EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation II, sections MA et PH
Chappelier J.-C.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Programmation II (SMA/SPH) : SÉRIE NOTÉE

28 avril 2017

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre série annulée dans le cas contraire.

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (15h15 - 17h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni autre couleur ;
3. Vous avez droit à toute documentation papier.
   Par contre, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; utilisez aussi le verso des feuilles, **MAIS** n'utilisez *que* le verso de la feuille sur laquelle se trouve la question, et non **pas** celui de la feuille précédente !
   Vous pouvez, si nécessaire, joindre des feuilles supplémentaires, mais veillez dans ce cas à *n'utiliser que le papier officiel fourni par les assistants* ; **aucune feuille non officielle ne sera corrigée**.
   N'oubliez pas dans ce cas d'*indiquer votre numéro interne* (0000) sur chacune des feuilles supplémentaires, en haut à gauche sous la rubrique « Anonymisation ». Aucune feuille sans ce numéro ne sera corrigée.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte deux exercices indépendants (pages 2, 5 et 7), qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 63). Ces deux exercices comptent pour la note finale.

Question 1 – Séquences de nombres [sur 25 points]

Considérez la fonction `main()` suivante :

```
void compute_and_print(NumberSequence const& seq,
                       string const& adjective = "positive")
{
    cout << adjective << " numbers: compute() = "
         << seq.compute() << endl;
}

int main()
{
    int number = 0;
    PositiveSequence p;
    NegativeSequence n;
    do {
        cout << "Please enter a number (0 to stop): " << flush;
        cin >> number;
        n.append(number);
        p.append(number);

    } while (number != 0);

    compute_and_print(p);
    compute_and_print(n, "negative");

    return 0;
}
```

Il s'agit un programme simple qui lit en boucle un nombre entier et l'ajoute à une séquence de nombres positifs ou à une séquence de nombres négatifs puis calcule et affiche chacune des moyennes de ces deux séquences, comme par exemple dans ce déroulement ci :

```
Please enter a number (0 to stop): 10
Please enter a number (0 to stop): -2
Please enter a number (0 to stop): -1
Please enter a number (0 to stop): 5
Please enter a number (0 to stop): -3
Please enter a number (0 to stop): 8
Please enter a number (0 to stop): 25
Please enter a number (0 to stop): 0
positive numbers: compute() = 10
negative numbers: compute() = -2
```

Dans cet exercice, vous devez définir les classes permettant à ce programme de compiler et s'exécuter correctement. Les contraintes à respecter sont les suivantes :

— les nombres entrés (saisis au claviers) sont tous des entiers ; la saisie s'arrête sur l'entrée du nombre 0 (déjà fait) ;

— la méthode `PositiveSequence::append()` ajoute le nombre reçu à la séquence correspondante uniquement si ce nombre est strictement positif (sinon elle ne fait rien) ;

— la méthode `NegativeSequence::append()` ajoute le nombre reçu à la séquence correspondante uniquement si ce nombre est strictement négatif (sinon elle ne fait rien) ;

— pour déterminer si un nombre doit être ajouté à une séquence (méthode `append()`), il *faut* utiliser une méthode `is_acceptable(int)` (à écrire, bien sûr) ;

— la méthode `compute()` retourne un `double` ;

— la méthode `PositiveSequence::compute()` retourne la moyenne géométrique de la séquence (pour rappel, la moyenne géométrique est donnée par $\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$; pour calculer x^y, utiliser `pow(x, y)`) ;

si la séquence est vide, cette méthode retourne simplement 1 ;

— la méthode `NegativeSequence::compute()` retourne la moyenne arithmétique de la séquence (pour rappel, la moyenne arithmétique est donnée par $\frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)$) ;

si la séquence est vide, cette méthode retourne simplement 0.

Utilisez l'espace ci-dessous (et la page suivante) pour écrire le code manquant au programme donné au début de l'exercice.

Solution : Il y a au moins deux façons également bonnes de représenter ces séquences : soit en stockant explicitement tous les nombres dans un `vector` (solution proposée ci-dessous), soit en gardant simplement leur accumulation (somme ou produit suivant le cas) et leur nombre ; cela suffit pour ce qui est demandé ici.

(exemple de code solution au dos.)

```
class NumberSequence
{
public:
    virtual ~NumberSequence() {}
    virtual double compute() const = 0;

    void append(int number)
    {
        if (is_acceptable(number)) {
            sequence_.push_back(number);
        }
    }
protected:
    virtual bool is_acceptable(int number) const = 0;
    vector<int> sequence_;
};

class NegativeSequence : public NumberSequence
{
public:
    bool is_acceptable(int number) const override
    {
        return (number < 0);
    }
    double compute() const override
    {
        // arithmetic mean
        double sum(0.0);
        if (sequence_.size() == 0) {
            return sum;
        }
        for (auto i : sequence_)
            sum += i;
        return sum / sequence_.size();
    }
};

class PositiveSequence : public NumberSequence
{
public:
    bool is_acceptable(int number) const override
    {
        return (number > 0);
    }
    double compute() const override
    {
        // geometric mean
        double product(1.0);
        if (sequence_.size() == 0) {
            return product;
        }
        for (auto nb : sequence_) {
            product *= nb;
        }
        return pow(product, 1.0 / sequence_.size());
    }
};
```

Question 2 – Batailles de Pokemon [sur 38 points]

On cherche ici à écrire un programme pour simuler des combats entre « dresseurs » de Pokemon.

Question 2.1 Le combat de Pokemon [sur 20 points]

Premièrement, nous devons définir une classe générale pour représenter les Pokemon (classe `PokemonBase`). Pour simplifier ici, un Pokemen doit avoir :

- un nombre entier qui représente son pouvoir ;
- une méthode `battle()` retournant un `bool` et recevant un autre Pokemon avec lequel aura lieu la bataille (la bataille ne modifie aucun des deux Pokemon) ;
- un opérateur `>` utilisé par le méthode `battle` ; il compare les pouvoirs des deux combattants comme expliqué ci-dessous.
  Pour simplifier, nous ne vous demandons pas ici de coder les opérateurs `>=`, `<`, ni `<=`, bien que ceci devrait normalement se faire dans un programme usuel.

Comme vous le savez peut être, il existe plusieurs sortes de Pokemon, lesquelles ont un effet sur le pouvoir pendant la bataille. Dans cet exercice, le pouvoir est doublé ou réduit de moitié en fonction des sortes de Pokemon qui se battent. Par exemple, si un Pokemon « de l'eau » combat un Pokemon « du feu », le pouvoir de celui « de l'eau » est doublé.

Dans cet exercice, nous modéliserons cet effet avec simplement deux autres classes : `FirePokemon` et `WaterPokemon`, qui représentent deux sortes de Pokemon, et le produit modulo 7 de deux nombres, disons a et b.

Pour un `FirePokemon`, a vaut 3 et b vaut 5 et pour un `WaterPokemon` a vaut 6 et b vaut 6. Lorsque deux Pokemon se combattent, on fait simplement le produit modulo 7 du a du premier combattant avec le b du second pour trouver le coefficient à appliquer au pouvoir du premier combattant. Si le produit vaut 4, le coefficient à appliquer est 0.5.

Ainsi, si un `WaterPokemon` (premier combattant) rencontre un `FirePokemon`, son pouvoir est multiplié par 2 (car $6 \times 5 = 2$ modulo 7) ; si un `FirePokemon` (premier combattant) rencontre un `WaterPokemon`, son pouvoir est multiplie par 0.5 car $3 \times 6 = 4$ modulo 7 (et que 4 signifie 0.5 avec nos conventions) ; si un `FirePokemon` (premier combattant) rencontre un autre `FirePokemon`, son pouvoir est multiplie par 1 car $3 \times 5 = 1$ modulo 7 (idem pour deux `WaterPokemon` : $6 \times 6 = 1$ modulo 7).

Dans un combat, seul le pouvoir du premier combattant est multiplié par ce coefficient ; le pouvoir du second combattant est utilisé tel quel. Si le pouvoir du premier combattant, modifié par le coefficient, est strictement supérieur à celui du second, alors l'opérateur `>` retournera `true`.

Vous devez concevoir une solution pour que l'opérateur `>` de classe `PokemonBase` ajuste correctement le pouvoir en fonction des sortes de Pokemon qui se combattent, suivant les principes décrit ci-dessus. La méthode `battle` utilisera simplement cet opérateur `>` pour décider si le combat est gagné (`>` retourne `true`) ou perdu.

Utilisez la page ci-contre pour définir *complètement* les classes `PokemonBase`, `WaterPokemon` et `FirePokemon`.

```
class PokemonBase
{
public:
  PokemonBase(int strength, int a, int b)
      : strength_(strength), coef1(a), coef2(b)
    {}

    // returns true if stronger than opponent
    bool battle(const PokemonBase& opponent) const
    {
        return *this > opponent;
    }

    int strength() const
    {
        return strength_;
    }

    // returns true if this is stronger than p
    bool operator>(const PokemonBase& p) const
    {
        double factor(coef1 * p.coef2 % 7);
        if (factor == 4) factor = 0.5;
        return factor * strength() > p.strength();
    }

private:
    const int strength_;
    const int coef1;
    const int coef2;
};

class FirePokemon : public PokemonBase
{
public:
  FirePokemon(int strength)
      : PokemonBase(strength, 3, 5)
    {}
};

class WaterPokemon : public PokemonBase
{
public:
    WaterPokemon(int strength)
      : PokemonBase(strength, 6, 6)
    {}
};
```

Note : ceci n'est qu'*une* solution, *simple*, à base d'attributs d'instances. Il existe d'autres solutions, meilleures, à base de templates d'int, de méthode virtuelles ou d'attributs de classes. Elles sont tout aussi valides mais non attendues (sauf peut être des méthodes virtuelles) à ce niveau du cours et en temps limité.

Question 2.2 Les dresseurs de Pokemon [sur 18 points]

En second lieu, nous devons définir une classe pour représenter les dresseurs de Pokemon (classe `Trainer`). Un dresseur de Pokemon doit avoir :

- une collection d'au maximum six Pokemon ;
- une méthode `add_pokemon()` pour ajouter un Pokemon à sa collection ; si le maximum de six Pokemon est déjà atteint, cette méthode retire/supprime le premier Pokemon (celui le plus ancien, à la position 0) avant d'ajouter le nouveau ; cette méthode ne retourne rien ;
- une méthode `remove_pokemon()` qui reçoit un indice en argument et retire/supprime le Pokemon stocké à cet indice/cette position ; tous les Pokemon doivent restés stockés de façon contiguë sans en changer l'ordre : par exemple, si l'on retire le Pokemon d'indice 3 et qu'il y avait cinq Pokemon en tout, alors celui d'indice 4 se retrouvera en position 3, et celui d'indice 5 au départ se retrouvera en position 4 ;
  cette méthode ne retourne rien ;
- une méthode `battle()` qui reçoit un autre dresseur (`Trainer`) et simule une bataille entre les deux : les Pokemon des deux dresseurs se battent dans l'ordre de leur stockage, avec le principe qu'un vainqueur continue à battre jusqu'à propre sa défaite ou jusqu'à ce que l'un des deux dresseurs soit vaincu (c.-à-d. que ses Pokemon aient tous perdus). Cette méthode `battle` ne modifié aucun des deux dresseurs (on ne supprime pas les Pokemon perdants). Elle retourne `true` si l'instance courante a gagné (c.-à-d. que le nombre de ses Pokemon perdant est strictement inférieur au nombre total de Pokemon qu'il possède dans sa collection) et `false` sinon. On suppose pour simplifier ici qu'il ne peut pas y avoir de match nul.

Un exemple de `main()` est donné ici :

```
void battle(Trainer const& t1, Trainer const& t2) {
    cout << "Trainer ";
    if (t1.battle(t2)) {
      cout << '1';
    } else {
      cout << '2';
    }
    cout << " has won." << endl;
}

int main()
{
    Trainer t1, t2;
    t1.add_pokemon(new WaterPokemon(8));
    t2.add_pokemon(new FirePokemon(10));
    battle(t1, t2);
    t2.remove_pokemon(0);
    t2.add_pokemon(new WaterPokemon(10));
    battle(t1, t2);
    return 0;
}
```

Celui-ci produirait la sortie :

```
Trainer 1 has won.
Trainer 2 has won.
```

```
class Trainer
{
public:
    void add_pokemon(PokemonBase* p)
    // p must be prealloc'd to make use of base pointer
    // takes ownership of p
    {
        while (pokemon_.size() >= max_) {
            remove_pokemon(0); // pop_front(), but they don't know about dequeue yet...
        }
        pokemon_.push_back(unique_ptr<PokemonBase>(p));
    }

    // remove at index
    void remove_pokemon(const size_t index)
    {
        if (index < max_ && pokemon_.size() > index) {
            // they don't know erase() yet: pokemon_.erase(pokemon_.begin() + index);
            const size_t before_last(pokemon_.size() - 1);
            for (size_t i(index); i < before_last; ++i) {
                swap(pokemon_[i], pokemon_[i+1]); // cannot use assignment with unique_ptr
            }
            pokemon_.pop_back();
        }
    }

    // battle does not remove
    bool battle(const Trainer& opponent) const
    {
        size_t me = 0, op = 0; // basically loss counters
        while (me < pokemon_.size() && op < opponent.pokemon_.size()) {
            if (pokemon_[me]->battle(*opponent.pokemon_[op])) {
                op++;
            } else {
                me++;
            }
        }
        return (me != pokemon_.size());
    }

private:
    static const size_t max_ = 6;
    vector<unique_ptr<PokemonBase>> pokemon_;
};
```