Programmation orientée objet en C++

Travaux pratiques séance 2

A la fin du TP, déposez une archive avec votre travail (code source) sur ecampus, dans le repertoire correspondant à votre groupe. Avant minuit, déposez également sur ecampus le compte-rendu détaillé (CR) correspondant à votre travail ; les modifications dans le code ultérieures à la fin du TP ne sont pas prises en compte - concentrez vous sur la rédaction du CR.
Conseils de base :

- le TP est **individuel** ; toute fraude sera sanctionnée drastiquement.
- dès que possible (toutes les quelques lignes idéalement), **compilez** et **testez** votre code. L'erreur la plus fréquente des débutants est d'écrire des dizaines de lignes de code sans tester, et de se retrouver tout à coup avec plein d'erreurs de compilation et de bugs
- si vous dupliquez du code dans vos méthodes, cela veut dire qu'il y a probablement un problème : soit une méthode devrait appeler l'autre, soit le code commun devrait être mis dans une méthode auxiliaire, qui est appelée chaque fois que sa fonctionnalité est nécessaire
- même si cela ne vous rapporte pas une meilleure note tout de suite, essayez d'aller au bout de ce sujet avant le TP suivant. Cela vous améliorera sûrement votre niveau, ainsi que les notes suivantes.

Des éléments pris en compte lors de la notation :

- vous avez respecté ce qu'on demande (la date limite pour l'envoi de votre code et compte-rendu, le format de la soumission etc.)
- votre code contient des **commentaires** qui justifient bien le rôle de chaque classe, membre de classe, méthode, et qui expliquent les détails moins évidents d'implémentation
- le compte-rendu est un document soigné et bien organisé (introduction, rappel bref de l'objectif, contenu, discussion finale) qui justifie tous vos **choix** de design des classes (i.e. pour quoi ce membre est initialisé par défaut ainsi, pour quoi l'allocation de ce tableau se fait dans cette méthode et pas dans le constructeur etc.), mais on ne met ni les détails d'implémentation (pour cela on a les commentaires), ni des copier-coller des classes et des méthodes entières pour faire du volume
- la syntaxe de votre programme est **correcte** (i.e. le code compile)
- votre programme fonctionne correctement (i.e. le résultat correspond à ce qui est demandé), les tests demandés sont implémentés
- votre code est de bonne qualité (i.e. pas de fuites de mémoire, listes d'initialisation pour les constructeurs, encapsulation efficace, utilisation correcte des attributs vus en cours (public, private, const, static etc.) bonne organisation du code en méthodes)

1 Calculs avec des entiers de très grandes dimensions

Le but de ce TP est de réaliser une classe pour la manipulation de nombres entiers de dimensions arbitraires. Cette classe nous permettra de résoudre le problème 48 proposé par le Projet Euler, accessible à `https://projecteuler.net/` Ce site vous permet de vérifier en ligne la réponse au problème proposé ; afin d'accéder à cette fonctionnalité, créez vous tout d'abord un compte (gratuit) et regardez l'énoncé du problème 48. L'objectif est de calculer la somme de la série :

$$S(1000) = 1^1 + 2^2 + 3^3 + \ldots + 1000^{1000}$$

et d'envoyer pour vérification les 10 derniers chiffres du résultat.

Exercice 1 — Limitation des types natifs

Dans l'énoncé du problème, on nous indique que

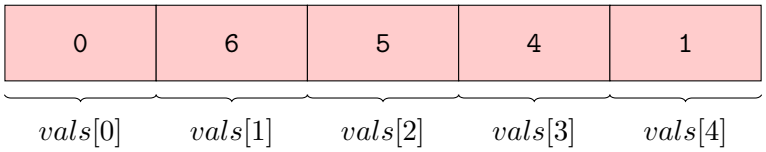
$$S(10) = 1^1 + 2^2 + 3^3 + \ldots + 10^{10} = 10405071317$$

. On essayera d'abord de vérifier cela en utilisant deux types entiers natifs, `int` et `long long int`.

1. Écrire une fonction libre `int puissanceInt(int n)` qui renvoie en résultat la valeur de n^n, et utilisez la dans une autre fonction libre `int serieInt(int n)` qui calcule $S(n)$. Dans une fonction de test `void test_types_natifs()` affichez les valeurs obtenues avec `int serieInt(int n)` pour $n = 1 \ldots 12$.
2. Faire la même chose, mais à travers une fonction libre `long long int puissanceLong(int n)` et une fonction `long long int serieLong(int n)`, qui utilisent cette fois des variables de type `long long int`.
3. Comparez les valeurs obtenues dans les deux cas ; quelles valeurs sont erronées et à partir de quel n ? Pour quoi ? Est-ce que le type `int` nous permet d'encoder la valeur de 10^{10} ? Est-ce que le type `long long int` nous permettrait d'encoder la valeur de 1000^{1000} ? Essayer de justifier votre réponse en comparant le nombre de bits disponibles dans la représentation de ces types avec le nombre de bits nécessaires pour représenter les valeurs souhaitées.

Exercice 2 — Entiers de dimension arbitraire

Pour contourner les limitations des types natifs, on écrit notre propre classe d'entiers, qu'on appelle `BigInt`. L'astuce est de ne pas sauvegarder la représentation d'un nombre entier très grand dans un type natif, mais dans une structure de données qui peut encoder autant de bits ou de chiffres qu'on veut. Pour ce TP, on choisit une représentation qui n'est pas très compacte, mais qui est facile à manipuler : un entier est encodé dans un tableau de `char`, ou les éléments du tableau encodent en ordre (unités, dizaines, centaines etc.) les chiffres de l'entier. Par exemple, le nombre 14560 contenant 5 chiffres serait représenté par le tableau `vals` suivant :



1. Quelle est le nombre de chiffres `N` nécessaire pour représenter 1000^{1000} ? Pour représenter $S(1000)$?
2. Écrire, **à partir de la classe modèle fournie en `fcc.h` et `fcc.cpp`**, une classe `BigInt` qui a
   — un champ privé `char vals[N]` qui sert a stocker l'entier
   — un champ privé `int numDigits` qui indique combien de chiffres contient l'entier
   — un constructeur `BigInt()` qui remplit `vals` avec des zéros et qui initialise `numDigits` à 0 ; pour remplir une zone de mémoire avec la même valeur, utilisez `memset` (et lisez bien la doc)
   — un constructeur `BigInt(int n)` qui initialise un objet de type `BigInt` pour représenter l'entier **positif** n. En dehors des valeurs en `vals` utilisées pour représenter n, le reste doit être mis à 0.
   — un constructeur de copie (utiliser `memcpy` après avoir lu la doc)
   — l'opérateur `=` surchargé (utiliser `memcpy`)
   — l'opérateur de comparaison `==` surchargé
   — l'opérateur `<<` surchargé, qui vous permettra de vérifier que tous ces constructeurs/méthodes fonctionnent correctement
   Dans la fonction libre `void test_constr()`, testez les différents constructeurs et l'opérateur `=`, à l'aide de l'opérateur `<<`.
3. Surchargez l'opérateur d'addition `+`, pour faire la somme de deux `BigInt`. On vous suggère l'algorithme suivant : faire la somme des chiffres correspondants jusqu'à ce que vous arrivez à la fin du

`BigInt` le plus long parmi les deux. Cela est correct car en dehors de la représentation du `BigInt` plus court, les cases contiennent des zéros. N'oubliez pas la retenue que vous devez reporter chaque fois pour l'addition des cases suivantes.

> **Important** : veillez à écrire un code lisible, bien indenté et avec des commentaires utiles. La mission des chargés de TP est surtout de vous aider avec la syntaxe du langage et avec les bonnes pratiques, ils ne sont pas là pour déboguer vos algorithmes, surtout si votre programme est illisible. Habituez vous à écrire du code que vous mêmes pouvez lire et déboguer facilement.

Dans la fonction libre `void test_addition()`, déclarez les variables `BigInt` b1, b2, b3 et initialisez les aux valeurs (100,1,101) et vérifiez `assert (b1 + b2 == b3)` et `assert (b2 + b1 == b3)`. Pareil pour les triplets (100,0,100) et (99,1,100).

Exercice 3 — Multiplication de `BigInt`

La multiplication nous intéresse également si on veut calculer rapidement les puissances demandées dans le problème, mais regardons plus en détail comment cela se passe. Par exemple, si on veut multiplier 2345 (b1) et 143 (b2), on le fait à la main de la manière suivante :

```
    2345 x
     143
--------
    7035
   9380
  2345
--------
  335335
```

On peut décrire ce processus ainsi : on multiplie 2345 (b1) avec le premier chiffre (3) de b2, et on met le résultat (7035) dans une variable intermédiaire (`temp`). On multiplie 2345 avec le deuxième chiffre (4) de b2 et on déplace toutes les chiffres du résultat (9380) d'une case, pour obtenir 93800, qu'on ajoute à `temp`. Enfin on multiplie 2345 avec le troisième et dernier chiffre de b2, on déplace toutes les chiffres du résultat (2345) de deux cases, pour obtenir 234500, qu'on ajoute également à `temp`. A la fin, `temp` contient la valeur recherchée.

1. Écrire une méthode privée `BigInt multDigit(char digit) const` de la classe `BigInt`, qui renvoie un nouvel objet `BigInt` qui est le résultat de la multiplication d'un `BigInt` par un chiffre.
2. Écrire une méthode privée `void shiftNpos(int n)` de la classe `BigInt`, qui déplace les chiffres de l'objet courant de n positions (par exemple 1234 déplacé de 3 positions devient 1234000).
3. A l'aide de ces deux méthodes auxiliaires, surcharger l'opérateur de multiplication `*` de la classe `BigInt`.
4. Proposer dans la fonction libre `void test_mult()` quelques tests pour montrer que l'opérateur de multiplication fonctionne correctement.

Exercice 4 — Résolution du problème 48

1. Écrire une fonction libre `BigInt puissanceBigInt(int n)` qui renvoie en résultat la valeur de n^n
2. Utiliser la fonction précédente dans une autre fonction libre `BigInt serieBigInt(int n)` qui calcule $S(n)$. Vérifiez dans une première étape que $S(10)$ correspond à la valeur indiquée en début du sujet.
3. Si on calcule et affiche $S(1000)$, cela ferait beaucoup de caractères à l'écran, ce qui n'est pas très élégant. Utilisez le flag `std::setw` pour choisir le nombre de chiffres que vous voulez afficher à partir du chiffre des unités, et modifier votre opérateur surchargé `<<` pour en tenir compte. Si vous ne

spécifiez pas `std::setw`, il faut afficher tous les chiffres, et si vous spécifiez `std::setw(n)` il faut en afficher juste n. Utiliser cela pour afficher juste les dernières 10 chiffres de $S(1000)$, et vérifiez votre solution sur le site du projet Euler.