Université libre de Bruxelles

Faculté des Sciences
Départment d'informatique

Langages de programmation

(INFO-F105)

Première année du programme de bachelier en Science Informatique

Examen de septembre 2021

- L'examen comprend quatre questions, chacune notée sur 10 points
- La pondération associée à chaque sous-question est fournie
- Répondez à chaque question sur une feuille séparée fournie à cet effet
- Indiquez vos nom, prénom et numéro d'étudiant sur chaque feuille de réponse
- Toutes les notes de cours, transparents et syllabus sont autorisés
- Calculatrices, ordinateurs, téléphones interdit
- Durée de l'examen : 2:00
- Pour rappel, la note de cours est consituée de soit
  - 2/3 examen + 1/3 projet
  - 100% examen

  (la note la plus favorable des deux)

Question 1

Considérons le code en assembleur NASM x86 muni des macros SASM ci-dessous :

```
1  %include "io.inc"
2
3  SECTION .data
4  m DW 637
5  n DW 364
6
7  SECTION .text
8  GLOBAL CMAIN
9  CMAIN:
10     MOVZX EAX, WORD [m]
11     MOVZX EBX, WORD [n]
12 etiquette1:
13     CMP EBX, 0
14     JE etiquette2
15     MOV EDX, 0
16     DIV EBX
17     MOV EAX, EBX
18     MOV EBX, EDX
19     JMP etiquette1
20 etiquette2:
21     PRINT_UDEC 4, EAX
22     XOR EAX, EAX
23     RET
```

1. Décrivez en une phrase ce que fait ce code. **[2 points]**

2. Qu'affiche le PRINT_UDEC à la ligne 21 ? **[1 point]**

3. Peut-on remplacer la ligne 14 par :

   ```
   JZ etiquette2
   ```

   Justifiez votre réponse. **[2 points]**

4. Peut-on remplacer les lignes 10-11 par :

   ```
   MOV EAX, [m]
   MOV EBX, [n]
   ```

   Justifiez votre réponse. **[2 points]**

5. Sur la page de réponse, complétez la fonction `main` (en C++) qui effectue **le même traitement** que ce code assembleur. **[3 points]**

Question 2

Étant donné le code C++ des deux pages suivantes, répondez **brièvement** (2-3 lignes maximum) aux questions suivantes :

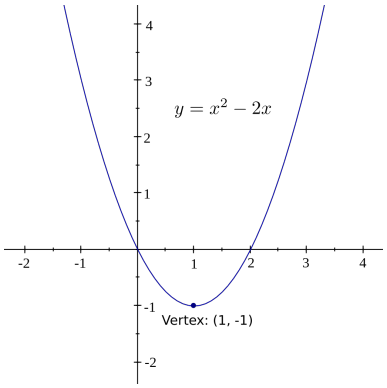
1. Indiquez quelle sera la valeur affichée pour `b` à la ligne n°24 ? **[1 point]**
2. Indiquez quelle sera la valeur affichée pour `c` à la ligne n°28 ? **[1 point]**
3. Y a-t-il une différence dans la manière de passer le paramètre à la `fonction1` et à la `fonction2` ? Expliquez brièvement en vous référant à la théorie. **[2 points]**
4. Peut-on supprimer purement et simplement les lignes n°3 et n°4, compte tenu que les `fonction1` et `fonction2` sont définies aux lignes n°56 et n°17 respectivement ? Justifiez brièvement votre réponse. **[1 point]**
5. Quelles sont les 3 valeurs affichées par la boucle `for` (lignes 33-37) ? Donnez également les valeurs des variables `d`, `e`, `i` aux lignes 38-40. Comment appelle-t-on en théorie ce qui est arrivé à la variable `d` ? **[2 points]**
6. Quelles vont être les valeurs affichées par la boucle `for` (lignes 43-45) ? **[1 point]**
7. En supposant que la valeur donnée au paramètre `x` de la `fonction3` soit bien de type `unsigned int`, indiquez quelle sera la plus petite valeur possible affichée par `y` à la ligne 14 ? **[1 point]**
8. En supposant que la valeur v donnée en entrée au paramètre `x` de `fonction3` soit bien de type `unsigned int`, indiquez la ou les affirmations ci-dessous qui s'avère(nt) correcte(s). **[1 point]**
   (a) Pour tout $v \geq 0$, `fonction3` affiche le plus petit exposant entier y qui satisfait $2^y > v$.
   (b) Pour tout $v \geq 0$, `fonction3` affiche le plus petit exposant entier y qui satisfait $2^y \geq v$.
   (c) Pour tout $v \geq 0$, `fonction3` affiche le plus grand exposant entier y qui satisfait $2^y < v$.
   (d) Pour tout $v \geq 0$, `fonction3` affiche le plus grand exposant entier y qui satisfait $2^y \leq v$.
   (e) Pour tout $v > 0$, `fonction3` affiche le plus petit exposant entier y qui satisfait $2^y > v$.
   (f) Pour tout $v > 0$, `fonction3` affiche le plus petit exposant entier y qui satisfait $2^y \geq v$.
   (g) Pour tout $v > 0$, `fonction3` affiche le plus grand exposant entier y qui satisfait $2^y < v$.
   (h) Pour tout $v > 0$, `fonction3` affiche le plus grand exposant entier y qui satisfait $2^y \leq v$.
   (i) La valeur finale de `y` dépend non seulement de la valeur initiale de `x` mais aussi du nombre de bits alloués en machine à la mantisse des nombres en virgule flottante.

```
1  #include<iostream>
2
3  int fonction1(int b);
4  void fonction2(int &c);
5
6  void fonction3(unsigned int x) {
7      unsigned int y=0;
8
9      do {
10         x/=2;
11         y+=1;
12     } while (x>0);
13
14     std::cout << "y vaut : " << y << std::endl;
15 }
16
17 void fonction2(int &c) {
18     c+=1;
19 }
20
21 int main() {
22     int b=7;
23     fonction1(b);
24     std::cout << "b vaut : " << b << std::endl;
25
26     int c=7;
27     fonction2(c);
28     std::cout << "c vaut : " << c << std::endl;
29
30
31     int d=0, e=1, i=1;
32
33     for (int i=0; i<3; i++) {
34         int d=2;
35         std::cout << d*e*i << ' ';
36         e+=1;
37     }
38     std::cout << std::endl << "d vaut : " << d
39             << " ; e vaut : " << e
40             << " ; i vaut : " << i << std::endl;
41
42
43     for (int j=4; j<8; j++) {
44         std::cout << (j & 0b11) << ' ';
45     }
```

```
46      std::cout << std::endl;
47
48      unsigned int x;
49      std::cout << "Introduisez un entier :" << std::endl;
50      std::cin >> x;
51      fonction3(x);
52
53      return 0;
54  }
55
56  int fonction1(int b) {
57      return b+=1;
58  }
```

Question 3

[10 points]



Une parabole est le lieu géométrique de tous les points situés à égale distance d'une droite fixe appelée directrice et d'un point fixe appelé foyer. Une parabole *verticale* est identifiée par l'équation suivante :

$$y = ax^2 + bx + c.$$

Le sommet de la parabole est situé aux coordonnées cartésiennes suivantes:

$$V = \left(-\frac{b}{2a}; \frac{-(b^2 - 4ac)}{4a} \right).$$

Le type requis pour a, b, c est **double**.
On vous demande

1. (a) d'implémenter une classe **Parabole** avec:
   - un ou plusieurs constructeurs (par défaut $a = 1.0$ et $b = 0.0$ et $c = 0.0$)
   - un destructeur
   - les getters
   - les setters (sans tests)
   - une méthode
     `value_y`
     qui prend comme paramètre un entier x et renvoie la valeur de y correspondent.
   - une méthode
     `vertex_x`
     qui renvoie uniquement la coordonnée x du sommet V.

   (b) de surcharger << pour pouvoir faire

   ```
   cout << par << endl;
   ```

   avec *par* une instance de **Parabole**. Si *par* est $y = 2x^2 - 8x + 4$, l'output affichera a, b, c et **les deux** coordonnées du sommet:

   ```
   Par = [2,-8,4]; V = (2,-4)
   ```

Remarque : veillez à utiliser judicieusement les mots clés vus tels que **public**, **private**, **const** et les références.

Question 4

Considérez le code ci-dessous, écrit en C++.

1. Expliquer l'effet des lignes 6 et 7. Est-ce que les valeurs de b et c sont égales après la ligne 7 ? Justifiez vos réponses. **[1 point]**

2. Qu'est-ce que le programme affiche en ligne 11 et 12? Justifiez votre réponse. **[1 point]**

3. Les lignes 14 et/ou 15 produiront-elles des erreurs à la compilation et/ou à l'exécution? Donnez et justifiez vos réponses pour les deux lignes et les deux types d'erreurs. **[2 points]**

4. Supposons que fool soit appelée avec un paramètre $n = 50$. Quelle sera la valeur de f après exécution de la ligne 19 ? **[1 point]**

5. La ligne 23 cause un problème bien connu en C/C++. Citez ce problème et expliquez pourquoi il s'applique ici. **[1 point]**

6. La ligne 24 produira-t-elle une erreur à la compilation et/ou à l'exécution si on la décommente ? **[1 point]**

7. Modifiez une ligne de code de la fonction *fool* pour causer un *buffer overflow*. **[1 point]**

8. Supposez que la fonction *fool* soit appelée un grand nombre de fois avec une valeur de n très grande. Quel(s) pourrai(en)t être le/les effet(s) sur le programme à la compilation et l'exécution ? Justifiez votre réponse. **[2 points]**

```
1  #include <iostream>
2  using namespace std;
3
4  void fool(int n) {
5      int a = 5;
6      int* b = &a ;
7      int* c = new int(a) ;
8
9      int* d = c;
10     *c = 10;
11     cout<<"a="<<a<<endl<<"*b="<<*b<<endl;
12     cout<<"*c="<<*c<<endl<<"*d="<<*d<<endl;
13     delete c;
14     *d = 15;
15     *c = 20;
16
17     int* e = new int[n] ;
18     for(int cnt=0; cnt<n; cnt++) {e[cnt]=cnt*cnt;}
19     int f = *(&((e+2)[-1])+2)+7;
20
21     int* g = new int[n] ;
22     for(int cnt=0; cnt<n; cnt++) {e[cnt]=2*cnt;}
23     g=b ;
24     // delete g;
25 }
```

Examen INFO-F105 Langages de Programmation 1 ; examen de septembre 2021

Nom, prénom:
Numéro d'étudiant:

Numéro de question: 1

---

1. Ce code est une implémentation de l'algorithme d'Euclide qui calcule le GCD entre les nombres stockés aux adresses `n` et `m`.

2. À la fin de l'exécution (l. 21), le registre `EAX` contient le le GCD entre `n` et `m` (i.e. 91) et `EBX` est à zéro.

3. Oui : `JE` et `JZ` sont associés à la *même* opcode et sont donc interchangeables. `JE` reste l'instruction adaptée conceptuellement dans le cas d'un saut conditionnel suivant un `CMP`.

4. Non : les quantitiés stockées aux adresses `n` et `m` sont des `WORD` (donc des entiers sur 16 bits) alors que `EAX` et `EBX` sont des registres de 32 bits. Le remplacement proposé ne poserait pas de problème à la compilation mais bien à l'exécution car les registres ne contiendraient pas les quantitiés attendues.

```
1  #include <iostream>
2
3  int main() {
4      uint16_t m = 637, n = 364;
5      while(n != 0) {
6          const uint16_t r{m%n};
7          m = n;
8          n = r;
9      }
10     std::cout << m << std::endl;
11     return 0;
12 }
```

Voici une autre implémentation possible (ne faisant pas intervenir la variable `r` qui stocke temporairement le reste de la division euclidienne de `m` par `n`.

```
1  #include <iostream>
2
3  int main() {
4      uint16_t m = 637, n = 364;
5      while(n != 0) {
6          m %= n;
7          std::swap(m, n);
8      }
```

```
 9     std::cout << m << std::endl;
10     return 0;
11 }
```

Examen INFO-F105 Langages de Programmation 1 ; examen de septembre 2021

Nom, prénom:
Numéro d'étudiant:

Numéro de question: 2

---

1. `b` vaut 7

2. `c` vaut 8

3. Oui, il y a une différence. Pour `fonction1`, il s'agit d'un passage par valeur (une copie de `b` est utilisée localement). Pour `fonction2`, il s'agit d'un passage par référence (la modification de `c` dans `fonction2` sera ensuite observable dans la fonction appelante (ici: `main`)).

4. La ligne 4 peut être supprimée car `fonction2` est définie avant `main`. En revanche, la déclaration de `fonction1` en ligne 3 est bien nécessaire car `fonction1` n'est définie qu'après le `main()`.

5. - les 3 valeurs affichées par la boucle for (lignes 33-37) sont: `0 4 12`. En effet, `d` vaut 2 à chaque itération, `e` vaut successivement $1, 2, 3$ au moment de l'impression ligne 35 et `i` vaut successivement $0, 1, 2$ au moment de l'impression ligne 35.
   - les valeurs des variables d, e, i aux lignes 47-49 : d vaut 0; e vaut 4 et i vaut 1. Les variables `d` et `i` disponibles seulement dans le scope de la boucle ne doivent pas être confondues avec les variables `d` et `i` déclarées à la ligne 31.
   - C'est un shadowing.

6. `0 1 2 3`. `0b11` est un masque qui ne préservera, après usage de l'opérateur `&` que les deux bits de poids faibles de la variable `j`. Cela revient ici à calculer le modulo de divisions par 4.

7. `y` vaut 1. Le code entre `{}` est toujours lu une fois minimum dans le `do-while`.

8. Seule l'affirmation (e) est correcte. L'affirmation (i) est incorrecte car seule des valeurs entières sont impliquées. A partir du moment où `x` est un entier, `x/=2` ne peut pas donner un nombre à virgule en C++. Les affirmations (a) et (b) sont incorrectes car `y` ne peut être inférieur à 1 au moment de l'affichage, or, dans les propositions (a) et (b), pour $v = 0$, il est possible d'avoir un exposant 0 plus petit que la valeur affichée de `y` et qui peut donc satisfaire $2^0 > 0$ ou $2^0 \geq 0$.

   En fait, pour $v > 0$, la variable `y` comptera le nombre de fois que `x` peut être divisé (en division entière) par 2 avant de valoir 0. Par exemple $v = 4$ pourra être divisé 3 fois par 2 avant de valoir 0 (`x` vaudra successivement 4, 2, 1, 0). Dans notre exemple, après le `do-while`, `y`=3 et $2^3 > 4$. Cela élimine les propositions (c), (d), (g) et (h). Dans le `do-while`, on réalise des divisions successives de `x` par 2 et la valeur

$v > 0$ sera bornée par une puissance de 2 fournie grâce au compteur d'iterations `y`. Procédons par récurrence: Pour $v = 1 = 2^0$ une division par 2 sera nécessaire avant d'obtenir 0. Pour $v = 2 = 2^1$, deux divisions seront nécessaires avant d'obtenir 0. Pour $v = 4 = 2^2$, trois $(2 + 1)$ divisions seront nécessaires avant d'obtenir 0. Pour $v = 2^y$, $(y + 1)$ divisions seront nécessaires avant d'obtenir 0. La proposition (f) peut donc être écartée. Seule la (e) est correcte.

Examen INFO-F105 Langages de Programmation 1 ; examen de septembre 2021

Nom, prénom:
Numéro d'étudiant:

Numéro de question: 3

```
#include <iostream>

class Parabole {
private :
    double _a;
    double _b;
    double _c;
public :
    Parabole(double a=1.0, double b=0.0, double c=0.0):
    _a(a), _b(b), _c(c){}
    ~Parabole() = default;
    double getA() const{ return _a;}
    double getB() const{ return _b;}
    double getC() const{ return _c;}
    void setA(const double &a) { _a = a;}
    void setB(const double &b) { _b = b;}
    void setC(const double &c) { _c = c;}
    double value_y(double x) const {return _a*x*x+_b*x+_c; }
    double vertex_x() const {return -(_b)/(2*_a); }
};

std::ostream& operator<<(std::ostream &s, const Parabole &p){
    s <<"Par = ["<< p.getA() << "," << p.getB() << ","  <<
    p.getC() << "]; V = (" << p.vertex_x() << "," <<
    p.value_y(p.vertex_x()) << ")";
    return s;
}

int main(){
    Parabole par;
    std::cout << par << std::endl;
    par.setA(2.0);
    par.setB(-8.0);
    par.setC(4.0);
    std::cout << par << std::endl;
    return 0;
}
```

Examen INFO-F105 Langages de Programmation 1 ; examen de septembre 2021

Nom, prénom:
Numéro d'étudiant:

Numéro de question: 4

---

1. La ligne 6 déclare et définit un pointeur, et initialise sa valeur à l'adresse de a ; la ligne 7 crée une nouvelle variable sur le tas (de façon dynamique) et lui donne la valeur de a. Les valeurs de b et c (adresses pointées) seront bien sûres différentes

2.

```
1 a=5
2 *b=5
3 *c=10
4 *d=10
```

3. La ligne 13 désalloue la mémoire sur le tas mais elle ne supprime pas le pointeur c, et la mémoire pointée existe bien sûr toujours (même si sa valeur est indéfinie à ce stade jusqu'à une réallocation éventuelle). Il n'y aura pas d'erreur à la compilation.

   Strictement parlant, le comportement est alors indéfini, et dépendra de la gestion par l'OS des zones mémoires désallouées. Il est donc encore possible de manipuler le pointeur et la zone de mémoire pointée. Il n'y aura a priori pas non plus d'erreur à l'exécution de ces lignes, mais elles pourraient indirectement causer d'autres erreurs à l'exécution dans certains cas, par exemple si la mémoire est réallouée entre le delete et l'assignation (hautement improbable ici vu que les lignes sont consécutives)

4. 16 = e[3]+7

5. Problème de fuite de mémoire : la mémoire allouée sur le tas en ligne 21 n'est plus accessible

6. Pas d'erreur à la compilation mais on pourrait avoir une erreur à l'exécution, car on tentera de désallouer de la mémoire non allouée de façon dynamique (sur la pile ici)

7. La solution n'est pas unique; on peut par exemple remplacer `cnt < n` par `cnt <= n`

8. Il y a beaucoup de mémoire allouée sur le tas, qui n'est jamais désallouée (pas de *delete* correspondant aux *new* des deux tableaux e et g). Il y a donc un risque d'exhaustion de la mémoire. (Avant que celui-ci se produise, il est probable aussi que la mémoire libérée en ligne 13 soit réallouée, mais ceci a peu de chance de causer un problème car les itérations successives sont bien distinctes.) Ceci n'amènera aucune erreur à la compilation, mais pourrait mener à des erreurs à l'exécution pour de très grandes valeurs de n.

De façon générale, les points ont été attribués d'avantage sur base des justifications fournies que sur base des "réponses" (vrai/faux, oui/non, 16). Certaines réponses correctes mal justifiées ont donc pu amener une note réduite, alors que des réponses fausses mais démontrant une bonne compréhension de la matière ont valu beaucoup de points.
Parmi les erreurs fréquentes, beaucoup se rapportent au concept même de pointeur et/ou à son utilisation, et montrent des lacunes de compréhension quant à l'utilisation et l'effet des new et delete; voir les sections du syllabus / transparents / exercices qui se rapportent à ces concepts.