Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC semaine 3: boucles

Exercice d'échauffement

Rappel: les boucles conditionnelles et l'itération

Bonnes pratiques en matière de portée d'une variable

Branchements inconditionnels avec break et continue

Une boucle particulière : la lecture

Boucle et accumulation d'erreurs

Exercice d'échauffement

Le code de **quel_est_le_resultat.cc** :

A : affiche 3 lignes de texte puis toutes les valeurs de 1.0 à 9.9
par pas de 0.1 et affiche the end.

B : affiche 3 lignes de texte puis entre dans une boucle infinie

C : affiche 3 lignes de texte puis la seule valeur 10.0 puis affiche the end

D : affiche 3 lignes de texte puis directement the end

E : ne compile pas

```
Affichage du chronometre à partir de : 1.0
Intervalle d'affichage: 0.1
Jusqu'à : 10.  (valeur exclue)
```

boucle conditionnelle : while

```
while( condition )
   une seule instruction contrôlée ;
```

1) Utiliser un bloc **{…} pour contrôler plus d'une instruction**
   - l'indentation ne suffit PAS !!
   - Si on dispose d'espace, toujours utiliser un bloc même avec une seule instruction contrôlée

2) **Ne PAS ajouter de point virgule en fin de ligne du while**
   le point virgule représente l'instruction nulle qui ne fait rien !!

boucle conditionnelle : do-while

```
do
    une seule instruction contrôlée ;
while( condition );
```

1) Utiliser un bloc **{…} pour contrôler plus d'une instruction**
   - l'indentation ne suffit PAS !!
   - utiliser un bloc avec **do-while** pour le rendre plus visible.

2) **NE PAS OUBLIER le point virgule en fin de ligne du while**
   il est nécessaire pour cette structure de contrôle

Usage classique: dialogue utilisateur et lecture d'une donnée puis vérification avec la condition du while.

Itération avec la boucle for «à la C»

```
for( init ; condition ; post )
    une seule instruction contrôlée ;
```

1) Utiliser un bloc **{…} pour contrôler plus d'une instruction**
   - l'indentation ne suffit PAS !!
   - Si on dispose d'espace, toujours utiliser un bloc même avec une seule instruction contrôlée

2) **Ne PAS ajouter de point virgule en fin de ligne du for**
   le point virgule représente l'instruction nulle qui ne fait rien !!

3) Privilégier l'itération **for** plutôt que **while** et **do-while**
   quand on connaît a priori le nombre de passages.

Itération avec la boucle for «à la C»

```
for( init ; condition ; post )
    une seule instruction contrôlée ;
```

Les expressions `init` et `post` peuvent contenir plusieurs sous-expressions séparées par une virgule.

```
for (int i(0),j(1) ; i<10 ; ++i,j*=2)
{
    cout << j << endl;
}
```

Bonnes pratiques en matière de portée des variables

1) Déclarer une variable au plus proche de son utilisation

2) **Déclarer une variable le plus localement possible**
⇒ seulement dans le bloc où elle est utile (bloc d'instructions contrôlées)

3) ATTENTION: **déclarer une variable globale en dehors de tout bloc**
=> EST UNE TRES MAUVAISE PRATIQUE

4) Règle de masquage des noms de variable: c'est «légal» de déclarer deux variables de même nom dans des blocs imbriqués. Dans ce cas, la variable du bloc le plus interne masque la variable du bloc externe (ex. code **masquage.cc**)

Branchements inconditionnels: `break` et `continue`

`break` permet de **quitter** la boucle dans laquelle il se trouve

En cas de boucles imbriquées : quitte seulement la boucle interne

Analogue à l'instruction «Sortir» utilisée pour décrire un algorithme

Ne pas en abuser **car le flot de contrôle devient moins lisible**

⇒ Augmente le nombre de points de sortie de la boucle
au lieu d'un avoir un seul dans le while ou le for

⇒ Il est préférable d'enrichir l'unique condition de la boucle

`continue` **ne quitte PAS la boucle ;**

effectue seulement un saut à la fin des instructions contrôlées

(ex. code **break_continue.cc**)

Une boucle «système» particulière: la lecture

Reprenons le code de **lecture_et_test.cc** :

Ce code lit successivement 3 nombres : un double, un int puis un float

La suite de **caractères alphanumériques** qui est fournie au clavier est **consommé selon le type de la variable** lue par les instructions **cin** successives du programme .

Que se passe-t-il avec: 0.1 1.1 suivi par Enter.

Il faut au moins un espace ou un passage à la ligne comme séparateur entre 2 valeurs destinées à 2 variable distinctes. Il peut y en avoir plusieurs : ils sont ignorés. Pour l'exemple ci-dessus, on a:

1) Lecture de la valeur **double 0.1** pour **a**

2) la lecture d'un **int** ne consomme QUE le **signe** et les **10 chiffres**.

PAS PLUS. La lecture consomme seulement **1** pour le **int b.**

3) la lecture se poursuit *avec les caractère non-consommés*.

Donc la lecture suivante récupère la valeur **double .1** pour **c**.

Boucle et accumulation d'erreurs

Examinons le code de **boucle_erreur_numerique.cc** :

Ce code montre qu'il est préférable de minimiser le nombre d'opérations pour obtenir un résultat recherché car l'erreur sur le résultat est directement liée **au nombre d'opérations effectuées**.

Dans cet exemple simple on a un moyen de calculer exactement la valeur théorique attendue ce qui permet de calculer l'erreur absolue sur un résultat obtenu par deux moyens de calcul:

- Une opération incrémentale sur le dernier résultat intermédiaire
- Une opération unique

Toujours se poser la question: mon résultat peut-il s'obtenir avec moins de calculs, idéalement un seul calcul, plutôt que incrémentalement ?