NOM :

PRÉNOM :

PLACE :

université PARIS-SACLAY

Correction partiel de programmation modulaire

—Licence MI/IM - Info 3—

Aucun document n'est autorisé à part la fiche résumé de *C++*, où vous pouviez consigner des notes manuscrites personnelles au verso. Tous les exercices sont indépendants. Même si l'on ne sait pas répondre à une question, on peut utiliser la réponse dans la suite de l'exercice. Une grande importance sera accordée à la qualité de la rédaction (lisibilité, indentation,...).

Répondre dans les cadres fournis. Si vous avez besoin de place supplémentaire, il y a une page libre à la fin du document. Dans ce cas, précisez-le toujours dans le cadre prévu pour la réponse.

Le barème est indicatif et pourra changer à la correction.

Durée : 2h00.

▸ Exercice 1. (Questions de cours – 4 pt.)

1. Pour chacun des éléments suivants, indiquez si on les trouve au niveau de la *déclaration* d'une fonction, de sa *définition* ou *les deux*.

   — Documentation (le plus approprié)

   Déclaration

   — Entête de la fonction

   Les deux

   — Code complet de la fonction

   Définition

2. Une fonction peut-elle être *déclarée* plusieurs fois dans un programme ?

   Oui

3. Une fonction peut-elle être *définie* plusieurs fois dans un programme ?

   Non

4. Donnez un exemple de *déclaration* de fonction (sans la définition) avec documentation

```
/**
 * Somme de deux entiers
```

PLACE :

```
 * @param a un entier
 * @param b un entier
 * @return la somme de a et b
 **/
int somme(int a, int b);
```

PLACE :

▸ **Exercice 2. (Guerrières Amazones – 6 points)** Dans cet exercice, on cherche à modéliser un jeu de rôle avec des guerrières Amazones de l'antiquité. Pour cela, nous aurons des personnages, des armes et des tribus.

1. Dans notre jeu, il existera 3 métiers de personnages : les guerrières, les sorcières et les ouvrières. Déclarer un type énuméré `MetierPerso` avec 3 valeurs possibles dans cet ordre : `Ouvriere` , `Sorciere`, `Guerriere`.

```
enum class MetierPerso { Ouvriere, Sorciere,  Guerriere};

//
```

2. Surcharger l'opérateur d'affichage `<<` pour le type `MetierPerso`.

```
ostream& operator<<(ostream &out, MetierPerso metier) {
    switch(metier) {
        case MetierPerso::Ouvriere: out << "Ouvrière";break;
        case MetierPerso::Sorciere: out << "Sorcière";break;
        case MetierPerso::Guerriere: out << "Guerrière";break;
    }
    return out;
}









//
```

3. Déclarer la structure `Arme` qui contiendra les champs (dans cet ordre) : `nom` de type `string`, le niveau d'attaque, `attaque` de type `int`, le niveau de défense, `defense` de type `int`. De telle sorte que la déclaration des variables suivantes fonctionne.

```
Arme hache = {"Hache", 5, 2};
Arme bouclier = {"Bouclier", 1, 5};
Arme sabre = {"Sabre", 7, 1};
```

```
struct Arme {
    string nom;
    int attaque;
    int defense;
};









//
```

PLACE :

4. Déclarer la structure `Personnage` qui contiendra les champs (dans cet ordre) : `nom` de type `string`, le métier du personnage `metier` de type `MetierPerso`, le niveau d'expérience du personnage `experience` de type `int`, le nombre de points de vie `ptsVie` de type int, la liste des ses armes `armes` sous forme d'un vecteur d'armes. De telle sorte que la déclaration des variables suivantes fonctionne.

```
Personnage antiope = {"Antiope", MetierPerso::Guerriere, 6, 10, {hache, bouclier} };
Personnage hippolite = {"Hippolité", MetierPerso::Guerriere, 4, 8,
{sabre, bouclier} };
Personnage ainia = {"Ainia", MetierPerso::Sorciere, 5, 10, {bouclier} };
Personnage ainippe = {"Ainippé", MetierPerso::Ouvriere, 6, 5, {} };
```

```
struct Personnage {
    string nom;
    MetierPerso metier;
    int experience;
    int ptsVie;
    vector<Arme> armes;
};









//
```

5. Écrire une fonction `nbArmes` qui prend en paramètre une référence constante sur un personnage et renvoie le nombre d'armes du personnages, de telle sorte que les tests suivants fonctionnent. (On compte toutes les armes, même s'il y a des doublons)

```
CHECK(nbArmes(antiope) == 2);
CHECK(nbArmes(hippolite) == 2);
CHECK(nbArmes(ainia) == 1);
CHECK(nbArmes(ainippe) == 0);
```

```
int nbArmes(const Personnage &perso) {
    return perso.armes.size();
}


//
```

6. Écrire une fonction `equilibreArmes` qui prend en paramètre deux références sur des personnages et qui
   - si le premier personnage a plus d'armes que le second, enlève la dernière arme du premier personnage et la donne au second (ajoutée à la suite des armes du second personnage)
   - ne fait rien sinon.

   La fonction renvoie `true` si une arme a été donnée et `false` sinon. Par exemple, comme Antiope et Hippolité ont le même nombre d'armes, la fonction `equilibreArmes(antiope, hippolite)` renvoie `false` et ne modifie rien. Mais comme Antiope a deux armes et Ainippé aucune, la fonction `equilibreArmes(antiope, ainippe)` renvoie `true`, et Antiope n'a plus que une hache tandis que Ainippé a un bouclier.

PLACE :

```
bool equilibreArmes(Personnage &p1, Personnage &p2) {
    if(nbArmes(p1) > nbArmes(p2)) {
        p2.armes.push_back(p1.armes[p1.armes.size()-1]);
        p1.armes.pop_back();
        return true;
    }
    return false;
}

// La méthode pop_back de C++ n'a pas de valeur de retour,
// elle se contente de détruire la case,
// il faut donc avoir sauvegardé la valeur avant.









//
```

7. On ajoute maintenant la structure `Tribu` ci-dessous ainsi qu'une variable pour les tests.

```
struct Tribu {
    string nom;
    vector<Personnage> membres;
};

Tribu anispians = {"The Anipians", {antiope, hippolite, ainia, ainippe} };
```

Écrire une fonction `comptePerso` qui prend en paramètre une référence constante sur une tribu ainsi qu'un métier de personnage de type `MetierPerso` et renvoie le nombre de personnages avec ce métier dans la tribu comme dans les exemples ci-dessous.

```
CHECK(comptePerso(anispians, MetierPerso::Guerriere) == 2);
CHECK(comptePerso(anispians, MetierPerso::Sorciere) == 1);
CHECK(comptePerso(anispians, MetierPerso::Ouvriere) == 1);
```

```
int comptePerso(const Tribu &tribu, MetierPerso metier) {
    int nb = 0;
    for(Personnage p: tribu.membres) {
        if(p.metier == metier) {
            nb++;
        }
    }
    return nb;
}
// Autre solution:
int comptePerso2(const Tribu &tribu, MetierPerso metier) {
    int nb = 0;
    for(int i=0; i < tribu.membres.size(); i++) {
        if(tribu.membres[i].metier == metier) {
            nb++;
        }
    }
    return nb;
}
```

PLACE :

▸ **Exercice 3. (Nombres réels avec 3 chiffres significatifs – 10 points)** On souhaite créer un type de données pour représenter des nombres réels avec 3 chiffres significatifs. Par exemple, 312, 4210000, 3.14, 2.00, 0.0301 sont tous des nombres avec trois chiffres significatifs. En revanche, 42310 et 0.04235 ont chacun quatre chiffres significatifs. Pour ceci, ces nombres seront représentés par un nombre entier nommé `mantisse` comportant exactement 3 chiffres (donc entre 100 et 999) multiplié par une puissance de dix : voici quelques exemples : $312 = 312 \times 10^0$, $4210000 = 421 \times 10^4$, $3.14 = 314 \times 10^{-2}$, $2.00 = 200 \times 10^{-2}$, $0.0301 = 301 \times 10^{-4}$. La représentation d'un tel nombre sera dite *normalisée*.

On ne représentera que des *nombres positifs*.

Enfin, le nombre 0 est une *exception*, car il ne peut pas être représenté avec une mantisse à trois chiffres significatifs. On le représentera par une mantisse et un exposant nuls. On utilisera donc les déclarations suivantes :

```
struct Reel {
  int mant;  // mantisse entre 100 et 999, ou 0
  int exp;   // exposant de la puissance de 10
};
const Reel zero = {0, 0};     // 0.00
const Reel pi = {314, -2};    // 3.14
```

1. Écrire une fonction `estNormalise` qui prend en paramètre un `Reel r` et qui renvoie `true` ou `false` selon que `r` est normalisé où non. On vous fournit des exemples sous forme de tests ci-dessous. La mantisse doit être comprise entre 100 et 999 ou être égale à 0 avec un exposant 0.

```
CHECK(estNormalise(zero));
CHECK(estNormalise({200,-3}));
CHECK_FALSE(estNormalise({1245,2}));
CHECK_FALSE(estNormalise({50, 1}));
CHECK_FALSE(estNormalise({0, 10}));
```

```
bool estNormalise(Reel r) {
  if (r.mant == 0)
    return r.exp == 0;
  return 100 <= r.mant and r.mant < 1000;
}



//
```

Dans la suite, on supposera que les nombres de type `Reel` sont normalisés.

2. Surcharger l'opérateur `<<` pour le type `Reel`. Le nombre `{0, 0}` devra être affiché par «`0`» et le nombre `{421, 4}` devra être affiché par «`421x10^4`»

```
ostream &operator<<(ostream &out, Reel r) {
  if (r.mant == 0)
    out << 0;
  else
    out << r.mant << "x10^" << r.exp;
  return out;
}



//
```

PLACE :

3. Surcharger les opérateurs `==` et `!=` pour le type `Reel`.

```
bool operator==(Reel a, Reel b) {
  return a.mant == b.mant and a.exp == b.exp;
}
bool operator!=(Reel a, Reel b) {
  return not(a == b);
}









//
```

4. Proposer 4 tests pour *chacun* des opérateurs `==` et `!=` (8 tests en tout). Les tests seront faits sur des réels normalisés et seront choisis de façon pertinente en vérifiant, par exemple, que les deux champs sont bien pris en compte par l'opérateur.

```
  CHECK(Reel{123, 2} == Reel{123, 2});
  CHECK_FALSE(Reel{123, 2} == Reel{142, 4});
  CHECK_FALSE(Reel{123, 2} == Reel{123, 4});
  CHECK_FALSE(Reel{123, 2} == Reel{142, 2});

  CHECK_FALSE(Reel{123, 2} != Reel{123, 2});
  CHECK(Reel{123, 2} != Reel{142, 4});
  CHECK(Reel{123, 2} != Reel{123, 4});
  CHECK(Reel{123, 2} != Reel{142, 2});









//
```

PLACE :

On souhaite à présent remplacer la structure `Reel` par une classe contenant :

- un constructeur par défaut qui construit le réel 0.
- un constructeur qui convertit un entier en réel normalisé.
- un constructeur qui construit un réel normalisé à partir d'une mantisse et d'un exposant.
- une méthode `normalise()` qui normalise le réel
- une méthode `valeur` qui renvoie la valeur du réel en type `double`
- des surcharges pour les opérateurs d'addition `+` et de multiplication `*` en tant que méthode

5. Écrire la *déclaration complète* de la classe. Seule la définition du constructeur par défaut sera donnée en ligne.
   Vous prendrez garde à noter les méthodes qui sont **constantes** et à passer les paramètres en tant que références constantes quand c'est utile.

```
struct Reel {
  int mant;  // mantisse entre 100 et 999, ou 0
  int exp;   // exposant de la puissance de 10

  Reel(): mant{0}, exp{0} {};
  Reel(int n);
  Reel(int m, int e);

  void normalise();
  double valeur() const;

  Reel operator+(const Reel &b) const;
  Reel operator*(const Reel &b) const;
};









//
```

PLACE :

6. Avant de transformer notre structure en classe, on avait écrit la fonction suivante qui prenait un `Reel` en paramètre et renvoyait un double.
   Transformer cette fonction pour écrire la définition de la méthode correspondante pour la classe `Reel` (Vous pouvez modifier directement sur le code ci-dessous ou écrire à côté)

```
double valeur(Reel r) {
    double d = r.mant;
    int e = r.exp;
    while(e > 0) {
        d*=10;
        e--;
    }
    while(e < 0) {
        d/=10;
        e++;
    }
    return d;
}
```

```
double Reel::valeur() const {
    double d = mant;
    int e = exp;
    while(e > 0) {
        d*=10;
        e--;
    }
    while(e < 0) {
        d/=10;
        e++;
    }
    return d;
}

//
```

7. Écrire la définition de la méthode `normalise()` qui fait en sorte que le réel ait une forme normalisée : mantisse et exposant égaux à 0 ou mantisse entre 100 et 999.
   Par exemple, si le réel a pour valeur `{31415, -4}` qui représente 3.1415 la valeur après normalisation doit être de `{314, -2}` = 3.14 (on a divisé par 100 et perdu des chiffres significatifs). De même, si le réel a pour valeur `{2, 0}`, c'est-à-dire la valeur 2, la valeur après normalisation doit être `{200, -2}` (on a multiplié par 100 et rajouté des chiffres significatifs).
   On supposera que la mantisse est toujours positive et on ne traitera pas le cas négatif.

```
void Reel::normalise() {
  if (mant == 0) {
    exp = 0;
    return;
  }
  while (mant > 999) {
    mant /= 10;
    exp += 1;
  }
  while (mant < 100) {
    mant *= 10;
    exp -= 1;
  }
}

//
```

PLACE :

8. Écrire la définition du constructeur qui prend en paramètre la mantisse et l'exposant. Le constructeur devra lever une exception `invalid_argument` si la mantisse est négative et normaliser l'objet `Reel` construit en appelant la méthode précédente.

```
Reel::Reel(int m, int e): mant{m}, exp{e} {
    if(mant < 0) {
        throw invalid_argument("Mantisse incorrecte");
    }
    normalise();
}









//
```

9. Donner une définition en 1 ligne du constructeur convertissant un entier en réel normalisé en déléguant la construction et la normalisation au constructeur précédent.

```
Reel::Reel(int n): Reel{n, 0} {};
//
```

10. Donner la définition de l'opérateur produit (en tant que méthode) dont voici des exemples sous forme de tests

```
CHECK(Reel{1,0} * Reel{314,-2} == Reel{314, -2});
CHECK(Reel{} * Reel{314, -2} == Reel{});
CHECK(Reel{111,2} * Reel{111,2} == Reel{123, 6});
CHECK(Reel{314,4} * Reel{2, -3} == Reel{628, 1});
```

```
Reel Reel::operator*(const Reel &b) const {
    return Reel{mant * b.mant, exp + b.exp};

}

// On utilise le fait que la normalisation est faite par le constructeur









//
```

PLACE :

Cette page peut être utilisée si vous n'avez pas eu la place dans les cadres prévus pour les réponses. **Attention !** Nous corrigeons en ne voyant QUE le cadre, si vous écrivez dans les espaces supplémentaires, NOTEZ LE au niveau du cadre prévu pour la réponse.

.