```

b) [2 valores] Escreva em C uma função para remover (ou seja, para inativar) a primeira ocorrência dum dado valor numa lista. A função não deve ter ineficiências desnecessárias: concretamente, como a lista está ordenada, ao ser atingido um valor maior ou igual, não vale a pena continuar a pesquisa.

```
List remove(List l, int value) {
```

c) [2 valores] Escreva em C uma função para inserir ordenadamente um valor numa lista. De acordo com o que se disse na introdução, só é preciso intercalar um novo nó se na posição de inserção não existir já um nó inativo disponível. [Resolva nas costas.]

```
List insert(List l, int value) {
```

4. [6 valores] O objetivo deste problema é a definição dum sistema de classes abstratas e concretas adequado à representação de **objetos celestes**, concretamente de galáxias, clusters de estrelas, estrelas, planetas e satélites. No futuro, é provável que queiramos ampliar o sistema introduzindo maior variedade de objetos celestes, digamos universos, clusters de galáxias, buracos negros, etc. Por isso você vai ter ter escrever código bem fatorizado e extensível.

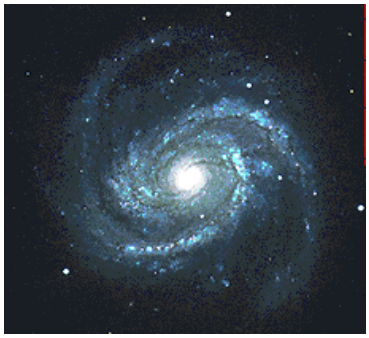

Figura: Galáxia Messier 100

Descrição das entidades: Para simplificar, adoptamos um modelo perfeitamente hierárquico do Cosmos. Vejamos como são os nossos objetos celestes. Uma **galáxia** é uma *coleção* de clusters. Um **cluster** é uma *coleção* de estrelas. Uma **estrela** tem uma *massa* própria e possui uma *coleção* de planetas que giram à sua volta. Um **planeta** tem uma *massa* própria e possui uma *coleção* de satélites que giram em sua volta. Um **satélite** tem uma *massa* própria e possui um *coleção* de (sub)satélites que giram em sua volta. Qualquer das coleções atrás referidas pode ser vazia. Finalmente, cada objeto celeste tem um *nome* (e.g. há uma galáxia que se chama "Messier 100" e um planeta que se chama "Terra").

Desta descrição, resulta a arrumação dos objetos celestes em duas categorias:

- **Objetos agregados**, sem centro físico - galáxias e clusters.
- **Objetos centrados**, que são centros de sistemas orbitais - estrelas, planetas e satélites.

Repare que todos os objetos celestes têm dois atributos em comum: um *nome* e uma *coleção* de outros objetos celestes. Já o atributo *massa* só ocorre nos objetos centrados.

Não existe a categoria dos *subsatélites*. Um subsatélite é simplesmente mais um satélite.

Métodos das entidades: Todos os objetos celestes têm quatro métodos públicos (mas você tem liberdade de definir os métodos auxiliares que precisar).

O método **INIT(…)** inicializa um objeto celeste. Os argumentos variam consoante o tipo de objeto e compete-lhe identificar quais são esses argumentos.

O método **belongs(name)** testa se o nome indicado é o nome do objeto **this** ou de alguma das suas partes.

O método **calcMass()** calcula a massa dum objeto celeste. A massa dum **agregado** é a soma das massas dos objetos participantes. A massa dum **centrado** é a massa do centro mais a massa de todos os objetos em redor.

O método **validate()** testa se um objeto celeste é válido. Como regra geral, um objeto só deve ter na sua coleção objetos de nível hierárquico imediatamente inferior. Os satélites são uma exceção pois têm (sub)satélites a girar à sua volta, ou seja objetos de igual nível. (Para escrever código extensível, pode começar por associar um nível hierárquico (um inteiro) a cada tipo de objeto, mas tenha cuidado para não quebrar a extensibilidade.)

Ajuda: Para representar coleções, use arrays. A primeira linha das várias classes já está escrita:

```
var Sky = EXTENDS(JSRoot, {
```

```
var Agreggate = EXTENDS(Sky, {

var Centered = EXTENDS(Sky, {

var Galaxy = EXTENDS(Agreggate, {

var Cluster = EXTENDS(Agreggate, {

var Star = EXTENDS(Centered, {

var Planet = EXTENDS(Centered, {

var Satellite = EXTENDS(Centered, {
```