université PARIS-SACLAY

Travaux Pratiques de programmation n°7

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Classes et Constructeurs

Dans cette séance de travaux pratiques, nous allons apprendre à contrôler la construction d'un objet.

▸ **Exercice 1. (Nombres entiers en précision arbitraire)** Il existe plusieurs formats de stockage des entiers en `C++`, notamment `int` et `long int` (mais aussi `short`, `unsigned short`, `unsigned int` et `unsigned long int`). Avec `int`, celui que nous avons le plus utilisé, on ne peut écrire des nombres que jusqu'à $2^{31} - 1 = 2147483647$. Si l'on incrémente ce nombre, il se produit un dépassement de capacité, le résultat est alors le nombre *négatif* -2147483648. Vous pouvez le tester avec le code du fichier `depcap.cpp` fourni :

```
int n = 2147483647;
cout << n+1 << endl;
```

Ce comportement peut nous bloquer lors d'opérations avec de très grands nombres. Ce problème peut être mitigé avec un `long int`, par exemple, qui a une taille en mémoire plus grande qu'un `int`, mais les `long int` sont aussi limités à $2^{63} - 1 = 9223372036854775807$.

En `C++`, on utilise habituellement la bibliothèque `GnuMultiPrecision` pour résoudre ce problème (voir `https://gmplib.org/` et `https://gmplib.org/manual/C_002b_002b-Class-Interface` pour les classes `C++`).

Dans cet exercice, nous allons écrire le code par nous mêmes en créant une classe pour modéliser et stocker des nombres entiers naturels sans limitation de précision. On rappelle que l'écriture 2375 en base dix signifie

$$2375 = 5 + 7 \times 10 + 3 \times 100 + 2 \times 1000 = 5 \times 10^0 + 7 \times 10^1 + 3 \times 10^2 + 2 \times 10^3 \,.$$

Le chiffre correspondant à la puissance i de 10 est appelé *chiffre de poids* i. Par exemple dans 2375, le chiffre de poids 0 est 5 et celui de poids 2 est 3.

Le format de stockage va être le suivant :

— les chiffres du nombre sont stockés dans un vecteur de `int` nommé `chiffres`.

— le chiffre de poids i est stocké dans la case i du vecteur. Ainsi pour l'affichage du nombre on commence par la **fin** du vecteur.

— Invariant : le chiffre de plus grand poids doit toujours être différent de zéro.

Ainsi, 456 est codé par le vecteur `{6,5,4}`, et 789065 est codé `{5,6,0,9,8,7}`.

Attention : une conséquence du choix précédent est que le nombre 0 est codé par le vecteur vide et non par le vecteur `{0}`.

Pour cet exercice, vous modifierez le fichier `entier.cpp`.

1. Déclarez et définissez un constructeur à partir d'un vecteur d'entiers donné (pour la classe `Naturel` fournie). Si l'une des entrées du vecteur ne contient pas un chiffre valide (entre 0 et 9), on signalera une exception `invalid_argument`. Pour le moment, on ne s'occupe pas des 0 qu'il pourrait y avoir à la fin du vecteur, ce sera le sujet de la question suivante.
2. Déclarez et définissez une méthode `normalise` qui supprime les 0 à la fin du vecteur de chiffres du `Naturel`, de manière à respecter l'invariant. Attention ! Il ne faut supprimer que les 0 à la fin. On pourra utiliser la méthode `pop_back` des vecteurs. Par exemple, 0420 s'écrira `{0,2,4,0}` et sera normalisé en `{0,2,4}`.
3. Appeler cette méthode à la fin du constructeur, pour que le `Naturel` construit soit normalisé.
4. Complétez les tests fournis afin de tester soigneusement le constructeur, en vérifiant bien tous les cas d'erreur et la suppression des 0.
5. Écrire le constructeur par défaut qui construit le nombre 0.
6. Écrire une méthode `nb_chiffres` qui retourne le nombre de chiffres. Tester cette méthode.
7. Écrire une méthode `ieme_chiffre` qui retourne le chiffre de poids i. Note : le chiffre de poids 5 de 421 est 0. Tester cette méthode.
8. Surchargez l'opérateur << pour la classe `Naturel`.
9. Écrivez un constructeur qui prend un `int` habituel. Testez.
10. Surchargez les opérateurs == et != pour comparer deux entiers. Testez ces fonctions.

▸ Exercice 2. (Nombres entiers en précision arbitraire (suite))

Cet exercice est prévu pour ceux qui vont vite. S'il vous reste moins d'une heure, passez à l'exercice suivant.

1. Surchargez les opérateurs <, >, <= et >= pour comparer deux Naturels. Vous pourrez vous aider de fonctions annexes, `compare` pour comparer deux int (qui pourra vous retourner différentes valeurs selon que vos deux arguments sont supérieurs l'un à l'autre ou égaux), que vous pourrez surcharger pour comparer deux Naturels. Testez ces fonctions.
2. Surchargez l'opérateur +. Testez cette fonction.
3. Surchargez l'opérateur -=. Testez cette fonction.
4. Surchargez l'opérateur -. Testez cette fonction.
5. Pour vérifier "l'efficacité" de notre méthode, nous allons regarder quelle est la plus grande valeur qu'un long int peut prendre en comparaison avec nos objets de la classe Naturel. En effet, les long int ont une taille limitée en mémoire, et ne peuvent pas avoir une valeur trop grande. Écrivez une boucle qui en commençant à 1, à chaque étape, double le long int et le naturel et les affiche. Observez le résultat.

La bibliothèque GMP fonctionne selon le même principe, sauf que c'est un gros gaspillage de place mémoire et de temps que de calculer en base 10 en ne stockant qu'un seul chiffre dans chaque case du vecteur. On pourrait faire mieux en stockant par exemples 3 chiffres, c'est à dire en calculant en base 1000. Pour optimiser au maximum, GMP calcule en base 2^{64}.

▸ Exercice 3. (Les tours de Hanoi)

Le jeu des tours de Hanoi (voir `https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF`) est un puzzle de réflexion dans lequel on déplace des disques de diamètres différents sur trois piquets (les tours). On a une tour de départ sur laquelle tous nos disques seront installés dans l'ordre décroissant de bas en haut (les plus grands en bas les plus petits en haut). On a une tour d'arrivée sur laquelle on veut reproduire la pyramide de disques de la tour de départ.

Pour bouger les disques, il y a deux règles :

— On ne peut déplacer qu'un disque à la fois ;

— On ne peut placer un disque que sur un disque plus grand ou un emplacement vide.

Dans cet exercice, nous allons modéliser les tours de Hanoi, mais nous n'allons pas modéliser leur résolution.

Vous modifierez le fichier `hanoi.cpp`. On vous y a fourni une surcharge de l'opérateur << pour les tableaux et pour les vecteurs.

1. Écrivez une classe Hanoi. Elle aura pour attributs :
   — le nombre de disques
   — un vecteur de la position des disques (la case d'indice i contient l'indice du piquet sur lequel se trouve le disque i) ; Par exemple `{0,1,1,2}` signifie que le disque zéro est sur le piquet 0, les disques 1 et 2 sur le piquet 1 et le disque 3 sur le piquet 2).
   — un vecteur de la position "objectif" des disques (exemple : si on veut que les quatre disques soient sur le piquet 1 : `{1,1,1,1}`)
   — le nombre de mouvements depuis le début.
2. Écrivez le constructeur de la classe Hanoi qui prend en paramètre le nombre de disques avec la syntaxe `Hanoi::Hanoi(int nb)`, et qui construit l'état initial du jeu.
3. Écrivez une méthode `reinitialise` permettant de réinitialiser le "plateau" (un élément de la classe Hanoi).
4. Écrivez une méthode `piquets_statut` permettant d'obtenir le tableau de vecteurs trié des disques sur chaque piquet (par exemple pour la situation : piquet 0 : 5 4 3, piquet 1 : 2, piquet 2 : 1 0, la méthode retournera le tableaux de vecteur `{{3,4,5},{2},{0,1}}`).
5. Écrivez une méthode `void Hanoi::bouge_disque(int deb, int fin)` pour bouger un disque. Cette méthode prendra en paramètres le piquet de départ et le piquet d'arrivée. Bien sûr, on signalera une exception si le mouvement n'est pas valide.
6. Testez cette fonction ainsi que la fonction `reinitialise`.

7. Écrivez une méthode `gagne` permettant de savoir si l'état du plateau est l'état "objectif", donc si on a bien gagné.
8. Écrivez le programme principal avec une boucle qui affiche l'état du jeu et demande à l'utilisateur deux entiers décrivant les piquets de début et fin du mouvement qu'il souhaite effectuer.
9. Vous pouvez maintenant essayer de résoudre le casse-tête !