TP1 : introduction au C++

1 La compilation

Le code correspondant à un programme est l'ensemble des instructions dans la fonction *main*.

Compiler le fichier *hello.cpp* en utilisant le *Makefile* : taper **make hello**. Observer les fichiers qui ont été créés (**ls**). Exécuter le code : taper **./hello**. Modifier le fichier source *hello.cpp* (par exemple changer le texte de sortie), recompiler et exécuter. Observer le changement de comportement. Taper **make clean** pour supprimer les fichiers de compilation. Observer leur disparition (**ls**).

2 Le typage des variables

La suite du TP utilise le fichier *var.cpp*.

Le C++ utilise un typage statique des variables. Ceci signifie que quand on introduit une nouvelle variable, il faut donner son type. Par exemple *int* pour un entier ou *float* (ou *double*) pour un réel. Ceci permet à l'ordinateur d'avoir accès à des informations comme par exemple l'espace mémoire qu'il doit réserver pour chaque variable.

Essayer de stocker un entier dans un *float*. Essayer de stocker un réel dans un *int*. Qu'observez-vous ? Obtenez-vous une erreur ? Vous pouvez aussi essayer d'autres types. Une conversion a lieu entre les différents types.

3 Les blocs de code et la portée des variables

On appelle bloc de code un morceau de code entre deux accolades (par exemple le code dans la fonction *main* est un bloc de code). On appelle portée d'une variable la partie du code dans laquelle cette variable est définie et donc peut être utilisée.

Les variables définies dans un bloc de code ont une portée qui vont de leur définition à la fin du bloc de code en question. En général, on ne définit une variable qu'au moment où on va l'utiliser pour la première fois. Ceci rend les fichiers plus lisibles.

Essayer d'utiliser une variable à l'extérieur du bloc de code où elle est définie. Obtenez-vous une erreur ? Cette erreur a-t-elle lieu à la compilation ou bien à l'exécution du code ?

4 Les opérateurs arithmétiques

Essayer d'effectuer les opérations suivantes : +,-,*,/ sur des réels (*float*), puis sur des entiers (*int*). En particulier, le comportement de la division est différent dans ces deux cas. Savez-vous à quoi correspond la division sur des entiers ?

Vous pouvez ensuite utiliser l'opérateur % entre deux entiers. $a\%b$ correspond au reste de la division euclidienne de a par b.

Pour finir, vous pouvez tester les opérateurs +=, -=, /=, *=, %=, ++ et --.

5 Les tests logiques

On peut effectuer des tests logiques avec les commandes *if*, *else* et *else if*. Le bloc suivant chacun de ces tests est exécuté uniquement si le booleen du test est vrai. Les *else if* et *else* ne sont considérés que si le *if* et les *else if* précédents ont des conditions fausses.

On peut évaluer une expression par les opérateurs ==, !=, <, <=, >, >=. De plus, on peut combiner des expressions simples par les opérateurs && (et) et || (ou). L'opérateur ! inverse la valeur d'un booleen : par exemple *!(true)* vaut *false*. Essayer de modifier les valeurs des variables et les tests pour comprendre le fonctionnement de cette partie.

Attention : ne jamais utiliser == ou != avec des réels. Préférer (abs(a-b) < 1.e-12) à (a == b).

6 Les boucles

On détaille trois boucles : *for*, *while* et *do-while*. Modifier les arguments de ces boucles et observer le résultat. Attention à ne pas créer une boucle infinie (si cela vous arrive, vous pouvez interrompre le programme avec Ctrl+C dans le terminal).

L'instruction *break* met fin à la boucle. L'instruction *continue* met fin à l'itération en cours.

7 Les fonctions

Pour cette dernière partie, on se place dans le fichier *fonctions.cpp*.

Quand les programmes deviennent grands, on les découpe en plusieurs fonctions. Chaque fonction remplit un rôle bien précis. Ceci permet d'organiser le code et donc de le rendre plus lisible.

Une fonction est composée d'un prototype et d'un bloc de code. Le prototype contient d'abord le type de retour de la fonction (*void* si la fonction ne retourne rien), son nom et les arguments qu'elle prend en entrée (entre parenthèses). A chaque argument on associe aussi son type. Par exemple :

```
int G(double a, bool b, int c)
```

signifie que la fonction s'appelle *G*, qu'elle renvoie un *int* et qu'elle prend en entrée trois arguments : un *double* (*a*), un *bool* (*b*) et un *int* (*c*).

Quand la fonction est appelée, le bloc de code s'exécute. On utilise l'instruction *return* pour indiquer la valeur à envoyer en sortie.

Les fonctions doivent être définies avant d'être utilisées, sinon une erreur apparaît. Si on ne veut pas la définir complétement, on peut se contenter d'écrire son prototype avant sa première utilisation et la fonction peut alors être définie plus loin (voir la fonction *f* dans le code).

Lorsque l'on définit une fonction, cette fonction est représentée par son nom et la lsite de ses arguments (avec leurs types). Ainsi, lorsque l'on définit une fonction, on ne peut pas définir une autre fonction avec le même nom et la même liste de types d'arguments. Par contre, on peut définir une autre fonction qui a le même nom mais pas la même liste de types d'arguments. Par exemple, si on définit une fonction

```
int G(double a, bool b, int c)
```

alors on peut définir une autre fonction

```
int G(double a)
```

ou même

```
int G(double a, int c, bool b)
```

mais par contre, on ne pourra pas définir une autre fonction

bool G(double a, bool b, int c)

(car il y a ici la même liste d'arguments)

Dans la définition d'une fonction, on peut aussi donner une valeur par défaut aux arguments. Par exemple, si l'on définit

int max(int a, int b=1)

et que l'on renvoie le *max* entre a et b, alors un appel à *max(5)* reviendra à un appel de *max(5,1)* : l'argument b prend la valeur 1 par défaut.

Quelle est la portée d'une variable définie dans une fonction ?

Essayez d'enlever le prototype de la fonction *f*, qu'observez-vous ?