Ano Lectivo 2014/2015
Linguagens e Ambientes Programação – Teste 2

02 de Junho de 2015 às 17:00

Open book test with duration of 1 four e 40 minutes + 15 minutes leeway

Nome:	Num:

Notas: *This exam paper consists of four groups of questions. Answerin the statement itself, using the front and the back of the pages.*
You can define auxiliary functions whenever you want.
Normally, imperfect answers deserve somepoints.
Fraud implies failure in the course.

1. [4 points] Multiple choice. Wrong answers do not discount. Indicate the most correct answers here:

A	B	C

A) Lifetime of the bindings of the variables in C.

a) The dynamic variables are created with *malloc* and then persist throughout the rest of the execution of the program.

b) Global variables and static variables persist throughout the execution of the program; local variables persist for the execution of the block or function to which they belong; dynamic variables have their persistence managed manually

c) Global variables and static variables persist throughout the execution of the program; local variables persist for the execution of the block or function to which they belong; dynamic variables have indefinite persistence, not controllable.

d) All variables, regardless of their characteristics, persist during the entire program execution.

B) Automatic memory management (garbage colection) compared to manual memory management.

a) The first ensures greater execution speed of programs..

b) The first ensures the absence of memory leaks and dangling pointers, and operates at compile time.

c) The first ensures the absence of memory leaks and dangling pointers, and operates at runtime.

d) The first does not allow manual memory allocation neither manual memory release.

C) What assures us a type system that is type safe?

a) The validated programs run without problems.

b) That validated programs run without problems as long the execution ends.

c) The validated programs run without type errors and without *null pointer exceptions*.

d) The validated programs run without type errors.

2. [4 points] Consider the following program written in GCC, a variant of C the language that supports nesting functions):

```
#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 ;
}
```

Show the state of the runtime stack at the time that the execution of the program reaches the point marked with the Greek letter Ψ. Note that the array `nodes` occupies eight memory locations. Do not forget the activation records of functions `start` and `main`.

Use the usual conventions: For purposes of first activation record imagine that each program in GCC is embedded in a function called `start`. Then treat all global entities of the program as being local to of the function `start`. Assume also that the first execution stack of the cell is identified as position 00, the second cell as position 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. In this problem we will work with **lists of integers sorted in ascending order, where repeated values can occur**. To speed up the removal of values, it was decided that the **removed nodes remain in the list, at the same position, but with a flag indicating that they are inactive**. To speed up the insertion of values, the inactive nodes are reused when by chance there is an inactive node at the insertion point.

We use the following ANSI-C type. Each node of the list contains: a flag indicating whether the node is active, the node value (in the case it is active), and a pointer to the next node in the sequence. The NULL pointer marks the end of the list. The function at the right can be used to create initialized nodes:

```
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;
}
```

Important: Write iterative solutions, without recursion, and transverse the argument-list only once.

a) [2 points] Program a C a function to determine the length of a list. For the length, only the active nodes count.

```
void length(List l) {
```

b) [2 points] Program a C a function to remove (i.e., to inactivate) the first occurrence of a given value from a list. The function avoid unnecessary inefficiencies: specifically, as the list is sorted, the search does not need to continue then a greater or equal value is reached.

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

c) [2 points] Program a C a function to perform the insertion of a value in an ordered list. Following the introduction, you only need to insert a new node if there is not an inactive node available at the insertion point. [Write your code at the back of this page.]

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

4. [6 points] The aim of this problem is the definition of a system of abstract and concrete classes for the representation of **celestial objects**, particularly galaxies, star clusters, stars, planets and satellites. In the future, it is likely that we need to expand this system with a greater variety of celestial objects, say universes, galaxy clusters, black holes, etc. So you'll have to write well factored and extensible code.

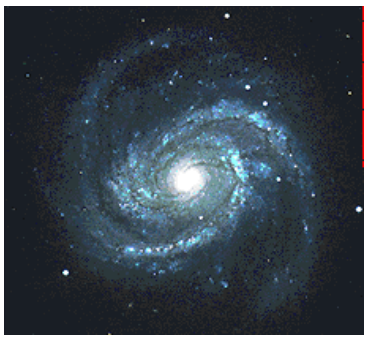

Figura: Galáxia Messier 100

Description of the entities: For simplicity, we adopt a perfectly hierarchical model of the cosmos. Let us see what the attributes of the celestial objects are: A **galaxy** is a *collection* of clusters. A **cluster** is a *collection* of stars. A **star** has its own *mass* and has a *collection* of planets revolving around them. A **planet** has its own *mass* and has a *collection* of satellites that revolve around it. A **satellite** has its own *mass* and has a *collection* of (sub)satellites that revolve around it. Any of the collections mentioned can be empty. To complete the description, each celestial object has a *name* (for example there is a galaxy called "Messier 100" and a planet called "Earth".)

From this description, results that the celestial objects can be placed in two categories:

- **Agreggate** objects, without phisical center – galaxies and clusters.
- **Centered**,objects, that are centers of orbital systems - stars, planets and satellites.

Note that all celestial objects have two attributes in common: a *name* and a *collection* of other celestial objects. The *mass* attribute only occurs in the centered objects.

There is no category of *subsatellites*. A subsatellite is simply another satellite

Methods: All celestial objects have four public methods (but you are free to define helper methods if necessary).

The method **INIT (...)** initializes a celestial object. The arguments vary according to the type of object.

The method **belongs (name)** tests whether the given name is the name of the object **this** or of any of its parts.

The method **calcMass()** determines the mass of a celestial object. The mass of an **aggregate** is the sum of the masses of the participating objects. The mass of a **centered** is the mass of the center plus the mass of all the objects around.

The method **validate()** tests whether a celestial object is valid. As a rule, an object should only own objects of the hierarchical level immediately below. Satellites are an exception as they own the (sub)satellites spinning around, ie. objects of the same hierarchical level. (To write extensible code, you can start by associating a hierarchical level (an integer) for each type of object, but be careful to make the code really extensible.)

Help: Use arrays to represent collections. We already provide the first line of the various classes.

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

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

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

```
var Galaxy = EXTENDS(Agreggate, {
```

```
var Cluster = EXTENDS(Agreggate, {
```

```
var Star = EXTENDS(Centered, {
```

```
var Planet = EXTENDS(Centered, {
```

```
var Satellite = EXTENDS(Centered, {
```