COURS 3
INSTRUCTIONS QUI CHANGENT L'ORDRE D'EXÉCUTION SÉQUENTIEL

- La conditionnelle: instructions "si" et "selon"
- Les boucles
- Comment raisonner sur les boucles: les invariants de boucle

Instructions conditionnelles

Elles ont la forme:

si <condition>
  alors <instruction>
fin si

(forme incomplète)

ou aussi:

si <condition>
  alors <instructions>
  sinon <instructions>
fin si

(forme complète)

Où <condition> est une expression de type booléen comme `a<b` construite en général avec les comparateurs `<, =, > ,...` et les opérateurs logiques `et, ou, non`

[0]Ces transparents sont basés sur l'excellent cours de Dominique Perrin à Marne La Vallée

Exemple

Calcul du minimum de deux valeurs a et b :

```
si (a < b)
  min <- a;
sinon
  min <- b;
fin si
```

Exemple

Minimum de trois valeurs :

```
si (a < b et a < c)
alors
    min <-  a
sinon
    si (b < c)
        min <- b
    sinon
        min <- c
    fin si
fin si
```

BRANCHEMENTS PLUS RICHES

Dans certains cas, l'utilisation de la conditionnelle pour l'aiguillage devient peu lisible.
Par exemple, pour écrire un chiffre en lettres:

```
si (x=0)
alors écrire "zéro"
sinon
  si (x=1)
  alors écrire "un"
  sinon
    si (x=2)
    alors écrire "deux"
```



```
    sinon
            ...
            si (x=9)
            alors écrire "neuf"
            sinon écrire "erreur"
            fin si
            ...
      fin si
    fin si
  fin si
fin si
```

BRANCHEMENTS PLUS RICHES: II

Si on a un choix avec plusieurs cas, on peut utiliser une instruction d'aiguillage, qui permet une écriture plus lisible.

```
selon (x)
  x=0: écrire "zéro"
  x=1: écrire "un"
  x=2: écrire "deux"
  ...
  x=9: écrire "neuf"
  sinon: écrire "erreur"
fin selon
```

CONDITIONNELLES EN C++

En C++, on retrouve les conditionnelles avec une syntaxe un peu plus concise. Voici un exemple de correspondance:

```
si (((x≠ 2) et (y=4) ou z <3)
alors
  écrire "trouve"
sinon
  écrire "non trouve"
fin si
```

devient

```
if (((x!=2)&&(y==4))||(z <3))
  {cout<<"trouve";}
else
  {cout<<"non trouve";};
```


BRANCHEMENTS MULTIPLES EN C++

En C++, on retrouve le branchement multiple dans la construction `switch`, et voici comment traduire le "selon":

```
selon (x)
  x=0: écrire "zero"
  x=1: écrire "un"
  x=2: écrire "deux"
  ...
  x=9: écrire "neuf"
  sinon: écrire "erreur"
fin selon
```

devient

```
switch (x) {
  case 0: cout<<"zero"; break;
  case 1: cout<<"un"; break;
  case 2: cout<<"deux"; break;
  ...
  case 9: cout<<"neuf"; break;
  default: cout<<"erreur";
}
```

Les itérations

1. Quatre formes possibles, deux types de boucles
2. Raisonner sur les boucles: les invariants et les fonctions de terminaison
3. Écriture en binaire
4. Traduction en C++

LES QUATRE FORMES POSSIBLES

Les boucles ont l'une des quatre formes suivantes:

1. boucle **pour**.
2. boucle **tant que ... faire**.
3. boucle **faire ... tant que**.
4. boucle **répéter ... jusqu'à**.

Mais on le classe aussi comme suit:

- boucles pour les itérations bornées (boucle **pour**)
- boucles pour itérations non bornée
  (boucle **tant que ... faire**, boucle **faire ... tant que** et boucle **répéter ... jusqu'à**)

Itération bornée: la boucle *pour*

pour <variable>`<-`<valeur initiale> *a* <valeur finale> *faire*
<liste d'instructions>
fin faire

comme dans:

```
pour i<-1 a n faire
  x <- x+1
fin faire
```

N.B.: il est *interdit* de modifier la variable de boucle dans le corps de la boucle, qui est automatiquement incrémentée à chaque cycle.
Si on respecte cette règle, on sait très exactement combien de fois la boucle est exécutée: <valeur finale> - <valeur initiale> +1 fois (si la valeur finale est au moins égale à la valeur initiale).

Exemples de boucle pour

Calcul de la somme des n premiers entiers:

```
s <- 0
pour i <- 1 a n faire
  s <- s+i
fin faire
```

Après l'exécution de cette instruction, `s` vaut:

$$1+2+\ldots+n=\frac{n(n+1)}{2}$$

Calcul de la somme des n premiers carrés:

```
s <- 0;
pour i <- 1 a n faire
   s <- s+i*i
fin pour
```

Après l'exécution de cette instruction, `s` vaut:

$$1+4+9+\ldots+n^2$$

Itération non bornée: la boucle *tant que ... faire*

Quand on ne connaît pas à priori le nombre exacte d'itérations à effectuer, on a recours aux itérations non bornées.

tant que <condition> *faire*
<liste d'instructions>
fin tant que

comme dans:

```
tant que (i ≤ n) faire
 i <- i+1
fin tant que
```

N.B. la vérification de la condition est faite en début de boucle, donc le corps est exécuté exclusivement quand la condition est vraie.

Exemple de boucle *tant que ... faire*

Calcul de la première puissance de 2 excédant (ou égalant) un nombre N:

```
p <- 1
e <- 0
tant que p < N  faire
  p <- 2*p
  e <- e+1
fin tant que
```

Résultats:
Pour N=50 on a p=64, e=6
Pour N=100 on a p=128, e=7
Pour N=200 on a p=256, e=8
...

Itération non bornée: la boucle *faire ... tant que*

faire
<liste d'instructions>
tant que <condition>

comme dans:

```
faire
 i <- i+1
tant que (i ≤ n)
```

N.B. la vérification de la condition se fait en fin de boucle, donc le corps de la boucle est exécuté au moins une fois.

Test de primalité

Voici un exemple de boucle *faire ... tant que*: le test de primalité d'un nombre entier.

```
lire n
d <- 1
faire
    d <- d+1
    r <- n % d
tant que (r ≥ 1 et  d*d < n)
si (r=0)
   écrire "nombre divisible par " d
sinon
   écrire "nombre premier"
fin si
```

Itération non bornée: la boucle *répéter ... jusqu'à*

repeter
<liste d'instructions>
jusqu'a <condition>

comme dans:


```
répéter
 i <- i+1
jusqu'à (i > n)
```

N.B. la vérification de la condition se fait en fin de boucle, donc le corps de la boucle est exécuté au moins une fois.

Relation entre les itérations

La boucle

pour <variable> `<-` <valeur initiale> *a* <valeur finale> *faire*
<liste d'instructions>
fin faire

est équivalente à

<variable> `<-` <valeur initiale>
tant que <variable> ≤ <valeur finale> *faire*
<liste d'instructions>
<variable> `<-` <variable>+1
fin faire

Relation entre les itérations

Aussi, la boucle

faire
<liste d'instructions>
tant que <condition>

est équivalente à

<liste d'instructions>
tant que <condition> *faire*
<liste d'instructions>
fin faire

Quelques considération

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:

```
tant que (1=1) faire
   i<-i
fin faire
```

ne s'arrête jamais.

Quelques considération: II

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

terminaison la boucle en question termine-t-elle?
Sous quelles conditions?

sémantique que calcule la boucle en question?

Pour les itérations bornées (boucle *pour*), le problème de la terminaison est trivial, alors que cela n'est pas toujours évident pour les autres.
Pour répondre à la deuxième question, on s'aiderait de propriétés appelées *invariants de boucle*.

Itération et récurrence

Les programmes itératifs sont liés à la notion mathématique de *récurrence*.

Par exemple la suite de Fibonacci est la suite de nombres définie par récurrence par $f_0 = f_1 = 1$ et pour $n \geq 2$ par

$$f_n = f_{n-1} + f_{n-2}$$

Voilà les premiers nombres de Fibonacci:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Elle peut être calculée par le fragment de programme suivant (qui affiche les MAX premières valeurs):


```
i,fi,fipred,temp: quatre entiers
fipred <- 1
fi <- 1                        /* Precondition: fibpred=fib(0), fi=fib(1) */
pour i<-1 a MAX faire          /* Invariant: fipred = fib(i-1), fi=fib(i) */
    temp <- fi+fipred
    ecrire temp
    fipred <- fi
    fi <- temp
fin pour
```

Résultat :

2	3	5	8	13	21	34
55	89	144	233	377	610	987
1597	2584	4181	6765	10946	17711	

Invariants de boucles: itérations bornées

Sur des itérations, on raisonne par *récurrence*, en utilisant une propriété vraie *à chaque passage* dans la boucle, appelée *invariant de boucle*. Par exemple, dans le programme précédent, la propriété:

$$P(i): \; fipred = f_{i-1}, fi = f_i$$

est un invariant de boucle: elle est vraie chaque fois que l'on passe par le debut de la boucle (elle l'est juste avant de commencer la boucle, elle reste vraie à chaque itération, et elle est vraie á la sortie de la boucle).

Pour le démontrer, on doit

1. vérifier qu'elle est vraie à la première entrée (pour $i =< valeurinitiale >$). Cela corresponde au cas de base d'une preuve par induction.
2. Que si elle est vraie à un passage (pour i), elle est vraie au suivant (pour $i+1$). C'est l'étape inductive.

Invariants de boucles: plus formellement

Pour que une certaine propriété P soit un invariant de la boucle:


```
pour i<-M a N faire
   <corps de la boucle>
fin pour
```

il faut que

- P soit vrai pour le valeur initial M de i et les valeurs des autres variables juste avant la boucle
- si $i \leq N$ et l'invariant P est vrai pour cette valeur de i, alors P est vrai pour les nouvelles valeurs des variables après l'exécution du corps de la boucle

Si P est bien un invariant de boucle, à la fin de la boucle on saura automatiquement que:

$$i = N + 1 \text{ et } P$$

Un autre exemple : La fonction factorielle

Le programme suivant

```
programme factoriel
  i,fact: deux entiers
  debut factoriel
    lire n
    fact<-1
    pour i<-2 a n faire
        fact <- fact * i
    fin faire
    ecrire "n!=" fact
  fin factoriel
```

calcule

$$n! = 1 \times 2 \times \cdots \times n$$

L'invariant de boucle est $fact =!(i-1)$.

A la sortie de la boucle, on sait donc que $i = n + 1$ et $fact = !(i-1) = !n$.

Invariants de boucles: le cas des itérations non bornées

Pour les boucles non bornées, on utilise aussi une propriété P vraie à chaque passage dans la boucle, appelée toujours *invariant de boucle*.
Pour démontrer que P est un invariant, on doit vérifier que

1. ell'est vraie à la première entrée
   (et donc pour les valeurs des variables juste avant l'entrée dans la boucle).
2. *si* la condition C de boucle est satisfaite, et l'invariant est vrai pour les valeurs des variables avant l'exécution du corps de la boucle, *alors* ell'est vraie aussi pour les nouvelles valeurs des variables après l'exécution du corps de la boucle.

Si P est bien un invariant de boucle, à la fin de la boucle on saura automatiquement que:

$$(\text{non } C) \text{ et } P$$

Un exemple d'invariant pour boucle non bornée: la division par soustraction

On veut calculer le quotient et le reste de la division entière de deux entiers positifs x et y; donc, on cherche q et r t.q. $x = q * y + r$ et $0 \leq r < y$.

On peut arriver facilement à écrire le programme suivant

```
q <- 0
r <- x
tant que (r ≥ y) faire
  q <- q+1
  r <- r-y
fin faire
```

Pour prouver que à la sortie de la boucle q est le quotient de la division et r le reste, il nous faut un invariant!

Un exemple d'invariant pour boucle non bornée: la division par soustraction

On arrivé à proposer

$$x = q * y + r \text{ et } r \geq 0$$

comme invariant; il nous faut donc vérifier que, pour x et y positifs

- x = q*y +r en entrée de boucle. Mais là, on a q=0 et r=x, donc q*y+r= 0*y+x =x. De plus, $r = x \geq 0$
- si x=q*y+r et $r \geq 0$ avant l'exécution du corps de la boucle, et la condition $r \geq y$ est vraie, alors x=q'*y+r', et $r' \geq 0$, où q' et r' sont les nouvelles valeurs de q et r après l'exécution du corps de la boucle. En effet, q'*y+r'=(q+1)*y+(r -y) = q*y+y+r-y =q*y+r =x, et puisque $r \geq y$, alors $r' = r - y \geq 0$.

A la sortie, l'invariant reste vrai, mais la condition de boucle est fausse (i.e. $r < y$), donc on obtient, comme souhaité,

$$x = q * y + r \text{ et } r \geq 0 \text{ et } r < y$$

Un exemple plus difficile : l'écriture en binaire

La représentation binaire d'un nombre entier x est la suite $(b_k, \dots, b_1, b_0)$ définie par la formule:

$$x = b_k 2^k + \cdots + b_1 2 + b_0$$

Principe de calcul : en deux étapes

1. On calcule $y = 2^k$ comme la plus grande puissance de 2 t.q. $y \leq x$.
2. Itérativement, on remplace y par $y/2$ en soustrayant y de x à chaque fois que $y \leq x$ (et en écrivant 1 ou 0 suivant le cas).

Proposez un programme qui resout le problème et un invariant de boucle pour le prouver.

La terminaison

Pour s'assurer qu'un boucle termine, on cherche à trouver une fonction des valeurs des variables modifiées dans la boucle qui diminue strictement à chaque itération.

Dans nos cas précédents

- division par soustraction: r diminue de y à chaque pas
- écriture binaire: y diminue strictement à chaque pas

Mais il y a des cas plus difficiles, comme pour le programme qui génère les nombres premiers inférieurs à Max.

Boucles en C++

En C++, on retrouve les boucles pour, faire ... tant que et tant que... faire voici comment traduire la boucle "pour":

```
pour   <variable><-<init>   a <fin>   faire
     <liste d'instructions>
fin faire
```

devient

```
for(<variable>=<init>;<variable><=<fin>;<variable>++) {
   <liste d'instructions>
};
```

Boucles en C++

Voici comment traduire la boucle "tant que ... faire":


```
tant que  <condition>  faire
        <liste d'instructions>
fin tant que
```

devient

```
while(<condition>) {
     <liste d'instructions>
};
```

Boucles en C++

Voici comment traduire la boucle "faire ... tant que":

```
faire
     <liste d'instructions>
tant que <condition>
```

devient

```
do  {
   <liste d'instructions>
} while (<condition>);
```

Conditionnelles et boucles en C++

Voici la traduction en C++ de notre test de primalité


```
lire n
d <- 1
faire
    d <- d+1
    r <- n % d
tant que (r ≥ 1 et  d*d < n)
si (r=0)
   écrire "nombre divisible par " d
sinon
   écrire "nombre premier"
fin si
```

devient ...

```
#include <iostream.h>

int main (){
int n,d,r;
cin>> n;
d=1;
do {
    d=d+1; r=n%d;
} while ((r>= 1) && (d*d<n));
if (r==0)
     { cout<<"divisible par "<<d<<endl;}
else { cout<<"nombre premier "<<endl;};
}
```