Année universitaire 2016-2017 – L2 informatique
Code UE : 21IM32
Durée : 2 heures

Le sujet comporte trois pages.

Consignes : aucun document autorisé, calculatrices et téléphones portables interdits, dictionnaires papiers autorisés.

Architecture des systèmes
Examen terminal – deuxième session

Partie 1 – Circuits logique (7 points)

Le but de cette question est la réalisation d'un circuit logique permettant de tester si deux entiers *A* et *B* représentés en RBNS (représentation binaire non signée) sur 2 bits, vérifient la propriété suivante : *A* est le double de *B*, c'est-à-dire *A=2B*.

Exercice 1.1 : Donnez la fonction booléenne sous les deux formes (somme des produits et produit des sommes) associée à ce circuit.
Exercice 1.2 : Simplifiez la fonction obtenue précédemment.
Exercice 1.3 : Donnez le schéma du circuit logique correspondant.

Partie 2 – Programmation assembleur (7 points)

Exercice 2.1 (4 points) : Programmez en assembleur la fonction suivante, sans changer l'algorithme :

```
void f() {
   int i = 0;
   int j = 0;
   do {
      j = j+i-1;
      if (j>100) {
         break;
      }
   } while (i<15);
}
```

Exercice 2.2 (3 points) : Le programme assembleur suivant devrait afficher un rectangle de 5x7 fois le caractère 'o'. Ce programme comporte cinq erreurs. Corrigez chaque erreur.

```
section .data
caractere db 'o', 10
section .text
global _start
_start:
   mov ecx, 5
   boucle1:
```

```
    push ecx
    mov ecx, 7
    boucle2:
       push ecx
       mov eax, 4
       mov ebx, 1
       mov ecx, caractere
       mov edx, 1
       int 80h
       dec ecx
       cmp ecx, 0
       jbe boucle2
    mov eax, 4
    mov ebx, 1
    mov ecx, caractere
    mov edx, 1
    int 80h
    dec ecx
    cmp ecx, 0
    jbe boucle1
mov eax, 1
xor ebx, ebx
int 80h
```

Partie 3 – Programmation avancée en C (6 points)

En C, le mécanisme usuel pour définir des modules est de disposer d'un fichier header pour l'interface et d'un fichier pour l'implémentation des fonctions de l'interface. Cependant, cette modularité mal implémentée peut avoir certaines limites, dont la plus critique est souvent la non-possibilité de disposer de plusieurs instances (sans conflit) du même objet.

En programmation objet, on distingue dans une instance d'un objet son *état* (souvent représenté par des variables locales) de son *comportement* (souvent la liste des fonctions que l'on peut utiliser avec cet objet). Cet ensemble est encapsulé dans ce que l'on appelle une *classe* et les instances d'une classe sont appelées les *objets*. Un exemple d'objets ce sont les complexes où l'état d'un complexe est défini par les parties réelle et imaginaire, et les méthodes que l'on peut associer à un complexe sont l'addition avec un autre complexe, la multiplication avec un autre complexe, calculer son module, sa forme exponentielle, son inverse, sa conjuguée, etc.

Exercice 3.1 (0.5 point) : Donnez un exemple de structure de données (que nous notons ci-dessous `Type`) en spécifiant ce qui définit son état et ce qui définit son comportement.

En C, le mécanisme usuel pour encapsuler plusieurs variables dans une seule est l'utilisation de `struct`. Cependant, comme le C est fortement typé, il faut que tout type créé avec `struct` soit exporté dans toutes les parties du programme où il est utilisé. Or, la définition dépend fortement de l'implémentation, et c'est ce que l'on veut masquer. Une façon de faire est de rendre public le nom d'un type qui référence un pointeur sur le type réel.

Exercice 3.2 (1 point) : Par souci de factorisation, écrivez un header `classe.h` qui contient une macro `CLASSE` qui prend en paramètre un nom de classe `idClasse` et qui crée un type `idClasse` synonyme de `struct idClasse*`.
Exercice 3.3 (0.5 point) : Proposez un header qui utilise la macro `CLASSE` et définit l'interface de `Type`.

Nous allons maintenant discuter de l'implémentation. Comme `CLASSE` crée des références aux pointeurs, nos modules vont manipuler des pointeurs. Par souci de factorisation, on va également écrire une macro pour s'occuper des instantiations.

Exercice 3.4 (1 point) : Proposez un header `constructeur.h` contenant la macro `CONSTRUCTEUR` qui prend en paramètre un nom de classe `idClasse` et qui alloue un objet de type `idClasse`.
Exercice 3.5 (1 point) : Sans préciser le contenu des fonctions de manipulation de `Type` (sauf celle d'initialisation), donnez le contenu du fichier `.c` correspondant à l'implémentation de `Type`.

Nous venons de proposer une méthode pour faire de la programmation objet avec les pointeurs et les macros. Cependant, notre implémentation présente le défaut que nous ne libérons pas les espaces alloués avec la macro `CONSTRUCTEUR`.

Exercice 3.6 (1 point) : Soit `o` un objet alloué dynamiquement et supposons qu'il référence d'autres objets alloués dynamiquement. Quel est le problème éventuel de `free(o)` ?
Exercice 3.7 (1 point) : Proposez une façon d'intégrer la libération (propre) de l'espace alloué dans les *classes* que nous implémentons.