TP Développement n°2

© 2012 tv <tvaira@free.fr> - v.1.0

Sommaire

Manipulations **2**

Structure d'un programme informatique simple 2

Objets, types et valeurs 2

Entrée 4

Opérations et opérateurs 5

Règles de codage 7

Affectation et initialisation 8

Questions de révision **11**

Travail demandé **11**

Exercice 1 : calcul de notes 11

Exercice 2 : pair ou impair 12

Exercice 3 : conversion 12

Exercice 4 : calculatrice 12

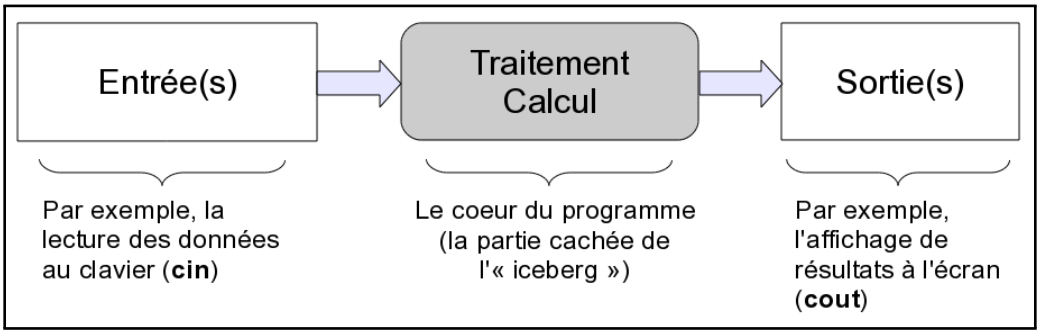
Bilan **13**

Les objectifs de ce tp sont de présenter les bases du stockage et l'utilisation des données dans un programme. Beaucoup de conseils et exemples sont issus du livre de référence de Bjarne Stroustrup (www.programmation.stroustrup.pearson.fr).

Manipulations

Structure d'un programme informatique simple

Un programme informatique est souvent structuré de la manière suivante :



Les vrais programmes ont donc tendance à produire des résultats en fonction de l'entrée qu'on leur fournit. Pour pouvoir lire quelque chose, il faut dire où le placer ensuite. Autrement dit, il faut un "endroit" dans la mémoire de l'ordinateur où placer les données lues. On appelle cet "endroit" un **objet**.

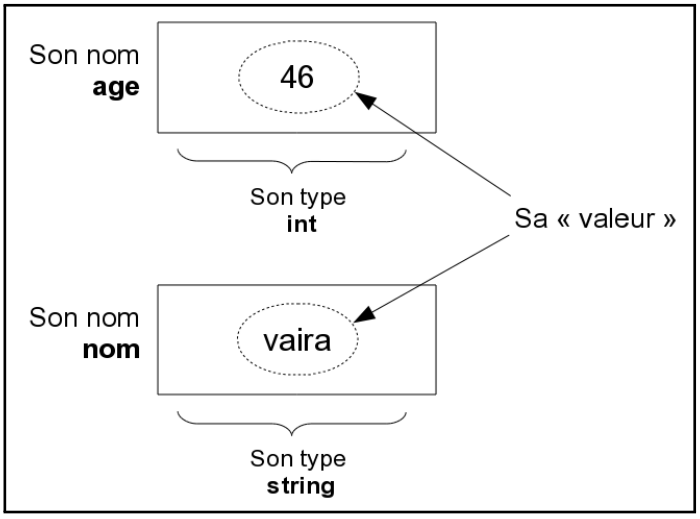
Objets, types et valeurs

Un objet est une région de la mémoire, dotée d'un **type** qui spécifie quelle sorte d'information on peut y placer. Un objet nommé s'appelle une **variable**.

Exemples :

- les **chaînes de caractères** sont stockées dans des variables de type `string`
- les **entiers** sont stockés dans des variables de type `int`
- les **réels** sont stockés dans des variables de type `float`
- etc ..

Vous pouvez vous représenter un objet comme "une boite" (une case) dans laquelle vous pouvez mettre une **valeur** du **type** de l'objet :



Représentation d'objets

```
// Réalise la saisie de l'age et du nom de l'utilisateur

#include <iostream>

int main ()
{
   int age; // Réserve une région de la mémoire destinée à contenir un entier et lui
      attribue le nom age
   string nom; // Réserve une région de la mémoire destinée à contenir une chaîne de
      caractères et lui attribue le nom nom

   std::cout << "Donnez votre âge : " << std::endl; // Affiche un message d'invite demandant
       à l'utilisateur d'exécuter une action
   std::cin >> age; // Lit la valeur saisie et la place dans age
   std::cout << "Donnez votre nom : " << std::endl;
   std::cin >> nom; // Lit les caractères et les place dans nom

   // ...

   return 0;
}
```

Saisies clavier avec cin

Remarque : il est fondamentalement impossible de faire quoi que ce soit avec un ordinateur sans stocker des données en mémoire (on parle ici de la RAM).

Une instruction qui définit une variable est ... une **définition** !

Une définition peut (et généralement doit) fournir une **valeur initiale**. Trop de programmes informatiques ont connu des *bugs* dûs à des oublis d'initialisation de variables. On vous obligera donc à le faire systématiquement. On appelle cela "respecter une régle de codage". Il en existe beaucoup d'autres.

```
int nombreDeTours = 100; // Correct 100 est une valeur entière
string prenom = "Robert"; // Correct "Robert" est une chaîne de caractères

// Mais :
int nombreDeTours = "Robert"; // Erreur : "Robert" n'est pas une valeur entière
string prenom = 100; // Erreur : 100 n'est pas une chaîne de caractères (il manque les
    guillemets)
```

Initialisation de variables

Remarque : Le compilateur se souvient du type de chaque variable et s'assure que vous l'utilisez comme il est spécifié dans sa définition.

 © 2012 tv <tvaira@free.fr>

Le C++ dispose de nombreux types (voir le cours). Toutefois, vous pouvez écrire la plupart des programmes en n'en utilisant que cinq :

```
int nombreDeTours = 100; // int pour les entiers
double tempsDeVol = 3.5; // double pour les nombres en virgule flottante (double précision)
string prenom = "Robert"; // string pour les chaînes de caractères
char pointDecimal = '.'; // char pour les caractères individuels
bool ouvert = true; // bool pour les variables logiques (booléenes)
```

Les types usuels en C++

Remarque : les types `string` *et* `bool` *n'existent pas en langage C.*

Entrée

Rappel : Une instruction qui introduit un nouveau nom dans un programme et réserve de la mémoire pour une variable s'appelle une ***définition****.*

Le nom `cin` (***c****haracter* ***in****put stream*) désigne le **flux d'entrée standard** (le clavier par défaut), dans la bibiothèque standard. L'opérateur `>>` (qui signifie "obtenir depuis") spécifie où va l'entrée. On remarque dans cet exemple que l'opérateur `>>` est sensible au type : autrement dit, la lecture dépend du type de la variable de destination. C'est la même chose pour l'opérateur `<<`.

```
// Réalise la saisie de l'age et du nom de l'utilisateur

#include <iostream>

int main ()
{
   int age; // Réserve une région de la mémoire destinée à contenir un entier et lui
      attribue le nom age
   string nom; // Réserve une région de la mémoire destinée à contenir une chaîne de
      caractères et lui attribue le nom nom

   std::cout << "Donnez votre nom et votre âge : " << std::endl; // Affiche un message d'
      invite demandant à l'utilisateur d'exécuter une action
   std::cin >> nom; // Lit les caractères et les place dans nom
   std::cin >> age; // Lit la valeur saisie et la place dans age

   // ...

   return 0;
}
```

Saisies clavier avec cin

Pourquoi `cin` ne place-t-il pas tout ce qui est saisi dans la variable `nom` ? Parce que, par convention, la lecture des chaînes de caractère se termine sur ce qu'on appelle un espace blanc (*whitespace*), c'est-à-dire le caractère espace, une tabulation ou un caractère de retour à la ligne. Notez que les espaces sont ignorés par défaut.

Question 1. Inversez la saisie (`cin`) de `nom` et `age`. Que constatez-vous ? Expliquez.

Question 2. Réaliser un programme qui saisit et affiche le nom et le prénom de l'utilisateur dans le message "Bienvenue non prénom !".

 © 2012 tv <tvaira@free.fr>

Opérations et opérateurs

Outre le fait qu'il spécifie quelles valeurs on peut stocker dans une variable, le **type** de celle-ci détermine quelles **opérations** on peut lui appliquer et ce qu'elles signifient.

Remarque : la liste des opérateurs est fournie dans le cours.

```
// Réalise des opérations sur l'age et le nom de l'utilisateur

#include <iostream>

using namespace std;

int main ()
{
   int age;
   string nom;

   cin >> nom; // Lit les caractères et les place dans nom
   cin >> age; // Lit la valeur saisie et la place dans age

   string fils = nom + " Junior"; // Ajoute (concatène) des caractères à la fin
   int ageApres = age + 1; // Additionne des entiers

   string s = nom - " Junior"; // Erreur : cet opérateur n'est pas défini pour string !
   int ageAvant = age - 1; // Soustrait des entiers

   return 0;
}
```

Opérations sur les types

Par "Erreur", nous entendons que le compilateur rejettera la ligne où on essaye de soustraire des chaînes car le compilateur sait exactement quelles opérations peuvent être appliquées à chaque type de variable.

```
erreur: no match for 'operator-' in 'nom - " Junior"'
```

Par contre, le compilateur acceptera sans broncher des opérations légales qui générèrent des résultats absurdes :

```
int age = -100;
```

Remarque : L'impossibilité d'avoir un âge négatif peut vous sembler évidente (n'est-ce pas ?), mais personne ne l'a dit au compilateur : il produira donc du code pour cette définition et vous aurez (peut-être) un bug à gérer par la suite.

La comparaison de chaînes est particulièrement utile et permet de montrer un **traitement** réalisable par un ordinateur :

```
// Lit et compare deux noms

#include <iostream>

using namespace std;

int main()
```

 © 2012 tv <tvaira@free.fr>

```
{
    cout << "Donnez deux noms :\n";
    string premier;
    string second;
    cin >> premier >> second; // Lit deux chaînes

    if (premier == second) cout << "Les deux noms sont identiques.\n";
    if (premier < second)
        cout << premier << " est avant " << second <<'\n';
    if (premier > second)
        cout << premier << " est après " << second <<'\n';

    return 0;
}
```

Comparaison de chaînes

Remarque : Nous utilisons ici une instruction ***if*** *pour sélectionner des actions soumises à des* ***conditions****. Vous pouvez consulter le cours pour obtenir plus d'informations sur les instructions conditionnelles.*

On obtient ceci à l'exécution :

```
$ ./a.out
Donnez deux noms :
titi toto
titi est avant toto

$ ./a.out
Donnez deux noms :
tata tata
Les deux noms sont identiques.

$ ./a.out
Donnez deux noms :
toto Toto
toto est après Toto
```

Question 3. Quelle est alors la définition d'"identique" dans le programme précédent ?

Question 4. Réaliser un programme qui réalise la même chose mais pour deux âges.

 © 2012 tv <tvaira@free.fr>

Règles de codage

Un nom de variable est un nom principal (surtout pas un verbe) suffisamment éloquent, éventuellement complété par :
– une caractéristique d'organisation ou d'usage
– un qualificatif ou d'autres noms

On utilisera la convention suivante : **un nom de variable commence par une lettre minuscule puis les différents mots sont repérés en mettant en majuscule la première lettre d'un nouveau mot**.

Certaines abréviations sont admises quand elles sont d'usage courant : `nbre`, `max`, `min`, ...

Les lettres `i`, `j`, `k` utilisées seules sont usuellement admises pour les indices de boucles.

Exemples : `distance`, `distanceMax`, `consigneCourante`, `etatBoutonGaucheSouris`, `nbreDEssais`, ...

Un nom de variable doit être uniquement composé de lettres, de chiffres et de "souligné" (`_`). Les noms débutant par le caractère "souligné" (`_`) sont réservés au système, et à la bibliothèque C. Les noms débutant par un double "souligné" (`__`) sont réservés aux constantes symboliques (`#define` ...) privées dans les fichiers d'en-tête (`.h`).

Il est déconseillé de différencier deux identificateurs uniquement par le type de lettre (minuscule/majuscule). Les identificateurs doivent se distinguer par au moins deux caractères, parmi les 12 premiers, car pour la plupart des compilateurs seuls les 12 premiers symboles d'un nom sont discriminants.

Les mots clés du langage sont interdits comme noms.

Remarque : l'objectif de respecter des règles de codage est d'augmenter la lisibilité des programmes en se rapprochant le plus possible d'expressions en langage naturel.

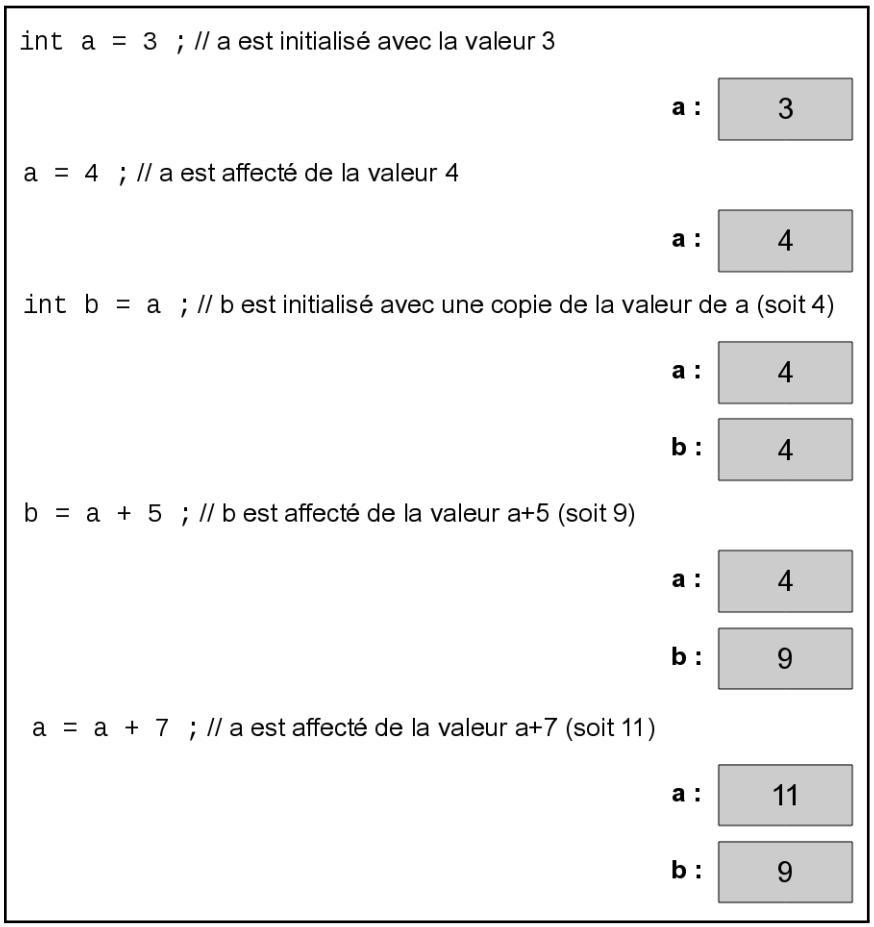
 © 2012 tv <tvaira@free.fr>

Affectation et initialisation

L'opérateur le plus intéressant (et le plus important à maîtriser) est l'**opérateur d'affectation**, représenté par `=`. Il attribue une nouvelle valeur à une variable.

L'opérateur `=` permet deux opérations similaires mais intellectuellement distinctes :

- l'**initialisation** qui donne une valeur initiale à une variable
- l'**affectation** qui donne une nouvelle valeur à une variable

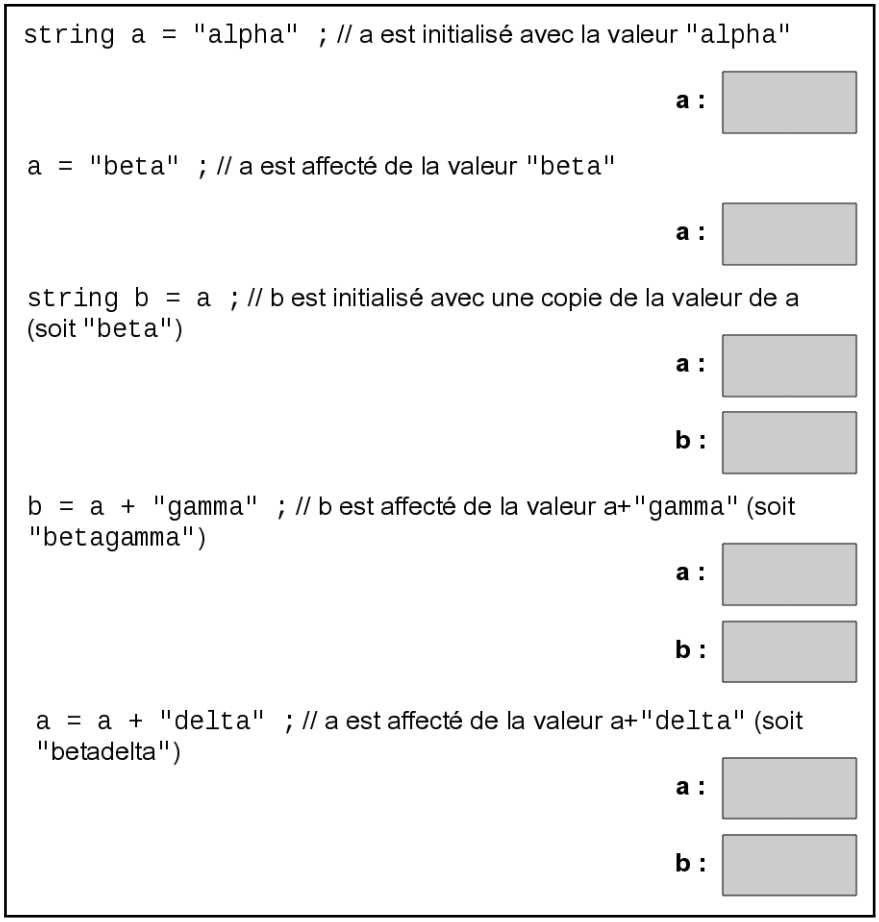


Exemple d'initialisations et d'affectations sur des entiers

Question 5. Réaliser un programme qui vérifie par des affichages successifs (`cout`) l'exemple ci-dessus.

 © 2012 tv <tvaira@free.fr>

Question 6. Compléter en indiquant les valeurs de `a` et `b` pour l'exemple ci-dessous.



Exemple d'initialisations et d'affectations sur des chaînes

L'affectation est nécessaire lorsqu'on veut placer une nouvelle valeur dans un objet. Elle trouve toute son utilité dès qu'on fera quelque chose plusieurs fois car il faudra une affectation pour refaire quelque chose avec une valeur différente (c'est le concept de variable).

Voici un programme qui illustre cela : un détecteur de mots répétés adjacents dans une suite de mots. Ce genre de code fait partie de la plupart des verificateurs de grammaire :

```
// Détecte des mots répétés adjacents
#include <iostream>
using namespace std;

int main()
{
    string motPrecedent = " "; // signifiera "pas un mot" !
    string motCourant;
    while (cin >> motCourant) // Lit un flux de mots
    {
        if (motPrecedent == motCourant) // Teste si le mot est le même que le précédent
            cout << "mot répété : " << motCourant << '\n';
        motPrecedent = motCourant;
    }
    return 0;
}
```

Détecteur de mots répétés

 © 2012 tv <tvaira@free.fr>

On lit donc un mot dans `motCourant` avec `cin`, puis on le compare au mot précédent (`motPrecedent`). S'ils sont "identiques", on l'affiche à l'écran. Puis on doit se préparer pour le mot suivant : on place donc dans `motPrecedent` le mot contenu dans `motCourant`.

Mais que doit faire ce programme pour le premier mot quand il n'y a pas encore de mot précédent à comparer ? Il faut initialiser `motPrecedent` lors de sa définition de telle manière qu'il indique qu'il ne contient pas (encore) un mot. Il contiendra qu'un seul caractère (le caractère espace) car on sait que l'opérateur `>>` les ignore. Il sera donc impossible à `cin` de lire cette valeur et lors du premier passage dans la boucle `while`, le test `if` échouera (c'est ce que nous voulions).

On obtient ceci à l'exécution :

```
$ ./a.out
toto et titi titi vont à la la plage avec titi
mot répété : titi
mot répété : la
Ctrl-d
```

Remarque : sous Linux, vous pouvez mettre fin au programme en combinant les touches Ctrl et d qui indiquera qu'il n'y a plus de saisie. Vous pouvez aussi stopper le programme avec Ctrl-z ou l'interrompre avec Ctrl-c.

Conseil : Une façon de comprendre ce programme est de "jouer à l'ordinateur", autrement dit de suivre le programme ligne par ligne en faisant ce qu'il spécifie. Dessinez des boîtes sur une feuille de papier et inscrivez-y leurs valeurs. Puis changez les valeurs comme indiqué par le programme. Les programmeurs expérimentés font cela alors pourquoi pas vous ?

Question 7. Quelle est la signification exacte de "mots répétés" dans ce programme ?

Question 8. Modifiez ce programme afin qu'il compte et affiche à la fin le nombre de mots répétés adjacents dans une suite de mots. Pour vous aider, répondez aux questions suivantes qui sont celles qu'un programmeur doit se poser :

a) Proposez un nom et un type pour la variable qui doit compter le nombre de mots répétés.
b) Donnez sa valeur initiale.
c) Écrivez alors sa définition.
d) Repérez l'endroit où la détection d'un mot répété est réalisée.
e) Écrivez alors l'incrémentation du compteur. Si vous devez écrire plusieurs instructions pour un `if`, il vous faudra alors mettre un bloc avec des accolades (`{}`).

*Remarque : pour incrémenter une variable, on peut soit utiliser l'**opérateur d'incrémentation** (++) soit l'**addition** (+).*

```
int a = 4;

// Au choix :
++a; // après cette instruction, a aura pour valeur 5 (écriture conseillée)
a++; // après cette instruction, a aura pour valeur 6
a = a + 1; // après cette instruction, a aura pour valeur 7
a += 1; // après cette instruction, a aura pour valeur 8
```

Incrémentation d'un entier

Question 9. Améliorez le programme précédent en assurant un affichage grammaticalement correct : "Aucun mot répété adjacent dans cette suite", "un seul mot répété adjacent dans cette suite" ou "Il y a n mots répétés adjacents dans cette suite" (où n est une valeur strictement supérieure à 1).

 © 2012 tv <tvaira@free.fr>

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de ce TP.

Question 10. Qu'entend-on par le terme "invite" ?

Question 11. Quel opérateur utilise-t-on pour lire une valeur et la placer dans une variable ?

Question 12. Qu'est-ce qu'une variable ?

Question 13. Quelle est la différence entre `=` et `==` ?

Question 14. Qu'est-ce qu'une définition ?

Question 15. Qu'est-ce qu'une initialisation et en quoi diffère-t-elle d'une affectation ?

Question 16. Qu'est-ce qu'une concaténation de chaînes ? Quel est l'opérateur qui permet cela en C++ ?

Question 17. Qu'est-ce qui termine l'entrée d'une chaîne ? d'un entier ?

Question 18. Quelles sont les bonnes règles pour choisir un nom de variable ?

Question 19. Qu'est-ce qu'une incrémentation ?

Travail demandé

Exercice 1 : calcul de notes

L'objectif de cet exercice est de montrer la décomposition en itérations d'un problème simple.

Question 20. Calculer la somme d'un certain nombre de valeurs tapées par l'utilisateur

Créez un programme qui lit une **série de valeurs numériques réelles** tapées successivement par l'utilisateur. La saisie de la valeur -1 comme note signale au programme qu'il n'y a plus de note à taper ; cette note -1 ne doit pas être considérée comme une vraie note pour la suite du problème, c'est uniquement une marque de fin. Au fur et à mesure que les nombres sont lus l'un après l'autre dans une variable (que vous pouvez nommer `note`), additionnez-les dans une variable que vous nommerez `accumulateur` et que vous initialiserez précédemment à la valeur 0 (élément neutre de l'addition).

Question 21. Compter les valeurs en même temps.

Modifiez votre source pour que le programme, en plus de toujours calculer la somme des valeurs tapées, calcule aussi automatiquement leur nombre... sans compter le -1 !

Question 22. Calculer la moyenne des valeurs.

Ajoutez l'affichage de la moyenne des valeurs, c'est-à-dire tout simplement le total divisé par le nombre de valeurs. Attention : il y a un petit piège dans cette question ! Si, pris par l'habitude (pourtant pas bien vieille) de toujours créer des variables de type int, vous avez jusqu'ici commis l'erreur de choisir int comme type pour la note saisie ou pour l'accumulateur, vous allez obtenir un autre calcul que celui que vous désirez ! En effet, diviser un entier par un autre entier conduit à calculer le quotient entier de la division euclidienne. Si vous voulez calculer une division réelle, il suffit qu'un des arguments (ici, choisissons l'accumulateur) soit un réel (comme indiqué à la première question de l'exercice). Si l'accumulateur est un réel, je vous signale qu'il serait logique que la variable note le soit aussi.

 © 2012 tv <tvaira@free.fr>

Question 23. Contrôler les saisies.

Améliorez l'opération de lecture pour que le programme vérifie que la valeur tapée est bien comprise entre 0 et 20 inclus, ou qu'il s'agit de -1. Si la valeur n'est pas acceptable, alors l'utilisateur voit un message qui lui demande de recommencer sa saisie de cette valeur jusqu'à ce qu'elle soit correcte.

Question 24. Déterminer le minimum et le maximum.

Trouvez comment modifier votre algorithme pour que le programme calcule (et affiche, à sa fin) la valeur du minimum et du maximum de la série de note en plus du calcul de moyenne déjà obtenu aux questions précédentes.

Question 25. Écrire un programme robuste.

Que donne votre programme si on ne saisit aucune note, c'est-à-dire si on tape -1 dès le début ? Un bon programme ne doit pas planter, et donc prévoir ce cas.

Exercice 2 : pair ou impair

Question 26. Écrire un programme qui teste une valeur entière et détermine si elle est paire ou impaire. Comme toujours, faites en sorte que la sortie soit claire et complète. Autrement dit, ne vous contentez pas d'afficher "oui" ou "non". La sortie doit se comprendre toute seule comme "La valeur 4 est un nombre pair". Indice : l'opérateur reste (modulo) est `%`.

Exercice 3 : conversion

Question 27. Écrire un programme qui convertit des chiffres écrits en toutes lettres comme "zéro" ou "deux" et affiche le chiffre correspondant (0 ou 2 ici). Faites ceci pour les valeurs 0, 1, 2, 3 et 4 et affichez "Je ne connais pas ce chiffre" si l'utilisateur tape autre chose.

Exercice 4 : calculatrice

Question 28. Écrire un programme qui accepte une opération suivi de deux opérandes et affiche le résultat. Stockez l'opération dans une chaîne `operation` et utilisez une instruction `if` pour déterminer l'opération voule par l'utilisateur. Stockez les opérandes dans des variables de type `double`. Implémentez ceci pour les opérations +, -, *, /, plus, moins, mult et div avec leur sens évident.

Voici une exemple d'exécution :

```
$ ./a.out
Entrez une opération (+, -, *, /, plus, minus, mul, div) suivi de deux valeurs séparées par
   un espace : + 100 3.14
100+3.14 == 103.14
Un nouveau calcul : * 4 5
4*5 == 20
Un nouveau calcul : div 15 0
Attention division par zéro
Un nouveau calcul : robert 5 5
Erreur : opérateur inconnu robert
Un nouveau calcul :
```

 © 2012 tv <tvaira@free.fr>

Bilan

Notez comment ces premiers programmes se sont construits : progressivement en modifiant un peu le code pour atteindre un nouvel objectif. C'est une technique très courante : en présence d'un problème à résoudre, on recherche un problème similaire et on utilise sa solution en ajoutant les modifications appropriées. Ne partez donc pas de zéro à moins d'y être vraiment obligé. Se baser sur une version précédente économise beaucoup de temps et permet de tirer profit des efforts investis dans le programme d'origine.

Conclusion : Les types sont au centre de la plupart des notions de programmes corrects, et certaines des techniques de construction de programmes les plus efficaces reposent sur la conception et l'utilisation des types.

Rob Pike (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

> « Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. »

Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! »

 © 2012 tv <tvaira@free.fr>