4. Tests `if`

En pratique on n'écrira presque jamais un programme sans faire des tests et prendre des décisions basées sur les résultats. Cela se fait avec la commande `if`. Syntaxe générale :

```
if (test) {
  bloc de commandes à exécuter si test est vrai;
}
else {
  bloc de commandes à exécuter si test est faux;
}
```

La partie `else` est optionnelle et peut être enlevée. Si un bloc ne contient qu'une seule commande *et seulement dans ce cas*, les accolades sont optionnelles. Les parenthèses autour du test sont toujours obligatoires.

Pour faire des tests il existe les opérateurs suivants :
$==, <, >, <=, >=, !=$ (inégal à), ainsi que && (et), || (ou), ! (négation).
Précédence des opérateurs : 1) !, 2) $*, /, \%$, 3) $+, -$, 4) $<, >, <=, >=$, 5) $==, !=$, 6) &&, 7) ||, 8) $=$. Utiliser des parenthèses pour changer.

Faites très attention à la différence entre `a == b` (teste si a est égal à b, résultat est 1 (vrai) ou 0 (faux)) et `a = b` (rend a égal à b, résultat est valeur de b, et chaque valeur différente de 0 est considérée comme vraie par C) !

Exemples :

- ```
  if (i == 5) {
    j = 0;
    cout << "La variable i est egale a 5" << endl;
  }
  else {
    j = 1;
    cout << "La variable i n'est pas egale a 5 mais a " << i << endl;
  }
  ```
- ```
  if ((i <= 5 || i > 10) && j != 1 &&!(k == 2*i && j == i))
    k = i + j;
  ```
- ```
  if (i < 0)
    cout << "i est negatif" << endl;
  else if (i == 0)
    cout << "i est zero" << endl;
  else
    cout << "i est positif" << endl;
  ```
  ```

Boucles `for`

Un autre élément très commun dans des programmes est de répéter une opération dans une boucle un certain nombre de fois jusqu'à ce qu'une condition soit satisfaite. La commande `for` est utilisée pour des boucles avec compteur. Syntaxe générale :

```
for (initialisation; condition; commande d'itération) {
  bloc de commandes à répéter;
}
```

Exemple (accolades optionnelles si bloc contient une seule commande) :

```
int i;
for (i = 0; i < 5; i++)
  cout << i << endl;
```

Dans cet exemple la variable `i` s'appelle le compteur. L'initialisation du compteur (ici `i = 0`) est exécutée une seule fois tout au début de la boucle. Puis la condition (ici `i < 5`) est testée, si vraie le bloc de commandes (ici juste `cout << i << endl;`) est exécuté (ces commandes peuvent dépendre du compteur, comme ici, mais ce n'est pas nécessaire), sinon la boucle est terminée et le programme passe à ce qui suit. Si le bloc a été exécuté, la commande d'itération (ici `i++` qui augmente `i` par 1) est ensuite exécutée, le programme repasse au test de la condition, etc.

Autres exemples :

```
for (i = 10; i > 4; i -= 2) cout << i << " "; // Affiche 10 8 6
for (i = 10; i < 10; i++) cout << "allo" << endl; // Affiche rien
```

Exemple programme complet pour calculer les premières puissances de 2 :

```
#include<iostream>
using namespace std;
int main() {
  int i, i2, n;
  n = 11;    // Nombre de puissances a calculer (en commencant par 0)
  i2 = 1;
  for(i = 0; i < n; i++) {
    cout << "2 puissance " << i << " = " << i2 << endl;
    i2 *= 2;
  }
  return 0;
}
```

Le programme affiche :
2 puissance 0 = 1
2 puissance 1 = 2
...
2 puissance 10 = 1024

Boucles `while`, `do while`

Si le nombre de répétitions est inconnu a priori, une boucle sans compteur convient mieux : `while` ou `do while`. Syntaxe générale :

```
while (condition) {
  bloc de commandes à répéter;
}
```

```
do {
  bloc de commandes à répéter;
} while (condition);
```

Tandis que la condition est vraie, le bloc de commandes est exécuté. La différence entre `while` et `do while` est que `while` teste la condition avant d'exécuter le bloc et `do while` après. Dans une boucle `do while` le bloc de commandes est donc toujours exécuté au moins une fois.

Remarque bien que, contrairement à la boucle `for`, il n'y a pas de compteur qui change automatiquement. Il faut donc que les commandes dans le bloc changent explicitement le résultat du test à un certain moment, sinon la boucle ne s'arrête jamais.
Exemple :

```
int x = 2;
while (x < 200) {
  cout << x << " ";
  if (x % 3 == 0) x = 2*x + 1;
  else x = x + 7;
}    // Le programme affiche 2 9 19 26 33 67 74 81 163 170 177
```

Exemple programme complet pour résoudre l'équation $x = \log(x+2)$:

```
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  double x, eps, xa, diff;
  x = -1.75;      // Valeur d'essai initiale
  eps = 1.e-5;    // Precision demandee du resultat
  do {
    xa = x;
    x = log(x+2.);
    diff = fabs(x-xa);
  } while(diff >= eps);
  cout << "Racine = " << x << endl;
  return 0;
}    // Le programme affiche : Racine = 1.14619
```

On voit ici l'utilité de la boucle `do while` : au début du premier tour la quantité `diff` sur laquelle est effectué le test n'est pas connue. Avec une boucle `while` il faudrait l'initialiser à une valeur artificielle (par exemple `2*eps`).

Interrompre une boucle : `break`

Les boucles `for`, `while` et `do while` peuvent être interrompues avec la commande `break`, qui fait sortir directement de la boucle. [Il existe aussi la commande `continue` qui fait passer directement à l'itération suivante de la boucle.] Exemple :

```
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  int i, s = 1, n = 1000;
  double x = 0., eps = 1.e-3;
  for (i = 1; i <= n; i += 2) {
    x = x + 4. * s / i;
    s = -s;
    cout << i << " : " << x << endl;
    if (fabs((x-M_PI)/M_PI) < eps)
      break;
  }
  return 0;
}    // Derniere ligne affichee : 637 : 3.14473
```

Ce programme calcule π en utilisant la formule $\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$. Le programme s'arrête quand la différence relative avec la vraie valeur de π est moins que `eps` ou si la valeur dans le dénominateur devient supérieure à `n`.

Erreurs fréquentes

Essaie toujours de vérifier d'abord que tu n'as pas fait l'une de ces erreurs avant d'appeler un enseignant en TD !

- Faire une division d'entiers inattendue : $1/2 = 0$.
- Utiliser $=$ au lieu de $==$ dans un `if` (ou autre condition).
- Faute de copier-coller : `for (j = 0; j < 10; i++)`
- Utilisation d'une variable déclarée mais non initialisée.
- Utiliser une virgule au lieu d'un point comme séparateur décimal.
- Oubli d'un $<<$ quelque part dans une commande `cout`.
- Oubli d'accolades autour de deux (ou plus de) commandes dans un bloc, ce qui fait que la deuxième commande est exécutée *après* et non pas *dans* la boucle.