Itérations

Il y a 4 formes de boucles :

- Pour
- Tant que . . .faire . . .
- Faire . . .tant que . . .
- Répéter . . .jusqu'à . . .

La première suppose la connaissance du nombre d'itérations à réaliser

Les 3 suivantes sont les itérations non bornées contrôlées par une condition

- de continuation en début de boucle
- de continuation en fin de boucle
- de terminaison en fin de boucle

Problèmes soulevés

- terminaison d'une boucle
- correction de la boucle

Itération bornée : boucle Pour

Adaptée aux situations dans lesquelles le nombre d'itérations est connu

- typiquement entre deux valeurs extrêmes formant une progression arithmétique
- pour parcourir et traiter les éléments d'un tableau d'éléments (c'est-à-dire des éléments de même type regroupés sous un même nom et désignés par un indice ou index)

Dans la forme la plus simple de la boucle,
la raison de la progression arithmétique est **1**

```
Pour vc <- expression1 à expression2 faire
     suite d'instructions
finPour
```

et sa traduction dans le langage C-- est

```
for (vc=expression1;vc<=expression2;vc++)
     bloc d'instructions  // corps de la boucle
```


Exemple : Puissance par multiplications successives

Dans ce qui suit, la variable p a la propriété de prendre successivement les valeurs x^i, i variant de 1 à n.

```
Fonction Puissance (x, n: entier)
                              résultat entier
  variables i, p: entier
Début
  p <- 1
  pour i <- 1 à n faire
      p <- p * x
  finPour
  retourner p
Fin
```


Un programme

```
#include <iostream.h>
int puissance(int x, int n) { // puissance n de x
   int i;
   int p = 1;
   for(i = 1; i <= n; i++)
        p = p * x;
   return p;
}
int main ( ) {
   int x, n;
   cout << "Donner le nombre entier x : ";
   cin >> x;
   cout << "Donner la puissance : ";
   cin >> n;
   cout << "Le résultat est : "
                      << puissance(x,n) << endl;
   return 0;
}
```

Autre exemple : fonction calculant $\sum_{i=1}^{m} i^n$

Dans l'algorithme suivant, **on réutilise** la fonction Puissance écrite précédemment pour calculer i^n pour des valeurs données de i et n.

Pour chaque valeur de $i \geq 1$, avant d'exécuter le corps de la boucle, on a
$s = \sum_{j=1}^{i-1} j^n$ et la propriété est encore vraie à la sortie de la boucle.

```
Fonction SommePuissance (m, n: entier)
                              résultat entier
variables i, s: entier
Début
  s <- 0
  pour i <- 1 à m faire
      s <- s + Puissance(i, n);
  finPour
  retourner s
Fin
```


Exemple : Un programme

```
#include <iostream.h>
/* ICI CE QU'IL FAUT POUR UTILISER */
/*     LA FONCTION puissance       */
int sommePuissance(int m, int n) {
   int i, s = 0;
   for(i = 1; i <= m; i++) s = s + puissance(i, n);
   return s;
}
int main ( ) {
   int m, n;
   cout << "Donner le nombre de nombres entiers : ";
   cin >> m;
   cout << "Donner la puissance : "; cin >> n;
   cout << "Le résultat est : "
                << sommePuissance(m,n) << endl;
   return 0;
}
```

Question : comment réutiliser la fonction puissance ?

- soit insérer son code dans le code précédent
- créer un fichier puissance.c, le compiler sous la forme
  `g-- -c puissance.c`.
  On obtient ainsi un fichier `puissance.o` utilisable comme argument supplémentaire de la commande g--.
  Par contre, il est important de déclarer la fonction puissance (sans en donner la définition) sous la forme
  `int puissance(int, int);`

Autres formes de la boucle "pour"

Il est possible d'utiliser une valeur entière différente de 1 pour faire varier la variable de contrôle d'une boucle.

Dans le pseudo-langage, on peut par exemple écrire

```
Pour vc <- expression1 à expression2 par pas faire
     suite d'instructions
finPour
```

où pas est le nombre entier utilisé pour modifier vc.

Nous donnons les formes les plus utiles de la boucle "pour" dans le langage de programmation utilisé :

```
for(v = n1; v < n2; v += entierPositif) ........
for(v = n1; v <= n2; v += entierPositif) ........
for(v = n1; v > n2; v --) ........
for(v = n1; v >= n2; v --) ........
for(v = n1; v > n2; v -= entierPositif) ........
for(v = n1; v >= n2; v -= entierPositif) ........
```

On en verra des applications dans les exercices

De la boucle "pour" à la boucle "tant que" et la notion de tableau

On considère une suite de n nombres a_1, a_2, . . ., a_n dont on souhaite déterminer le maximum.

Une solution itérative consiste à
- supposer que le premier élément, c'est-à-dire a_1, est le plus grand et mémoriser sa valeur x (ou sa position)
- parcourir la suite à partir du second et pour chaque élément d'index i, le comparer à x. Si cet élément a_i est plus grand que x, x prend a_i comme nouvelle valeur

Dans le processus itératif ainsi décrit, on voit que pour une position i donnée dans la suite, on va comparer a_i à une valeur x calculée précédemment, la propriété

$$x = max(\{a_j / 1 \leq j < i\})$$

étant supposée vraie.

Deux approches sont possibles :
- réaliser une itération dans laquelle on lit un élément, on le traite et on l'oublie
- lire tous les éléments en les mémorisant, puis faire le traitement. L'avantage de cette seconde approche est évidemment de permettre de réaliser d'autres traitements sur la suite

Solution 1 : Boucle tant que

Dans cette première solution on réalise une itération dont le nombre de pas n'est pas connu.

La forme la plus courante de ce type d'itérations est la boucle "tant que" qui s'exprime dans le pseudo-langage sous la forme

```
tant que condition faire
    liste d'instructions
finTantQue
```

Son interprétation est la suivante :
(1) On commence par évaluer la condition
(donc, éventuellement la liste d'instructions constituant le corps de la boucle ne sera exécutée aucune fois)
(2) Si elle est fausse, on sort de la boucle
(3) Si elle est vraie, on effectue la liste d'instructions.
Puis on recommence en (1)

Le principe de la première solution est donc de continuer l'itération tant que la valeur lue n'est pas égale à une valeur convenue à l'avance marquant la fin de la suite.

Dans la solution donnée ci-dessous sous forme d'une séquence écrite dans le pseudo-langage, on itère jusqu'à ce qu'on lise une valeur négative.

L'interprétation des variables utilisées est la suivante :
– a est utilisé pour lire un élément de la suite
– n est le nombre d'entiers déjà lus
– M est le maximum des entiers déjà lus
Les points de suspension correspondent à l'exploitation qu'on souhaite faire du résultat : on sait qu'en cet endroit M contient le maximum des n nombres qui ont été lus
et on aura $M = -1$ si et seulement si $n = 0$


```
variables a : entier
          n : entier initialisé à 0
          M : entier initialisé à -1
Début :
   Afficher
      "Entrer un nombre [négatif pour arrêter] ? "
   lire a
   tant que a >= 0 faire
      n <- n + 1
      si a > M alors M <- a finSi
      Afficher
         "Entrer un nombre [négatif pour arrêter] ?
      lire a
   finTantQue
   .............
Fin
```


Traduction en un programme

Une boucle de la forme

```
tant que condition faire
    liste d'instructions
finTantQue
```

se traduit dans le langage C-- sous la forme :

```
while (condition)
    bloc d'instructions
```

d'où le programme suivant :

```
#include <iostream.h>
int main( ) {
   int a, n = 0, M = -1;
   cout << "Entrer un entier [<0 pour arrêter] ? ";
   cin >> a;
   while (a >= 0) {
     n++;
     if (a > M)  M = a;
     cout << "Entrer un entier [<0 pour arrêter] ? ";
     cin >> a;
   }
   if (M == -1) cout << "La suite est vide\n";
   else
     cout << "Le maximum des " << n
          << " nombres lus est " << M << endl;
   return 0;
}
```

Solution 2 : traitement après mémorisation

Dans cette seconde solution, on suppose que les éléments de la liste ont été préalablement lus et sont accessibles par un index dans une suite.

On utilise les **notations** suivantes :
- a[i] désigne l'élément indexé par i dans la suite a
- les éléments de la suite sont indexés par 1, 2, . . ., n

Une fonction réalisant le calcul du maximum d'une suite de n nombres qu'elle reçoit en paramètre (ainsi que n) s'exprime alors en pseudo-langage :

```
Fonction Maximum(n : entier; a : suite d'entiers)
                               résultat entier
variables i, x : entier
Début
  x <- a[1]
  pour i <- 2 à n faire
      si x < a[i] alors
          x <- a[i]
      finSi
  finPour
  retourner x
Fin
```


Traduction en un programme

- on utilise un **tableau** qui regroupe sous un même nom une suite de nombres entiers
- les éléments du tableau seront indexés par 0, 1, . . .n-1.
  Ceci est imposé par le langage `C--`

On remarquera sur le code la manière dont le tableau est défini dans la fonction `main` et comment il apparaît dans l'en-tête de la fonction `maximum`

```
#include <iostream.h>
int maximum(int a[ ], int n) {
   int i, x;
   x = a[0];
   for(i = 1; i < n; i++)
        if (x < a[i]) x = a[i];
   return x;
}
int main ( ) {
   int i, t[5];   // UN TABLEAU DE 5 ENTIERS
   for(i = 0; i < 5; i++){
      cout << "valeur de c[" << i << "] ? ";
      cin >> c[i]; }
   cout << maximum(t, 5) << endl;
   return 0;
}
```


Autres exemples de boucle "Tant que"

Exemple 1 : Division euclidienne par soustractions successives

On suppose que les nombres sont positifs

```
Fonction Division(a, b : entier)
                       résultat couple d'entiers
  variables quotient, reste : entier
Début
  quotient <- 0
  reste <- a
  tant que reste >= b faire
      reste <- reste - b
      quotient <- quotient + 1
  finTantQue
  retourner le couple (quotient ,reste)
Fin
```

Exécution à la main pour des valeurs de a et b particulières

Remarque...

Rappelons que le quotient q et le reste r de la division euclidienne de deux entiers positifs a par b sont définis par :

$$a = b \times q + r \land 0 \leq r < b \tag{1}$$

On remarquera qu'à chaque itération on a dans le programme précédent :

`a = b*quotient + reste`

D'autre part, on sort de la boucle quand `reste` est inférieur strictement à b ce qui assure (1)

Traduction de l'algorithme en un programme

Le principal problème qui se pose est de représenter les couples d'entiers.

Cela est réalisée par une **structure** dont la définition est donnée ci-dessous :

```
struct couple {
   int quotient;
   int reste;
   }
```

Après cette définition l'identificateur `couple` a le même statut que par exemple `int` ou `float`. Il est donc possible de déclarer un couple sous la forme

```
couple c;
```

Enfin étant donné un couple c ainsi déclaré,

`c.quotient` et `c.reste`

identifient respectivement les deux entiers constituant ce couple.

Cela conduit au programme C-- suivant :

```
#include <iostream.h>
struct couple{
   int quotient;
   int reste;};
couple division(int a, int b) {
   couple resultat;
   resultat.quotient = 0;
   resultat.reste = a;
   while (resultat.reste >= b) {
      resultat.reste = resultat.reste - b;
      resultat.quotient = resultat.quotient + 1;}
   return resultat;
}
int main () {
   int x, y;
   couple c;
   cout << "entrer la valeur de x ? " ; cin >> x;
   cout << "entrer la valeur de y ? " ; cin >> y;
   if (x >= 0 && y > 0) {
      c = division(x, y);
      cout << c.quotient << "  " << c.reste << endl;
      }
   else
      cout << "valeurs incorrectes pour (x,y)\n";
   return 0;
}
```


Puissance (autre version)

Remarquons que :

$$x^{2m+1} = x^{2m} \times x \quad (2)$$

$$x^{2m} = x^m x^m \quad (3)$$

De ces propriétés on peut déduire une autre façon de calculer x^n :

```
Fonction Puissance(x, n: entier)
                             résultat entier
  variables p : entier
Début
  p <- 1
  tant que n > 0 faire
      si n est impair
          alors p <- p * x
      finSi
      x <- x * x
      n <- n div 2
  finTantQue
  retourner p
Fin
```

Exécution à la main par exemple pour $x = 2$ et $n = 10$

Questions :

- Le nombre d'itérations dépend-il de la valeur du paramètre x ?
- Combien y-a-t-il d'itérations pour $n = 2$, $n = 8$, $n = 1024$?
- Combien y-a-t-il d'itérations si n est un nombre représenté en binaire par $x_n x_{n-1} \ldots x_0$ chaque x_i étant un chiffre binaire avec $x_n = 1$?
- Quels problèmes peut-on avoir si $x = 2$ et $n > 32$?
- Pourquoi cet algorithme est-il plus "efficace" que celui présenté au début ?

Comment ça marche ?

Notons x_0 et n_0 les valeurs des paramètres de la fonction `Puissance`. On suppose que $n_0 > 0$.

On remarque qu'à chaque début et fin d'itération, en notant n, p, x les valeurs des variables `n`, `p` et `x`, on aura :

$$x_0^{n_0} = p \times x^n \wedge n \geq 0 \qquad (4)$$

Une telle propriété est appelée un *Invariant* de boucle.

Vérifions-le !

En effet :

avant la première itération, $p = 1$, $n = n_0$ et $x = x_0$ et donc aussi (4)

ensuite, si n, p et x (respectivement n', p' et x') représentent les valeurs des variables `n`, `p` et `x` *avant* (respectivement *après*) l'exécution du corps de la boucle :
– Supposons que (4) est vérifié avant une itération, si le corps de la boucle est exécutée on a $n > 0$.
• Si n est pair nous avons $x' = x^2$ et $n = 2n'$ et $p' = p$.
Par conséquent, $n' \geq 0$ et

$$x_0^{n_0} = px^n = p'x^{2n'} = p'(x^2)^{n'} = p'(x')^{n'}.$$

• Si n est impair nous avons $x' = x^2$ et $n = 2n'+1$ et $p' = px$.
Par conséquent, $n' \geq 0$ et

$$x_0^{n_0} = px^n = px^{2n'+1} = px(x^2)^{n'} = p'(x')^{n'}.$$

Dans les deux cas, (4) est bien vérifié à la fin de l'itération.

– Donc, en sortant de la boucle nous avons

$$x_0^{n_0} = p \times x^n \text{ et } n \geq 0 \text{ et } n \leq 0.$$

Par conséquent, $n = 0$ et $p = x_0^{n_0}$ et le programme calcule bien x^n.

Invariants...

Pour trouver la solution d'un problème par itérations, il est souvent plus simple de considérer ce qui ne change pas à chaque itération . . . :

Exemple :
On a une urne contenant b billes blanches et n billes noires, tant que l'urne contient au moins deux billes on procède ainsi :
- Prendre 2 billes dans l'urne,
  - Si les deux billes ont la même couleur les jeter,
  - Si les deux billes sont de couleurs différentes, jeter la bille noire et remettre dans l'urne la bille blanche,

  De plus si l'urne est vide on remet une bille noire dans l'urne.

Question :
- Montrer que ce processus termine toujours (facile)
- Quelle est, suivant b et n, la couleur de la dernière bille dans l'urne (moins facile)

Solution

- Il est difficile de trouver la solution en essayant des valeurs particulières de n et b,
- Il vaut mieux s'intéresser à ce qui ne change pas à chaque itération :

la parité de b reste la même à chaque itération !

Solution :
La dernière bille est blanche si et seulement si b est impair.

Prouvons-le !

- On termine toujours car, si le nombre de billes dans l'urne n'est pas nul, le nombre de billes diminue d'au moins un à chaque tirage.
- Soit B le nombre de billes blanches avant un tirage, et B' le nombre de billes blanches à la fin de ce tirage.
  On a les cas suivants :
  - soit les deux billes choisies sont deux billes blanches alors $B' = B - 2$,
  - soit les deux billes choisies sont deux billes noires alors $B' = B$,
  - soit les deux billes sont de couleurs différentes alors $B' = B$,
  - soit l'urne était vide et on a rajouté une bille noire alors $B' = B$,

  Dans tous les cas, B' est pair si et seulement si B est pair.
- On arrête les tirages quand le nombre de billes est égal à un. D'après ce qui précède, à ce tirage le nombre B_0 de billes blanches a la même parité que b :
  - Si b est pair, alors B_0 est pair, comme $0 \leq B_0 \leq 1$ on a $B_0 = 0$ et la dernière bille est donc noire,
  - Si b est impair, B_0 est impair, comme $0 \leq B_0 \leq 1$ on a $B_0 = 1$ et la dernière bille est blanche.

Preuve de correction d'une boucle : Invariants

```
// Invariant : I
tant que C faire
    liste d'instructions
finTantQue
```

- Montrer que I est vrai avant d'entrer dans la boucle.
- Montrer que si C et I sont vrais, alors, après l'exécution de la liste d'instructions, I est encore vrai.
- On en déduit que (I et non C) est vrai à la sortie de la boucle (si la boucle termine).

De l'invariant à la boucle

On suppose dans la suite que a et b sont des entiers positifs ou nuls

Partons de :

Théorème :
(1) $pgcd(a, a) = pgcd(a, 0) = a$
(2) $pgcd(a, b) = pgcd(b, a)$
(3) Si a et b sont deux entiers tels que $a > b$ alors $pgcd(a, b) = pgcd(a - b, b)$

Exercice :
Prouvez-le !

Le (3) du théorème précédent fournit un "invariant" pour une boucle calculant le $pgcd$ de deux nombres :

si quand a est plus grand que b on remplace a par $a - b$ on ne change pas le $pgcd$.

Exercice :
- écrire un programme pour calculer le $pgcd$ de deux nombres utilisant ce théorème,
- écrire explicitement l'invariant de ce programme,
- prouver que le programme est correct.

pgcd

```
fonction PGCD(a,b : entiers)
  Variables m,n;
  m<-a;
  n<-b;

  tant que m != n faire
      si m > n alors m <- m - n finsi
      si m < n alors n <- n - m finsi
  finTantQue
  retourner n
```

INV : $pgcd(a, b) = pgcd(m, n)$

Prouvons-le !

INV est un invariant :

1. Juste avant la boucle, $m = a$ et $n = b$, et donc trivialement $pgcd(a, b) = pgcd(m, n)$
2. Soient m et n (resp. m' et n') les valeurs des varaibles m et n avant (resp. après) l'itération.
   Supposons que INV soit vrai avant l'itération : $pgcd(a, b) = pgcd(m, n)$. Considérons les cas :
   – $m > n$: après l'itération on a :
   $pgcd(m', n') = pgcd(m - n, n)$ (car $m' = m - n$)
   $pgcd(m - n, n) = pgcd(m, n)$ (par (3))
   $pgcd(m, n) = pgcd(a, b)$ (par hypothèse)
   – $m < n$: après l'itération on a :
   $pgcd(m', n') = pgcd(m, n - m)$ (car $n' = n - m$)
   $pgcd(m, n - m) = pgcd(n - m, m)$ (par (2))
   $pgcd(n - m, m) = pgcd(m, n)$ (par (2) et (3))
   $pgcd(m, n) = pgcd(a, b)$ (par hypothèse)
   Dans tous les cas, on a bien :
   $INV : pgcd(m', n') = pgcd(a, b)$
3. Si la boucle termine, au sortir de la boucle, on aura :
   $pgcd(a, b) = pgcd(m, n) \wedge m = n$
   Du (1) du théorème, on déduit :
   $n = pgcd(a, b)$

autre théorème autre $pgcd$...

Un autre théorème :

Théorème :

(4) Si $a = bq + r$ alors $pgcd(a, b) = pgcd(b, r)$

En déduire une algorithme de calcul du $pgcd$ utilisant (4) au lieu de (3)

Attention...

– Tout invariant n'est pas bon à prendre : il faut qu'il soit suffisamment puissant pour résoudre le problème (ou prouver le programme).
  $1 = 1$ est un invariant de toutes les boucles...
– Il faut aussi que les valeurs des variables initiales vérifient aussi certaines conditions (on appelle ces conditions des *préconditions*) par exemple dans les exemples précédents, on a supposé que a et b étaient positifs ou nuls.
– Il faut aussi que le programme termine.

Terminer...

```
fonction PGCD(a,b : entiers)
  Variables m,n;
  m<-a;
  n<-b;
  tant que m != n faire
      si m > n alors m <- m - n finsi
  finTantQue
  retourner n
```

On vérifie aisément que INV : $pgcd(a, b) = pgcd(m, n)$ est bien un invariant du programme, mais ce programme ne termine pas toujours...

Terminaison des boucles

Attention à la terminaison des boucles

Exemple : Reprenons le programme de division euclidienne par soustractions et vérifions que ce programme termine

```
Fonction Division(a, b : entier)
                       résultat couple d'entiers
  variables q, r : entier
Début
  q <- 0
  r <- a
  tant que r >= b faire
      r <- r - b
      q <- q + 1
  finTantQue
  retourner le couple (q ,r)
Fin
```

Exécution à la main pour $a = 27$ et $b = -2$.

Exécution à la main pour $a = 27$ et $b = 0$.

Prouver la terminaison des boucles

Définir une expression entière qui dépend des variables.

On note e (resp. e') la valeur de l'expression avant (resp. après) l'exécution des instructions de la boucle.

Montrer que :

- soit $e' < e$ et la condition de la boucle devient fausse lorsque l'expression devient inférieure à une constante,
- soit $e' > e$ et la condition de la boucle devient fausse lorsque l'expression devient supérieure à une constante.

Exemple : Division euclidienne par soustractions

L'expression entière est simplement r.

Elle décroît strictement si $b > 0$: $r' = r - b < r$.

La condition de la boucle devient fausse lorsque $r < b$.

Donc la boucle termine si $b > 0$.

Remarque : il faut s'assurer de la condition $b > 0$ à la lecture des données.

Exemple : puissance

L'expression entière est simplement n.

La condition de la boucle devient fausse lorsque $n \leq 0$.

L'expression décroît strictement si $n > 0$: $n' = n \operatorname{div} 2 < n$.

Donc la boucle termine.

Terminaison

prouver la terminaison n'est pas toujours facile.

(En fait, on peut montrer qu'il n'existe pas d'algorithme général permettant de déterminer si une boucle termine)

1) **(Conjecture de Syracuse)** On considère la fonction $f : \mathbb{N}^* \to \mathbb{N}^*$ définie par

$$f(n) = \begin{cases} n/2 & \text{si } n \text{ est pair} \\ 3n+1 & \text{sinon.} \end{cases}$$

On conjecture que si on itère cette fonction à partir d'un entier $n > 0$, on finit toujours par atteindre la valeur 1. Cette conjecture, vérifiée expérimentalement jusqu'à 32×10^{15}, n'a pas encore été démontrée.

On appelle *vol* la suite obtenue à partir d'un entier avant *d'atterrir* à la valeur 1. On appelle *durée du vol* le nombre d'itérations nécessaires pour atteindre la valeur 1 et *hauteur du vol* la valeur maximale obtenue durant le vol.

Écrire un programme qui lit un entier, puis calcule et affiche la durée et la hauteur de son vol.

Cette conjecture, dont l'origine vers 1950 reste confuse, porte une grande variété de noms : problème de Collatz, de Kakutani, d'Ulam, de l'algorithme de Hasse. Le nom de conjecture de Syracuse est lié à l'université de Syracuse aux États Unis où le problème fut étudié. S. Kakutani fit circuler le problème et raconte : « Pendant un mois, tout le monde à l'Université de Yale travailla dessus sans résultat. Un phénomène semblable se produisit à l'université de Chicago. Cette énigme, pensaient certains, avait été avancée par le KGB pour ralentir la recherche en mathématiques aux États-Unis. » (D'après J.-P. Delahaye).

Boucle Faire (Do)

Pseudo

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

C++

```
do {
    Liste d'instructions
} while (condition);
```

- On commence par effectuer la liste d'instructions.
- Puis on évalue la condition.
- Si elle est vraie, on recommence sinon on sort de la boucle.

Cela équivaut à

```
liste d'instructions
tant que condition faire
    liste d'instructions
finTantQue
```

Exemple

```
Faire
  Lecture de deux entiers
  Calcul et affichage du pgcd
  Afficher "Voulez-vous recommencer (O/N) ?"
  Lire réponse
Tant que réponse = 'O'
```

Terminaison :

On procède comme pour la boucle tant que.

Remarque : la boucle ci-dessus ne termine pas forcément.

Correction :

On se ramène à une boucle tant que et on utilise la méthode des invariants. Il faut donc

- Montrer que I est vérifié après la première exécution de la liste d'instructions.
- Montrer que si (C et I) est vrai, alors après une exécution de la liste d'instructions, I est encore vrai.
- On en déduit que (I et non C) est vrai à la sortie de la boucle (si la boucle termine).

Boucle Répéter

Pseudo

```
répéter
    liste d'instructions
jusqu'à condition
```

C++

```
do {
    Liste d'instructions
} while (! condition);
```

- On commence par effectuer la liste d'instructions.
- Puis on évalue la condition.
- Si elle est vraie, on sort de la boucle sinon on recommence.

Cela équivaut à

```
liste d'instructions
tant que non condition faire
    liste d'instructions
finTantQue
```


Exemple

```
répéter
  Afficher "Donner deux entiers positifs
            non tous deux nuls"
  lire m,n
jusqu'à (m>0 et n>=0) ou (m>=0 et n>0)
```

Terminaison :

On procède comme pour la boucle tant que.

Remarque : la boucle ci-dessus ne termine pas forcément.

Correction :

On se ramène à une boucle tant que et on utilise la méthode des invariants. Il faut donc

- Montrer que I est vérifié après la première exécution de la liste d'instructions.
- Montrer que si (non C et I) est vrai, alors après une exécution de la liste d'instructions, I est encore vrai.
- On en déduit que (I et C) est vrai à la sortie de la boucle (si la boucle termine).

Return et break dans une boucle

- `return` à tout endroit dans un programmefait sortir de la fonction en cours d'exécution
- `break`, déjà utilisé dans `switch` a pour effet de faire sortir de la boucle en cours

Ainsi, dans la séquence suivante

```
    .....
int i, tab[1000], val;
    .....
    ..... // initialiser tab et val
    .....
for(i = 0; i < 1000; i++){
    .....
    if (condition(tab[i], val))
        break;
    .....
    }
```

on sort de la boucle `for` pour la première valeur de i (si elle existe) telle que $tab[i]$ et val satisfont une $condition$ donnée

Exercices à faire en C/TD

On exécutera systématiquement les fonctions à la main sur des exemples et on donnera les invariants de boucles.

1) **(Somme des impairs)** Écrire une fonction qui calcule et renvoie comme valeur la somme des nombres impairs inférieurs ou égaux à un entier donné.

2) **(Moyenne des impairs)** Écrire une fonction qui, étant donné une suite de n nombres entiers, calcule la moyenne (sous forme de réel) de tous les entiers impairs qu'elle contient.

3) **(Nombre d'éléments pour atteindre une valeur)** Écrire une fonction qui, étant donné une suite de n nombres entiers et un nombre entier x, recherche le plus grand entier m tel que la somme des m premiers nombres de la suite soit strictement inférieure à x (et donc, ou bien $m = n$, ou bien la somme des $(m + 1)$ premiers nombres est supérieure ou égale à x). Si le nombre x est nul, la fonction renverra 0.

4) **(Conversion binaire/décimal)** Écrire une fonction qui, étant donné une suite de n caractères ($n < 32$) supposée représenter un nombre entier sous forme binaire (c'est-à-dire ne contenant que des caractères '0' ou '1') calcule si cela est possible le nombre décimal correspondant et le renvoie en valeur de retour. Si un caractère incorrect (ni '0', ni '1') est rencontré ou si n n'est pas un entier positif inférieur à 32, la fonction renverra la valeur -1.

5) **($\lceil \log_2 x \rceil$)** Écrire un programme qui, étant donné un nombre $x \geq 1$, affiche le plus petit entier $n \geq 0$ tel que $x \leq 2^n$.

6) **Tri d'une suite de nombres** Il s'agit de trier une suite de nombres entiers dans l'ordre croissant.
On se propose de procéder comme suit :

- modifier, afin de pouvoir la réutiliser pour résoudre le problème, la fonction `maximum` afin qu'elle renvoie l'indice du plus grand élément d'une suite de n nombres
- appeler cette suite itérativement sur une suite de n, $n-1$, 2 éléments et au retour de chaque appel (donc pour une suite de longueur i) échanger l'élément en position i avec le plus grand de la suite des i premiers.

A la fin la suite est triée

Exercices à faire en TD/TP

En plus des fonctions dont l'écriture est demandée, on écrira des programmes (fonctions main) appelant ces fonctions

7) **(Factorielle)** Écrire un programme qui lit un entier $n \geq 1$, puis calcule et affiche $n! = 1 \times 2 \times \cdots \times n$.

8) **(Fibonacci)** Écrire un programme qui lit un entier $n \geq 1$, puis calcule et affiche $F(n)$, le $n$ième nombre de Fibonacci.
Les nombres de Fibonacci sont définis par

- $F(0) = F(1) = 1$,
- Si $n \geq 2$ alors $F(n) = F(n-1) + F(n-2)$.

9) **(π)** On définit la suite double :

$$\begin{cases} u_0 = 1, \\ v_0 = 2 \end{cases} \quad \text{et} \quad \begin{cases} u_{n+1} = (u_n + v_n)/2, \\ v_{n+1} = \sqrt{u_{n+1} v_n} \end{cases}$$

On suppose acquis que ces deux suites sont adjacentes, u_n croissante et v_n décroissante, leur limite étant $\sqrt{27}/\pi$. Écrire un programme qui calcule la n-ième itération de v_n, et affiche l'approximation de π correspondante.

10) **(Zéro par dichotomie)** On cherche à approximer un zéro d'un polynôme unitaire à coefficients réels :

$$P(x) = x^3 + ax^2 + bx + c$$

On montrera que si $m = 2 * max(|a|, \sqrt{|b|}, \sqrt[3]{|c|})$ alors

$$P(-m) \leq 0 \leq P(m).$$

Écrire une fonction qui raffine par dichotomie jusqu'à une certaine précision l'intervalle $[-m, m]$ de telle façon qu'il contienne toujours un zéro.

11) **(Écriture en binaire)** Écrire une fonction qui, étant donné un entier positif x calcule sa représenation binaire. Cette représentation sera construire sous forme d'une suite de n caractères '0' et '1'.

Corrigés

1) Somme des impairs

```
 Fonction SommeDesImpairs(n : entier)
                    résultat entier
     Variables s, i : entier
   Début
     s <- 0 ; i <- 1
     tant que i <= n faire
         s <- s + i ; i <- i + 2
     finTantQue
     retourner s
   Fin
```

Traduction en un programme

```
#include <iostream.h>
int sommeDesImpairs(int n) {
   int s = 0 , i = 1;
   while (i <= n) {
        s = s + i; i = i + 2;  // i += 2
        }
   return s;
}
int main ( ) {
   int n;
   cout << "valeur de n ? "; cin >> n;
   cout << sommeDesImpairs(n) << endl;
   return 0;
}
```


2) Moyenne des impairs

```
 Fonction MoyenneDesImpairs(
          n : entier,
          a : suite de n entiers)
                résultat réel
   /* i parcourt, s somme, m  compte */
   variables s, i, m : entiers
   Début
     s <- 0
     i <- 1
     m <- 0
     tant que i <= n faire
         si (a[i] est impair) alors
             s <- s + a[i]
             m <- m + 1
         finSi
         i <- i + 1
     finTantQue
     retourner s / m
   Fin
```


Traduction en un programme

```
#include <iostream.h>
float moyenneDesImpairs(int n, int a[ ]) {
   int s, i, m;
   i = m = s = 0;
   while (i < n) {
        if ((a[i] % 2) == 1) {
            s = s + a[i];
            m++; }
        i++;
        }
   // DIVISION ENTIERE vs DIVISION
   return (float)s / m;
}
int main ( ) {
   int t[8] = {1, 8, 6, 7, 9, 11 ,23 ,4};
   cout << moyenneDesImpairs(8, t) << endl;
   return 0;
}
```


3) Nombre d'éléments pour atteindre une valeur

Le problème est un peu plus difficile que ceux que nous avons considérés jusqu'ici : pour obtenir le résultat il est nécessaire d'anticiper.

Dans l'exemple considéré, lorque la somme des éléments considérés est supérieure à x, il est trop tard.

Ce qu'il faut donc éviter : accéder à un élément en dehors des limites autorisés pour les index

Dans une solution itérative, on devra avoir en fin de boucle :
- ou bien $i = n + 1$ **et** $(\sum_{i=1}^{n} a_i) < x$
- ou bien $i \leq n$ **et** $(\sum_{j=1}^{i-1} a_j) < x$ **et** $(\sum_{j=1}^{i} a_j) \geq x$

On en déduit la solution suivante :

```
Fonction sommePourAtteindre(
         x, n : entiers
         a : suite de n entiers)
                   résultat entier
  variables s, m : entiers
            continuer : booléen
  Début
    si (x = 0) ou (n = 0) retourner 0
    s <- 0 ; m <- 1
    continuer <- vrai
    tant que (continuer) faire
        si (s + a[m]) >= x alors
           continuer <- faux
        else
           s <- s + a[m];
           si m = n alors continuer <- faux
           sinon m <- m + 1;
           finSi
        continuer <- faux
        finSi
    finTantQue
    retourner m
    finSi
  Fin
```

L'invariant sous-jacent est :

$i \leq n$ **et** $(s = \sum_{j=1}^{i-1} a_j)$ **et** $s < x$

Traduction en un programme

La traduction prend en compte le fait que dans le langage utilisé les index commencent à 0

```
#include <iostream.h>
int sommePourAtteindre(int x, int n, int a[ ]) {
   int s, m;
   bool continuer;
   if (n == 0) return 0;
   s = 0; m = 0; continuer = true;
   while (continuer == true) {
        if ((s + a[m]) >= x) continuer = false;
        else {
            s = s + a[m];
            if (m ==  n) continuer = false;
            else m = m + 1;
            }
        }
   return m;
}
int main ( ) {
   int tab[5] =  {3, 7, 4, 8, 10};
   int x;
   cout << "valeur de x ? "; cin >> x;
   cout << sommePourAtteindre(x, 5, tab) << endl;
   return 0;
}
```


4) Conversion binaire/décimal

La suite est $c_1, c_2, \ldots, c_n$ qu'on doit convertir en la valeur décimale correspondante, c'est-à-dire
$c_1 \times 2^{n-1} + c_2 \times 2^{n-2} + \ldots + c_{n-1} \times 2 + c_n$

On peut procéder de différentes manières :

- soit parcourir la liste de caractères de gauche à droite (itération de 1 à n) et pour chaque valeur i calculer 2^{n-i}, puis $c_i \times 2^{n-i}$ et cumuler les résultats
- soit parcourir la liste de caractères de droite à gauche afin de calculer les puissances de 2 de manière croissante, ce qui permettra d'utiliser la valeur de 2^{i-1} pour calculer celle de 2^i
- soit parcourir la liste de caractères de droite à gauche et utiliser une technique de calcul (dite schéma de Horner) correspondant au parenthésage suivant :

  $(\ldots(((c_1 \times 2 + c_2) \times 2 + c_3) \times 2 + \ldots c_{n-1} \times 2) + c_n$

Le programme suivant est une solution à la deuxième approche dans laquelle le cacule de 2^i se fait en multipliant par 2 le résultat obtenu pour 2^{i-1} à l'itération précedente.
On utilise ici un pas de boucle égale à -1.
Nous donnons ci-dessous la version C-- de la fonction de conversion.

```
include <iostream.h>
int binaireDecimal(char c[ ], int n){
   int i,      // pour parcourir les chiffres binaire
       val = 0,  //pour le cacul du nombre
       deuxPuis = 1;  // les puissances successives d
   if (n <= 0 || n >= 32)
      return -1;  // erreur sur n, on renvoie -1
   for(i = n - 1; i >= 0; i--){
      if (c[i] != '0' && c[i] != '1')
         return -1; // erreur : on renvoie -1
      else
         if(c[i] == '1')
            val += deuxPuis;
      deuxPuis *= 2; // deuxPuis = 2 * deuxPuis
      }
   return val;
}
```

Le code suivant est celui d'un programme principal qui commence par lire les chiffres du nombre et les mémorise dans un tableau de caractères.
La lecture s'arrête
– soit lorsqu'on a lu 32 caractères '0' ou '1',
– soit lorsqu'on lit un caractère différent de '0' et '1'.
Le programme appelle ensuite la fonction précédente dont il affiche la valeur de retour :

```
int main( ){
char c[31];
bool continuer = true;
int chiffre, i, n;
for(n = 0; n < 31; n++) {
   cin >> chiffre;
   switch (chiffre){
      case 0: c[n] = '0'; break;
      case 1: c[n] = '1'; break;
      default: continuer = false;
      }
   if (! continuer) break;  // SORTIE DE BOUCLE
   }
for(i = 0; i < n; i++)
   cout << c[i];
cout << " = " << binaireDecimal(c, n) << endl;
return 0;
}
```


On peut observer sur la fonction `binaireDecimal` et le programme, la possibilité de sortir d'une boucle de manière impérative (autrement que par le test contrôlant l'itération) par exécution de certaines instructions dans le corps de la boucle
– `return` fait sortir non seulement de la boucle, mais aussi de la fonction en renvoyant une valeur
– `break` fait simplement sortir de la boucle
Un exemple d'exécution du programme (supposé s'appeler `conversion`) est donné ci-après :

```
--> conversion
1 1 1 0 0 1 0 1 2
11100101 = 229
```


La fonction ci-dessous est une description en pseudo-langage de la fonction avec la méthode de Horner :

```
 Fonction Horner(
          n : entier,
          c : liste de n caractères)
                              résultat entier
   variables val, i: entier
   Début
      si (n <= 0) ou (n >= 32) retourner -1
      val <- 0;
      pour i = 1 à n faire
          si c[i] est différent de '0' et de '1'
             alors retourner -1
             sinon  val <- 2 * val
                    si c[i] = '1'
                         alors val <- val + 1
                    finSi
          finSi
      finTantQue
      retourner val
   Fin
```


Traduction dans le langage C--

```
#include <iostream.h>
int horner(char c[ ], int n){
   int i, val = 0;
   if (n <= 0 || n >= 32) return -1;
   for(i = 0; i < n; i++)
      if (c[i] != '0' && c[i] != '1')
         return -1; // erreur: on renvoie -1
      else { val = 2 * val;
             if (c[i] == '1') val ++;
      }
   return val;
}
```

Un programme principal (fonction `main`) utilisant cette fonction est obtenu en remplaçant celui de la version précédente l'appel à `binaireDecimal` par un appel à `horner`.

Il sera intéressant d'exécuter ce programme à la main sur l'exemple d'exécution donné pour la version précédente.

5) ($\lceil \log_2 x \rceil$)

```
Fonction PartEntLog2 (x: entier)
           résultat entier
  variables n, p:entier
Début
  p<-1
  n<-0
  tant que p < x faire
      p = 2 * p
      n = n + 1
  finTantQue
  retourner n
Fin
```

Invariant : $p = 2^n$ et $p < 2x$.

Traduction en un programme

```
#include <iostream.h>
int PartEntLog2(int x) {
    int p=1,n=0;
    while (p < x) {
         p = 2 * p ;
         n = n + 1 ;
        } // end while
    return n;
}
```



```
int main () {
  int x;
  cout << "Calcul de la partie entière " << endl;
  cout << "supérieure du logarithme en " << endl;
  cout << "base 2 d'un entier x" << endl;
  do {
    cout << "Donner l'entier x > 0 :" << endl;
    cin >> x;
  } while (x < 1);
   cout << "Résultat : " << PartEntLog2(x) << endl;
  return 0;
}
```

Sur ma machine, ce programme écrit 10 si $x = 555$, 30 si $x = 1073741824$, et ne s'arrête jamais si $x = 1073741825$! ! !

6) Tri d'une suite de nombres entiers

Nous donnons le code en langage C-- de la fonction maximum modifiée et d'un programme l'utilisant pour faire le tri en ordre croissant d'une suite de nombres mémorisés dans un tableau.

On notera en particulier comment on réalise l'échange des valeurs de deux variables en en utilisant une troisième

```
#include <iostream.h>
int maximum(int a[ ], int n) {
   int i, x, indmax;
   indmax = 0; x = a[0];
   for(i = 1; i < n; i++)
        if (x < a[i]){ x = a[i]; indmax = i;}
   return indmax;
}
int main ( ) {
   int t[5], i, ind, n = 5, x;
   for (i = 0; i< 5; i++)
      cin >> t[i];
   for(i = n ; i > 1 ; i--){
       ind = maximum(t, i);
       /* échange de t[i] et t[ind] en passant par x
       x = t[ind] ; t[ind] = t[i-1] ; t[i-1] = x;
       }
   for(i = 0; i < n; i++) cout << t[i] << " ";
   cout << endl;
   return 0;
}
```


7) (Factorielle)

```
Fonction Factorielle (n : entier)
           résultat entier
  variables i,F : entier
Début
  F <- 1
  pour i <- 2 à n faire
      F <- F * i
  finPour
  retourner F
Fin
```

Invariant : $F = 1 \times 2 \times \cdots \times (i-1)$ et $i \leq n+1$.

Faire la preuve.

Traduction en un programme

```
#include <iostream.h>
int Factorielle(int n) {
    int i, F = 1;
    for(i = 2;i <= n; i++) F = F * i;
    return F;
}
int main ( ) {
    int n;
    do {
        cout <<"Donner un entier strictement "<<endl;
        cout << "positif " << endl;
        cin >> n;
    } while (n <= 0);
    cout << "Résultat : " << Factorielle(n) << endl;
    return 0;
}
```

Sur ma machine, ce programme écrit 10 si $x = 555$, 30 si $x = 1073741824$, et ne s'arrête jamais si $x = 1073741825$! ! !

8) (Fibonacci)

```
Fonction Fibonacci(n : entier)
        résultat entier
 variables i,a,b : entier
Début

 a <- 1; b <- 1

 pour i <- 1 à n-1 faire
   a <- a + b
   b <- a - b
 finPour

 retourner a
Fin
```

Invariant : $a = F(i)$ et $b = F(i-1)$ et $1 \leq i \leq n$.

Faire la preuve.

Remarque : $a' = a + b$ et $b' = a' - b = a$.

9) (π)

```
Fonction Pi (n : entier)
        résultat réel

   variables u : réel initialisé à 1
             v : réel initialisé à 2
             pi : réel
             i,n : entier
Début Pi
   pour i <- 1 à n faire
     u <- (u+v)/2
     v <- racine(u*v)
   finPour

   retourner  racine(27)/v

Fin
```

L'invariant consiste à vérifier que u et v collent bien à la définition de la suite.

Invariant : $u = u_{i-1}$ et $v = v_{i-1}$.

Traduction en un programme

```
#include <iostream.h>
double sqrt(double);
double Pi(int n) {
  double u=1,v=2;
  int i;
  for(i=1;i<=n;i++){
    u=(u+v) /2;
    v=sqrt(u*v);
  }

  return sqrt(27)/v;
}
int main () {
  int n;
  do {
    cout << "Donner un entier strictement "<<endl;
    cout << "positif représentant la "<<endl;
    cout << "profondeur du calcul"<<endl; cin >> n;
  } while (n <=0);
  cout << "Une approximation de Pi" << endl;
  cout << "est : " << Pi(n) << endl;
  return 0;
}
```

10) (Zéro par dichotomie)

```
Fonction Zéro(a, b, c, e : réels)
        résultat couple de réels
            //a,b,c : réels (les coefficients)
            //e : réel (pour la précision)
  variables m, M : réels (pour l'intervalle)
            x,y : réels
Début
```

$M \leftarrow 2 * max(|a|, \sqrt{|b|}, \sqrt[3]{|c|})$

```
  m <- -M
  tant que M-m > e faire
    x <- (M+m)/2
```

$y \leftarrow x^3 + ax^2 + bx + c$

```
    si y >= 0 alors M <- x sinon m <- x finSi
  finTantQue
  retourner le couple (m,M)
Fin
```


Traduction en un programme

```
#include <iostream.h>
#include <math.h>
double sqrt(double);
double cbrt(double);
double abs(double);

struct intervalle{
  double init;
  double fin;
};

//Maximum de abs(a), sqrt(abs(b)) et de sqrt(3,abs(c)
double Max3 (double  a, double b, double c) {
  double xa,xb,xc;
  xa=abs(a);
  xb=sqrt(abs(b));
  xc=cbrt(abs(c));
  if ((xa>=xb) && (xa >=xc)) return(xa);
  if ((xb>=xa) && (xb >=xc)) return(xb);
  if ((xc>=xb) && (xc >=xa)) return(xc);
}

intervalle Zero(double a,double b,double c,double e)
  //a,b,c : réels (les coefficients)
  // e : réel (pour la précision)
  double m, M, x, y; //m, M : réel (pour l'intervalle
  intervalle I;
```



```
  M = 2 * Max3(a, b, c);
  m = -M;
  while(M-m > e) {
    x = (M+m)/2;
    y = x*x*x + a*x*x + b*x + c;
    if (y > 0)
      M = x;
    else
      m = x;
  }
  I.init = m; I.fin = M;
  return I;
}
int main () {
  double a,b,c;//a,b,c : réels (les coefficients)
  double e;   // e : réel (pour la précision)
  intervalle I;
  cout << "Entrer les coefficients a,b,c"<<endl;
  cin >>a >>b >>c;
  do {
    cout << "Entrer la précision >0"<<endl;
    cin >> e;
  } while (e <=0);
  I=Zero(a,b,c,e);
  cout << "il y a un zéro entre "<< I.init<< endl;
  cout << " et "<< I.fin << endl;
  return 0;
}
```


11) (Écriture en binaire)

On note + la concaténation des chaînes de caractères.

```
Fonction Binaire(n : entier)
                résultat chaîne de caractères
 variables bin : chaîne de caractères;
Début
  bin <- ""  // chaîne vide
  tant que n > 0 faire
       si n est pair alors
         bin <- "0" + bin // concaténation de chaînes
       sinon
         bin <- "1" + bin
       finSi
  n <- n div 2
  finTantQue
   retourner bin
Fin
```