Université de Nice-Sophia Antipolis
Polytech/Xidian

mardi 3 novembre 2015
Durée: 1h30

Énoncés itératifs
Travaux Pratiques – Séance n° 5

---

1 Les énoncés itératifs

Les énoncés itératifs, appelés couramment *boucles*, servent à répéter une ou plusieurs instructions un certain nombre de fois jusqu'à ce qu'une condition arrêt soit vraie. Tout comme pour les conditions, il y a plusieurs sortes d'énoncés itératifs. Au bout du compte, cela revient à faire la même chose : répéter les mêmes instructions un certain nombre de fois. Il existe en langage C trois énoncés itératifs :

- **while**
- **do ... while**
- **for**

Théoriquement, seul le premier énoncé itératif est suffisant. Les deux autres peuvent s'exprimer en fonction du premier. Il est cependant recommandé d'utiliser l'énoncé le plus adapté au problème à résoudre. Nous allons étudier dans ce TP les deux premières formes de boucle.

1.1 La boucle while

Elle permet d'exécuter répétitivement une suite d'instructions tant qu'une condition est vraie :

```
while (condition)
  instruction
```

L'utilisation des blocs d'instructions est naturellement possible :

```
while (condition)
{
  instruction1
  instruction2
  instruction3
}
```

Notez que si la condition d'arrêt est immédiatement vérifiée, les instructions de la boucle ne sont pas exécutées. L'énoncé **while** permet l'exécution de zéro ou plusieurs fois des instructions du cours de la boucle.

Plutôt qu'un long discours, voyons plutôt un exemple simple. Dans l'exemple qui suit, nous allons calculer la somme des entiers compris entre `1` et `MAX` :

```
somme=0;
i=1;
while (i<=MAX) {
  somme+=i;
  i++;
```



```
}
```

1.2 La boucle do-while

L'énoncé **do-while** s'écrit de la manière suivante :

```
do
  instruction
while (condition);
```

Les énoncés **while** et **do-while** sont différents :

- l'énoncé **while** évalue la condition avant d'exécuter le bloc d'instructions.
- l'énoncé **do-while** évalue la condition après avoir exécuté le bloc d'instructions. Ainsi le bloc d'instructions est exécuté au moins une fois.

En pratique, à chaque fois que vous êtes sûr que votre boucle doit faire au moins une itération, vous utiliserez l'énoncé **do-while**.

```
do {
  printf("Entrez une année supérieure a 1600 :");
  scanf("%d",&annee);
}
while (annee<1600);
```

1.3 Invariant et finitude

Les itérations permettent d'écrire des programmes qui exécutent plusieurs fois les mêmes instructions. Cela constitue un saut conceptuel important : ce sont les seules instructions qui nous permettent d'écrire un programme dont le temps d'exécution est arbitrairement long, ou même infini, comme par exemple, le fragment de code (erroné) suivant qui ne s'arrête jamais :

```
a=1;
b=0;
while (a<2)
  b++;
```

Avec les boucles, il y a deux questions qui surgissent tout naturellement :

- Terminaison : la boucle se termine-t-elle ? Sous quelles conditions ?
- Sémantique : que calcule la boucle en question ?

Pour répondre à ces questions, nous utilisons les invariants de boucle. Un invariant de boucle est une affirmation qui reste vraie à chaque itération. Il est toujours possible de placer de telles affirmations avant et après chaque instruction du corps de la boucle. Toutefois, une affirmation joue un rôle particulier. C'est celle qui se trouve juste avant la condition d'arrêt de la boucle. On appelle cette affirmation *invariant de boucle* et cette affirmation décrit la *sémantique* de la boucle. Normalement, avant de programmer une boucle, il faut chercher son invariant et construire *ensuite* la boucle avec des instructions qui respecteront cet invariant.

Par exemple dans le cas d'une boucle qui calcule itérativement $n!$, une façon consiste à produire des entiers, un par un, et de multiplier chaque entier produit (nous utiliserons i comme identificateur de variable pour l'identifier) par le produit des entiers produits précédemment (ce produit sera conservé dans une variable que nous appellerons $fact$). La fonction d'arrêt de la

boucle doit être atteinte quand le produit des entiers vaudra $n!$. On voit qu'à la ième itération, nous aurons calculé $fact = i!$, ce qui est notre invariant de boucle.

À la fin des répétitions $fact = i!$ et $i = n$. Le choix logique de la raison pour boucler est quand $i < n$. Il est facile de constater maintenant que ce que nous voulons que le corps de la boucle fasse est de calculer $fact = i!$ pour tout i tel que $i <= n$. En nous basant sur les informations que nous venons de relever, la boucle peut être écrite comme suit :

— À chaque fois que la boucle procède à une itération, nous saurons que $i < n$ (c'est la raison de la boucle pour boucler) et que $fact = i!$;

— Nous devons produire dans le corps de la boucle un nouvel entier i (i.e. $i = i + 1$). Avec ce nouvel entier i, nous devons faire la multiplication avec le produit (i.e. le calcul courant) des entiers précédents contenu dans $fact$. Ainsi , le corps de la boucle a deux instructions : $i = i + 1$ et $fact = fact * i$.

— Nous devons procéder à l'initialisation de i et de $fact$ avant d'entrer dans la boucle pour que les instructions constituant le corps de la boucle puissent être exécutées. Puisque $fact = i!$ et $i <= n$ doivent être vrais à la fin de la boucle, ils doivent être également vrais à l'entrée de la boucle : les initialisations doivent faire de sorte que cette condition soit vraie initialement. La plus simple manière de faire est de poser $i = 1$ et $fact = 1! = 1$. L'invariant de boucle est par conséquence la propriété $fact = i!$ et $i <= n$.

L'algorithme complet pour calculer $n!$ en tenant compte de tout ce qu'on vient de dire est le suivant :

```
/* Antécédent : n >=0 */
/* Conséquent : factorielle = n! */
int factorielle(int n) {
  int i, fact;

  i=0;
  fact=1; /* fact = i! = 0! = 1  */
  /* Invariant : fact=i!  */
  while(i<n){
    /* fact * (i+1) = i!* (i+1) et i<n */
    i=i+1;
    /* fact*i= i!  */
    fact=fact*i;
    /* fact= i! */
  }
  /* i=n et fact= i! = n! */
  return fact;
}
```

L'invariant est essentiel pour justifier la validité de la boucle, mais il faut être certain que le processus itératif s'achève ; sinon on dit que le programme boucle. La finitude des énoncés itératifs (aussi appelée terminaison) est une propriété fondamentale des programmes informatiques. Une façon de démontrer la finitude d'une boucle est de rechercher une fonction $f(X)$ positive qui décroît strictement vers une valeur particulière. Pour cette valeur, par exemple zéro, on déduit que (non C) est vérifiée, et que la boucle s'achève. X représente des variables mises en jeu dans le corps de la boucle, et qui devront nécessairement être modifiées par le processus itératif. Par exemple dans le cas de la factorielle une fonction qui permet de démontrer la finitude de la boucle est $f(i) = n - i$. Lorsque $i = n$, le prédicat d'achèvement est vérifié.

1) En mathématiques, la factorielle d'un entier naturel n, notée $n!$ et appelée « factorielle de n » ou « factorielle n », est le produit des nombres entiers strictement positifs inférieurs ou égaux

à n. Soit $n \in \mathbb{N}$, sa factorielle est formellement définie par :

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

Écrivez un programme qui demande à l'utilisateur un entier naturel n.

Écrivez deux fonctions qui prennent comme paramètre donné ce nombre n et qui retournent sa factorielle :

— La première, nommée **int** `factorielleCroissant(int n)`, utilise une boucle **while** croissante, comme précédemment ;

— La seconde, nommée **int** `factorielleDecroissant(int n)`, utilise une boucle **while** décroissante.

Note : Commencez par chercher les invariants de boucles.

Nous pouvons utiliser ces deux implémentations car le produit de nombres entiers réels est commutatif.

2) La division entière de deux entiers naturels a et b se définit comme suit :

$$\forall a \geq 0,\ b > 0,\ a = q \times b + r,\ 0 \leq r < b$$

Pour calculer la division entière, nous allons procéder par soustractions successives de la valeur b de a jusqu'à ce que le résultat de la soustraction soit inférieur à b.
Écrivez la fonction **int** `divisionEntier(int a, int b)` basée sur cet algorithme et construite autour d'une boucle **while**.

3) Le produit de deux entiers x et y consiste à sommer y fois la valeur x.

$$x \times y = \underbrace{x + x + \cdots + x}_{y\text{fois}}$$

Toutefois, on peut améliorer cet algorithme rudimentaire en multipliant le produit calculé par deux et en divisant y par deux chaque fois que y est pair. Les opérations de multiplication et de division par deux sont des opérations très efficaces puisqu'elles consistent à décaler de un bit vers la gauche ou vers la droite.
Écrivez la fonction **int** `produit(int x, int y)` basée sur cet algorithme et construite autour d'une boucle **while**.

4) Dans cet exercice, nous allons dessiner des triangles isocèles sur la sortie standard en utilisant le caractère `'*'`. Un triangle est dit isocèle si :

— Un triangle a deux côtés de même longueur.

— Un triangle a deux angles de même mesure.

— Un triangle a un axe de symétrie.

La figure 1 illustre un triangle isocèle de sommet A et de base $[BC]$.

— Écrivez une fonction dessinant un triangle isocèle tel que $AB = AC = n$. Si $n = 6$, la sortie standard doit afficher le dessin suivant :

```
*
**
***
****
*****
******
```

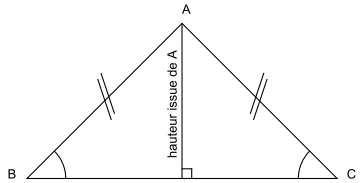


FIGURE 1 – Triangle isocèle

5) Écrivez une fonction dessinant un triangle isocèle tel que $BC = 2n - 1$ et tel que sa hauteur issue de A soit égale n. Si $n = 4$, la sortie standard doit afficher le dessin suivant :

```
*
**
***
****
***
**
*
```

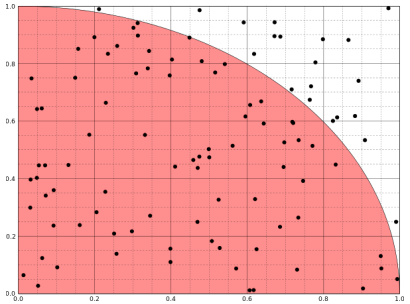
6) Écrivez une fonction dessinant un triangle isocèle tel que $BC = 2n - 1$ et tel que sa hauteur issue de A soit égale n. Si $n = 6$, la sortie standard doit afficher le dessin suivant :

```
     *
    ***
   *****
  *******
 *********
***********
```

1.4 Approximation de π par la méthode de Monte-Carlo

La méthode Monte-Carlo consiste à calculer une valeur numérique en utilisant des procédés aléatoires et des techniques probabilistes. Nous allons utiliser cette méthode pour calculer une approximation de π.

Soit un point M de coordonnées (x, y), où $0 \leqslant x < 1$ et $0 \leqslant y < 1$. On tire aléatoirement et de façon uniforme n valeurs x et y. Le point M appartient au cercle de centre $(0, 0)$ de rayon 1 si et seulement si $x^2 + y^2 \leqslant 1$. D'une façon générale, un point $M = (x, y)$ appartient à un cercle de centre (a, b) et de rayon r, si $(x - a)^2 + (y - b)^2 \leqslant r^2$.



La probabilité que le point M appartienne au cercle est égale à $S_c/S_k = \pi/4 = m/n$, où S_c est la surface du cercle, S_k est la surface du carré, et m le nombre de points M appartenant au cercle. On a donc $\pi/4 = m/n$ et donc $\pi = (m \times 4)/n$.

7) Écrivez un programme C qui calcule une approximation de π par cette méthode.