

HLIN302 – Examen n° 1



Programmation impérative avancée
Alban MANCHERON et Pascal GIORGI
[Durée 2h – Documents autorisés]

Important

Avant toute chose, il convient de rappeler certaines règles et bons usages relatifs aux épreuves. Tout d'abord, le barème n'est fourni qu'à titre indicatif. Ensuite, seules vos notes de cours, de TD et de TP sur support papier sont autorisées. Il est bien entendu que les documents des voisins et autres collègues ne font pas partie de l'ensemble des documents autorisés. Les ordinateurs, smartphones, tablettes, calculatrices, ..., sont interdits. De surcroît, seul l'enseignant est habilité à répondre à vos questions pendant l'épreuve. Il vous est par ailleurs conseillé de lire attentivement le sujet dans son intégralité.

1 C'est bidon [2 points]

Fichier bidon.h

```
1 class Bidon {
2     private:
3         unsigned int n;
4     public:
5         Bidon();
6         Bidon(unsigned int n);
7         unsigned int getN();
8     };
9     void PrintBidon(const Bidon &b);
10    void ModifBidon(Bidon &b);
```

Fichier bidon.cpp

```
1 #include "bidon.h"
2 #include <iostream>
3 using namespace std;
4 Bidon::Bidon():n(-1) { }
5 Bidon::Bidon(unsigned int n):n(n) { }
6 unsigned int Bidon::getN() { return n; }
7 void Bidon::setN(unsigned int n) { this->n = n; }
8 void PrintBidon(const Bidon &b) { cout << "getN() = " << b.getN() << endl; }
9 void ModifBidon(Bidon &b) { b.n = 3; }
10 int main(int argc, char** argv) {
11     Bidon b1, b2(2);
12     ModifBidon(b1);
13     PrintBidon(b1);
14     PrintBidon(b2);
15     return 0;
16 }
```

1. La compilation du code précédent provoque deux erreurs.

- (a) Que faut-il modifier dans le code de *bidon.h* pour supprimer la première erreur tout en conservant le résultat attendu (ligne 7) ?
- (b) Que faut-il modifier dans le code de *bidon.cpp* pour supprimer la seconde erreur tout en conservant le résultat attendu (ligne 9) ?
2. Une fois les deux erreurs corrigées, écrire ce qu'affiche l'exécution du binaire produit par la compilation sur le terminal.

2 Euclide a dit... [6 points]

En des temps très anciens, le mathématicien Euclide a démontré qu'étant donnés deux entiers naturels a et b avec b non nul, il existe un unique couple d'entiers naturels q et r tel que

$$a = q \times b + r \quad \text{avec } r < b \quad (1)$$

Ce théorème est aujourd'hui connu sous le nom de « théorème de la division euclidienne pour les entiers naturels ». Dans cette équation, a , b , q et r sont respectivement appelés le dividende, le diviseur, le quotient et le reste. Lorsque le reste r est nul, on dit que a est un multiple de b et que b divide a (b est un diviseur de a).

1. En vous basant sur l'équation (1) ci-dessus, écrire la fonction C++ *afficheDiviseursTries* qui prend en entrée un entier positif ou nul n et qui affiche sur la sortie standard tous les entiers qui divisent n dans l'ordre croissant.
2. En ordre de grandeur, combien d'opérations sont effectuées par cette fonction ?

Il est possible de remarquer que lorsque le reste est nul dans l'équation (1) (*i.e.*, $a = b \times q$), alors b et q sont tous deux des diviseurs de a .

3. En exploitant la remarque précédente, écrire la fonction C++ *afficheDiviseurs* qui prend en entrée un entier positif ou nul n et qui affiche sur la sortie standard tous les entiers qui divisent n (sans contrainte d'ordre) et qui soit plus efficace (en nombre d'opérations).
4. En ordre de grandeur, combien d'opérations sont effectuées par cette fonction ?

On s'intéresse dorénavant, étant donné un entier naturel n , à la recherche du couple de diviseurs v_1 et v_2 tel que $n = v_1 \times v_2$ et que la distance entre v_1 et v_2 soit minimale (plus formellement : $\nexists(v'_1, v'_2) \in \mathbb{N}^2, (v_1 \times v_2 = v'_1 \times v'_2 = n) \wedge (|v'_1 - v'_2| < |v_1 - v_2|)$).

Un tel couple correspond en réalité à la recherche des dimensions du rectangle dont l'aire vaut n et qui se rapproche le plus possible d'un carré.

Fichier quasiCarre.h

```

1 class QuasiCarre {
2     private:
3         unsigned int w, h;
4     public:
5         QuasiCarre(unsigned int n);
6         unsigned int getWidth() const;
7         unsigned int getHeight() const;
8         unsigned int getArea() const;
9     };

```

5. Écrire l'implémentation correspondant à la déclaration de la classe *QuasiCarre*, dont les objets se définissent à partir d'un entier positif ou nul n et disposent de trois méthodes permettant respectivement de récupérer la largeur (*getWidth*), la longueur (*getHeight*) et l'aire (*getArea*) du *QuasiCarre* (d'aire n tel que la largeur et la longueur sont les plus proches possibles).

6. Est-il possible d'écrire la déclaration d'une classe *QuasiCarreBis*, disposant exactement des mêmes méthodes que la classe *QuasiCarre* et ayant le même comportement, mais ne nécessitant qu'un seul attribut de type **unsigned int**? Le cas échéant, reportez le tableau suivant avec des + et des - pour montrer la meilleure solution :

	classe <i>QuasiCarre</i>	classe <i>QuasiCarreBis</i>
Espace utilisé par un <i>QuasiCarre(n)</i>	+	-
Temps de calcul de <i>QuasiCarre(n)</i>	+	-
Temps de calcul de <i>getWidth()</i>	-	+
Temps de calcul de <i>getHeight()</i>	-	+
Temps de calcul de <i>getArea()</i>	-	+

3 Sudoku [12 points]

Sudoku (source <http://fr.wikipedia.org/wiki/Sudoku>)

Le sudoku, est un jeu en forme de grille défini en 1979 par l'Américain Howard GARNES, mais inspiré du carré latin, ainsi que du problème des 36 officiers du mathématicien suisse Leonhard EULER.

Le but du jeu est de remplir la grille avec une série de chiffres (ou de lettres ou de symboles) tous différents, qui ne se trouvent jamais plus d'une fois sur une même ligne, dans une même colonne ou dans une même sous-grille. La plupart du temps, les symboles sont des chiffres allant de 1 à 9, les sous-grilles étant alors des carrés de 3×3 . Quelques symboles sont déjà disposés dans la grille, ce qui autorise une résolution progressive du problème complet.

Deux enseignants – appelons les Alain et Patrice – discutent d'un sujet d'examen pour leurs étudiants. Alain propose à son collègue de s'intéresser au jeu du Sudoku, ce qui permettra d'aborder les concepts vus en cours et en TD/TP sur la programmation impérative objet en C++. Patrice trouve l'idée intéressante mais lorsqu'Alain lui propose de réutiliser le concept de matrice creuse vu lors des TD/TP, Patrice lui explique que l'utilisation d'une matrice creuse ici n'a pas de sens et après argumentation, Alain est bien obligé d'admettre son erreur.

1. Selon vous, quel argument Patrice a-t-il mis en avant pour convaincre son collègue ?

Alain propose alors de définir une classe *Sudoku* permettant de définir des grilles génériques à n symboles. Pour cela, il faut que chaque ligne, chaque colonne et chaque zone compte exactement n cases.

En utilisant la classe *QuasiCarre* (ou *QuasiCarreBis*), il est possible de définir les différentes zones en remarquant qu'il suffit de créer une instance q de la classe *QuasiCarre* (*QuasiCarre q(n)*) et n zones de dimension $q.getWidth() \times q.getHeight()$, en les agençant sur $q.getHeight()$ lignes et $q.getWidth()$ colonnes.

Voici deux exemples simples pour vous aider à comprendre :

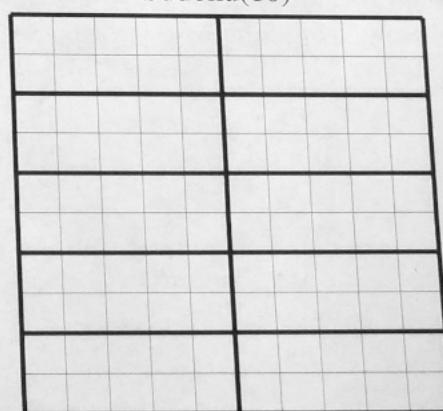
Sudoku(9)



Sudoku(9).getWidth() ⇒ 3

Sudoku(9).getHeight() ⇒ 3

Sudoku(10)



Sudoku(10).getWidth() ⇒ 2

Sudoku(10).getHeight() ⇒ 5

Il est donc possible de représenter une grille de Sudoku par une matrice carrée de dimension n où chaque case est un symbole. Ici nous représenterons les symboles par des caractères ASCII.

2. Écrire la déclaration minimale de la classe *Sudoku* permettant de construire dynamiquement une grille vide pour n symboles (caractères). Toute méthode ou tout attribut superflu sera sanctionné, donc chaque méthode ou attribut fourni devra être clairement justifié.
3. On souhaite pouvoir gérer le niveau de difficulté des grilles lorsqu'elles sont créées au hasard (niveaux débutant, confirmé ou expert). Proposer un mécanisme permettant la gestion du niveau. Modifier la déclaration de la classe en conséquence.
4. Écrire l'implémentation correspondant à la déclaration de votre classe (les cases « vides », seront remplies avec le caractère '0'). Attention, votre classe n'initialisera pas le jeu pour l'instant.

On considérera que cette classe dispose de méthodes permettant :

- de connaître le niveau de difficulté de la grille (*getDifficulty*);
- de changer le niveau de difficulté de la grille (*setDifficulty*);
- d'initialiser une partie au hasard (*init*);
- d'initialiser une partie à partir d'un flux d'entrée (*init*);
- d'afficher la grille en cours sur un flux de sortie (*print*);
- de dire si la grille est terminée (toutes les cases sont remplies – *isCompleted*);
- de vérifier, étant donné les coordonnées d'une case ainsi qu'un symbole, si celui-ci est valide (le caractère fait partie des symboles valides, il n'est pas déjà présent ni sur la ligne, ni sur la colonne, ni dans la zone en cours – *isValid*);
- de remplir une case donnée avec un symbole donné (*set*).

et que vous pouvez utiliser ces méthodes dans la suite.

5. Écrire l'implémentation des deux dernières méthodes ci-dessus (*isValid* et *set*). On supposera que le caractère '0' est utilisé pour dénoter une case « vide », et que les symboles valides sont compris entre le caractère '1' et le caractère de code ASCII '0' + n . Si le symbole n'est pas valide à cette position, la méthode *set* devra mettre fin au programme.

Bonus [3 points]

6. Déclarer et implémenter les surcharges des opérateurs d'entrée/sortie d'une instance de la classe *Sudoku* depuis/vers un flux.
7. Écrire la déclaration minimale de la classe paramétrique *Sudoku* permettant de s'affranchir de l'allocation dynamique pour construire une grille vide de n symboles, tel que n est le paramètre du template. Toute méthode ou tout attribut superflu sera sanctionné, donc chaque méthode ou attribut fourni devra être clairement justifié.
8. Expliquer ce qu'il faut modifier dans la classe pour que le type des symboles utilisés soit générique. Préciser quels sont les prérequis sur le type des symboles.

Bon Courage...