5. Tests et boucles

Dans les programmes vus jusqu'ici, les instructions sont toutes exécutées une fois et une seule et ce, dans l'ordre où elles sont écrites dans le fichier. Les tests permettent de n'exécuter qu'une partie des instructions, en fonction des valeurs que prennent certaines variables, et les boucles de répéter certaines instructions.

1 Tests

1.1 Alternative `if`

On rappelle que des instructions forment un bloc si elles sont comprises entre deux accolades.
L'instruction `if` permet de faire exécuter un bloc d'instructions si et seulement si une certaine condition est réalisée et un autre bloc si et seulement si elle ne l'est pas.

1.1.1 Exemple

On veut calculer la fonction $\sin x/x$. Il faut faire un cas particulier pour $x = 0$. On écrit donc :

```
...
double x, f;
...
x = ...;
if(x == 0.) {
  f = 1.;
}
else {
  f = sin(x)/x;
}
...
```

Dans cet exemple particulier les blocs d'instructions à effectuer dans l'un ou l'autre cas se réduisent chacun à une seule instruction.

1.1.2 Forme générale

```
if(expression) {
  instructions;
}
else {
  instructions;
}
```

Si `expression` est vraie (donc vaut 1), le premier bloc d'instructions est exécuté puis le second est sauté et le cours normal du programme reprend à la première instruction qui suit le second bloc.
Si `expression` est fausse (donc vaut 0), le premier bloc est sauté, le second est exécuté puis, le cours normal du programme continue à la première instruction qui suit le second bloc.

Remarques

1) Si un des blocs ne contient qu'une instruction on n'est pas obligé de mettre les `{ }`
2) S'il n'y a pas d'instructions à exécuter lorsque `expression` est fausse on peut ne pas mettre le `else`
3) Il est possible d'inclure un second `if` dans un des blocs d'instructions, d'en inclure un troisième dans le second et ainsi de suite sans limitation. Il suffit de respecter la règle d'inclusion : un `else` se rapporte toujours au dernier `if` rencontré auquel un `else` n'a pas encore été attribué.

Exemples

1) Conformément à la remarque 1) le `if` du programme qui calcule $\sin x/x$ peut être écrit sans accolades :

```
if(x == 0.)
  f = 1.;
else
  f = sin(x)/x;
```

2) Pour illustrer la remarque 2) on calcule la fonction d'Heaviside :

$$\begin{aligned} H(x) &= 0 \quad \text{pour } x < 0 \\ H(x) &= 1 \quad \text{pour } x \geq 0 \end{aligned}$$

On peut l'écrire de la façon suivante :

```
...
if(x < 0.)
  H = 0.;
else
  H = 1.;
...
```

Mais aussi, ce qui peut être plus simple dans certains cas :

```
...
H = 0.
...
if(x >= 0.)
  H = 1.;
...
```

3) Pour illustrer la remarque 3) on calcule le point d'intersection de deux droites dans le plan, d'équations :

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

Il faut, avant d'écrire le programme, examiner les différents cas possibles :

$a = 0$ et $b = 0$ ou $d = 0$ et $e = 0$	une des deux droites n'est pas definie	
$ae - db = 0$	$af - dc = 0$	les droites sont confondues
	$af - dc \neq 0$	les droites ne se coupent pas
$ae - db \neq 0$	les deux droites se coupent	

Ce qui peut s'écrire :

```
#include<iostream>
using namespace std;
int main() {
  double a, b, c, d, e, f, det, det1, det2;
  a = ...; b = ...; c = ...; d = ...; e = ...; f = ...;
  if(a == 0. && b == 0. || d == 0. && e == 0.)
    cout << "Au moins une des deux droites n'est pas definie" << endl;
  else {
    det = a*e-d*b;
    det1 = a*f-d*c;
    if(det == 0.) {
      if(det1 == 0.)
        cout << "Les deux droites sont confondues" << endl;
      else
        cout << "Les deux droites ne se coupent pas" << endl;
    }
    else {
```

```
        det2 = b*f-e*c;
        cout << "Solution unique : x=" << det2/det << "  y=" << det1/det << endl;
      }
    }
    return 0;
  }
```

On remarque que, lorsque comme ici, la structure devient un peu compliquée, il faut indenter [1] avec soin pour que le programme soit lisible.

1.1.3 Choix multiple

On considère par exemple trois intervalles disjoints sur l'axe des réels : $[x_1^{\min}, x_1^{\max}]$, $[x_2^{\min}, x_2^{\max}]$, $[x_3^{\min}, x_3^{\max}]$. Etant donné une variable réelle `x`, on veut qu'une variable entière `i` vaille :

1 si `x` appartient à l'intervalle $[x_1^{\min}, x_1^{\max}]$
2 si `x` appartient à l'intervalle $[x_2^{\min}, x_2^{\max}]$
3 si `x` appartient à l'intervalle $[x_3^{\min}, x_3^{\max}]$
0 sinon.

Si on applique strictement ce qui précède on doit écrire :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, x1min = -1., x2min = 0.5, x3min = 2.;
  double x1max = -0.5, x2max = 0.8, x3max = 2.9;
  x = 0.7;
  if(x >= x1min && x <= x1max)
    i = 1;
  else {
    if(x >= x2min && x <= x2max)
      i=2;
    else {
      if(x>=x3min && x<=x3max)
        i=3;
      else
        i=0;
    }
  }
  cout << "i=" << i << endl;
  return 0;
}
```

Mais en réalité les accolades pour imbriquer les `if ... else` les uns dans les autres ne sont pas nécessaires et on peut écrire :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  double x, x1min = -1., x2min = 0.5, x3min = 2.;
  double x1max = -0.5, x2max = 0.8, x3max = 2.9;
  x = 0.7;
  if(x >= x1min && x <= x1max)
    i=1;
  else if(x >= x2min && x <= x2max)
```

1. utiliser l'indentation automatique de *emacs*

```
    i=2;
  else if(x >= x3min && x <= x3max)
    i=3;
  else
    i=0;
  cout << "i=" << i << endl;
  return 0;
}
```

Remarque

S'il y a plusieurs instructions à exécuter dans le cas où un `if` ou un `else` est réalisé [2], alors il est évidemment nécessaire de mettre ces instructions entre accolades, aussi bien dans le premier cas que dans le second.

1.2 Aiguillage `switch`

Dans le cas du choix multiple le `switch` peut être plus pratique que le `if` mais n'est pas toujours utilisable. Supposons qu'on doive faire exécuter le groupe d'instructions (1) si la variable entière `n` vaut 1, le groupe d'instructions (2) si elle vaut 2, le groupe d'instructions (3) si elle vaut 3 et le groupe d'instructions (4) sinon. Avec un `if` il faut écrire :

```
...
if(n == 1) {
  instructions_1;
}
else if(n == 2) {
  instructions_2;
}
else if(n == 3) {
  instructions_3;
}
else {
  instructions_4;
}
...
```

Avec un `switch` cela s'écrit :

```
...
switch(n) {
case 1 :
  instructions_1;
  break;
case 2 :
  instructions_2;
  break;
case 3 :
  instructions_3;
  break;
default :
  instructions_4;
}
...
```

La forme générale du `switch` est la suivante :

2. au lieu d'une seule comme ici : `i = ...`

```
switch(expression) {
case constante_1 :
  instructions_1;
  break;
case constante_2 :
  instructions_2;
  break;
...
default :
  instructions;
}
```

Deux cas se présentent :

- `expression` vaut une des `constante_i` : alors les `instructions_i` sont exécutées. Le `break` n'est pas obligatoire : si `instructions_i` se termine par un `break` on passe ensuite après l'accolade de fin du bloc et le `switch` est terminé, sinon ce sont les `instructions_(i+1)` qui sont exécutées et ainsi de suite jusqu'à rencontrer un `break`
- `expression` ne vaut aucune des `constante_i` : ce sont les `instructions` situées après `default` qui sont exécutées, puis on sort du `switch`.

Remarques

`expression` doit être de type `int` ou `char`. C'est pourquoi l'exemple précédent des trois intervalles ne peut pas être traité par un `switch`, sinon d'une façon qui n'apporte pas de simplification par rapport à un `if`.

Les `constante_i` peuvent être explicites ou non.

Certaines `instructions_i` peuvent être vides (y compris de `break`). Ceci permet d'avoir plusieurs `constante_i` conduisant aux mêmes `instructions_j`.

Exemple 1 :

```
...
char rep;
...
cout << "Doit-on calculer le perimetre (p), la surface (s) ou le volume (v) ?" << endl;
cin >> rep; // met dans la variable caractère rep la réponse frappée au clavier
switch(rep) {
case 'p':
  ...
  break;
case 's':
  ...
  break;
case 'v':
  ...
  break;
default :
  ...
}
```

Exemple 2 :

Soit deux disques, l'un centré au point de coordonnées a_1, b_1 et de rayon r_1, l'autre centré en a_2, b_2, de rayon r_2. On veut déterminer si un point de coordonnées x, y appartient aux deux disques, à l'un des deux seulement ou à aucun des deux.

```
/* Position d'un point par rapport a deux disques */
#include<iostream>
using namespace std;
int main() {
```

```
  double a1 = -3., b1 = 5., r1 = 2., a2 = 1.5, b2 = 4., r2 = 5.;
  double x = 0., y = 10., d1, d2;
  d1 = (x-a1)*(x-a1) + (y-b1)*(y-b1);
  d2 = (x-a2)*(x-a2) + (y-b2)*(y-b2);
  switch((d1 <= r1*r1) + (d2 <= r2*r2)) {
  case 0 :
    cout << "Le point est exterieur aux deux disques" << endl;
    break;
  case 1 :
    cout << "Le point est interieur a l'un des disques et exterieur a l'autre" << endl;
    break;
  case 2 :
    cout << "Le point est interieur aux deux disques" << endl;
    break;
  }
  return 0;
}
```

Dans ce cas il est inutile de mettre un cas `default` puisque la valeur de l'expression ne peut jamais être différente de 0, 1 ou 2.

2 Boucles

2.1 Boucle `while`

2.1.1 Exemple

Supposons qu'on veuille calculer les puissances successives de 2. Dès que l'on veut en calculer plus de trois il serait lourd d'écrire :

```
...
x = 1;   cout << x << endl;
x = x*2; cout << x << endl;
x = x*2; cout << x << endl;
...
```

On utilise alors une boucle. Par exemple le `while` :

```
...
x = 1;
while(x <= 1000) {
  x = x*2; cout << x << endl;
}
...
```

calcule les puissances de 2 inférieures ou égales à 2000. Tant que `x` est inférieur ou égal à 1000 on effectue les instructions du bloc. Sinon on passe après le bloc.

2.1.2 Forme générale

```
while(expression) {
  instructions;
}
```

La première fois que le `while` est rencontré `expression` est évaluée :

— si elle est fausse on passe directement après le bloc sans être entré une seule fois dans la boucle

— si elle est vraie les instructions du bloc sont exécutées puis on revient au `while` où l'expression est de nouveau évaluée. Tant qu'elle reste vraie on ré-exécute le bloc jusqu'à ce qu'`expression` devienne fausse. Si elle ne devient jamais fausse, la boucle est exécutée indéfiniment et le programme ne s'arrête jamais[3], sauf si on utilise une sortie de boucle à l'aide de l'instruction `break`, présentée à la section suivante.

2.1.3 `break` et `continue`

Les instructions `break` et `continue` se placent nécessairement dans une boucle :

`break` fait sortir définitivement de la boucle et passer à l'instruction qui la suit

`continue` interrompt le tour en cours et fait passer au tour suivant

En cas de boucles imbriquées `break` et `continue` concernent la boucle la plus interne les contenant.

Remarque

`break` et `continue` sont utilisables non seulement dans la boucle `while` mais aussi dans toutes celles vues dans la suite.

2.1.4 Exemple de boucle infinie

La boucle :

```
while(1) {
...
}
```

ne s'arrête, à priori, jamais, puisque l'expression testée par le `while` vaut toujours 1 et est donc toujours vraie. En ajoutant un `break` elle peut être utilisée comme dans l'exemple suivant :

```
/* Calcul du carre d'un nombre */
#include<iostream>
using namespace std;
int main() {
  int n;
  cout << "Répondre 0 pour terminer l'exécution" << endl;
  while(1) {
    cout << "n=? "; cin >> n;
    if(n == 0)
      break;
    cout << n << " au carré = " << n*n << endl;
  }
  return 0;
}
```

2.1.5 Exemple de boucle utilisant un `continue`

On veut faire la liste des nombres inférieurs ou égaux à 1000 divisibles par 2 ou 3 :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  i = 1;
  while(i <= 1000) {
    if(i%2 == 0) {
      cout << i << endl;
      i++;
      continue;
    }
```

3. arrêter alors l'exécution par *Ctrl-c*

```
    if(i%3 == 0)
      cout << i << endl;
    i++;
  }
  return 0;
}
```

Si le nombre est divisible par 2 on ne teste pas inutilement s'il est divisible par 3.

Remarque

Le programme précédent peut aussi s'écrire :

```
#include<iostream>
using namespace std;
int main() {
  int i;
  i = 1;
  while(i <= 1000) {
    if(i%2 == 0 || i%3 == 0)
      cout << i << endl;
    i++;
  }
  return 0;
}
```

Comme dans le programme précédent, si `i%2==0` est vrai, le test `i%3==0`, inutile puisqu'il s'agit d'un « ou » , n'est pas effectué. La seconde version est donc meilleure puisque plus simple.

2.2 Boucle `do ... while`

2.2.1 Forme générale

Elle ressemble beaucoup à la précédente mais, contrairement à celle-là, elle est toujours exécutée au moins une fois. Sa forme générale est :

```
do {
  instructions;
} while(expression);
```

Les instructions du bloc sont tout d'abord exécutées une première fois, puis `expression` est testée :

si elle vraie (valeur 1) le bloc est de nouveau exécuté et ainsi de suite jusqu'à ce que `expression` devienne fausse

si elle est fausse (valeur 0) le programme se poursuit avec les instructions suivant le `while` et la boucle n'aura donc été exécutée qu'une seule fois.

2.2.2 Exemple

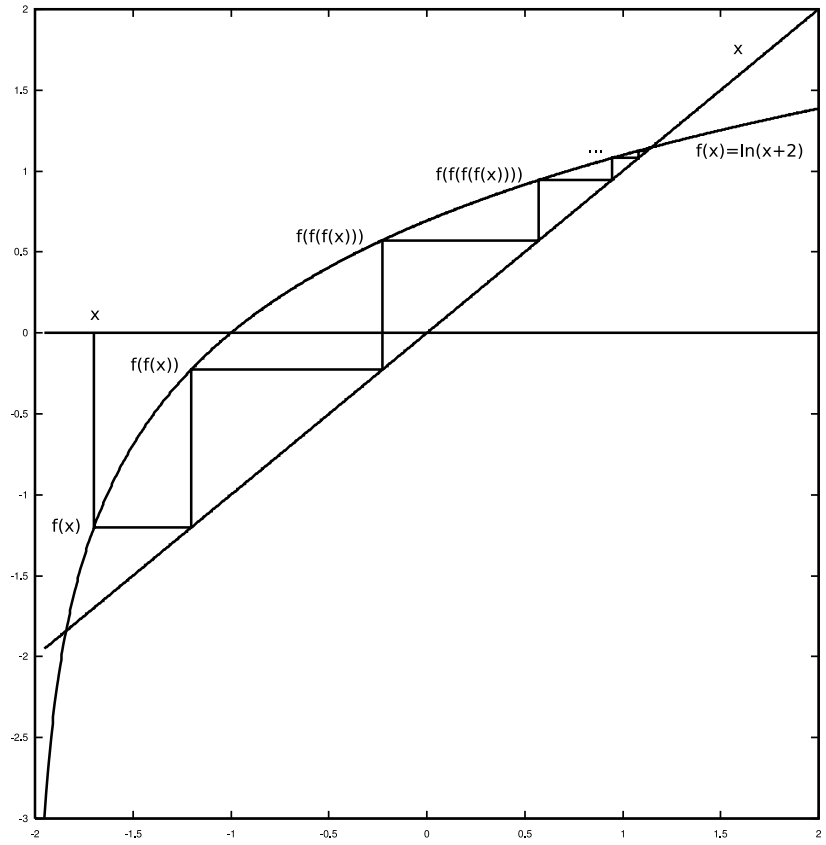


FIGURE 1 –

```
/* Resolution de l'equation x=log(x+2.) */
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  double x, eps, xa, diff;
  x = -1.75; eps = 1.e-5;
  do {
    xa = x;
    x = log(x+2.);
    diff = fabs(x-xa);
  } while(diff >= eps);
  cout << "Racine=" << x << endl;
  return 0;
}
```

Remarques

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

Dans le programme précédent rien ne garantit à priori que `diff` va devenir inférieur à `eps`. Il peut donc être prudent d'ajouter une limite au nombre de tours de boucle effectués.

2.3 Boucle `for`

Elle permet d'imposer à l'avance le nombre de tours de boucle à effectuer.

2.3.1 Exemple

On veut calculer les 10 premières puissances de 2 :

```
x = 1;
for(i = 1; i <= 10; i++) {
  x = x*2;
}
```

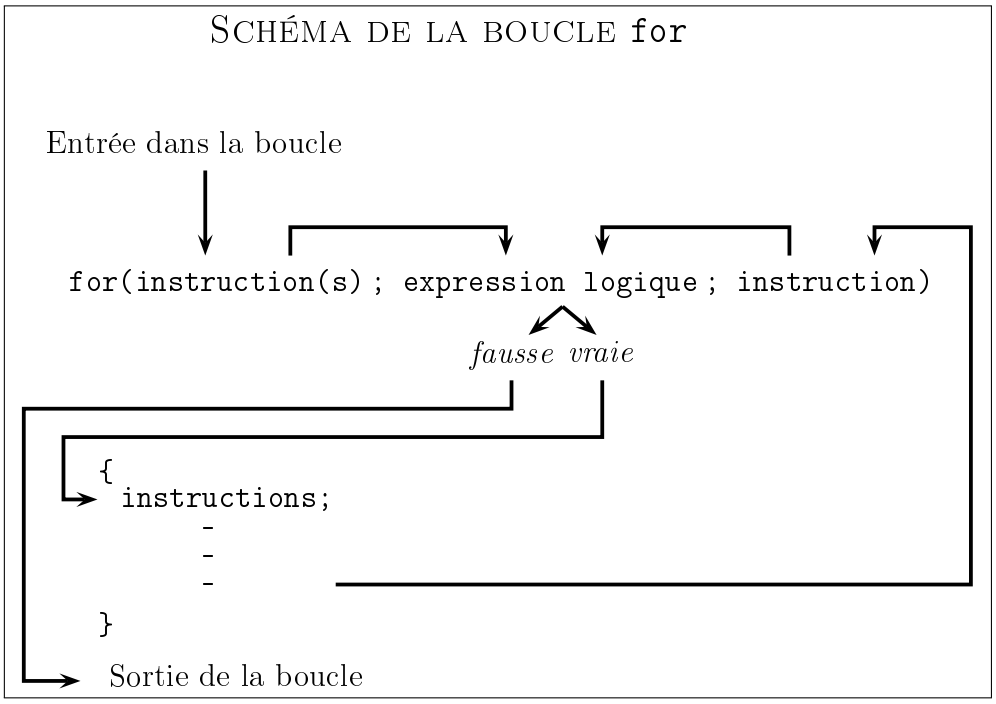
La boucle sera décrite pour `i` variant de 1 à 10 inclus.

2.3.2 Forme générale

Les parenthèses suivant le `for` contiennent trois champs séparés par deux point-virgules :

```
for(champ 1; champ 2; champ 3) {
  ...
}
```

Le `champ 1` contient une ou plusieurs instructions séparées par des virgules, exécutées une seule fois avant d'entrer dans la boucle (initialisation).
Le `champ 2` contient une expression à valeur logique calculée et interprétée comme « vrai» ou « faux» au début de chaque tour.
Le `champ 3` contient une instruction exécutée à la fin de chaque tour.



Chacun des trois champs est facultatif. Ainsi :

```
for(; expression logique;) {
  ...
}
```

est strictement équivalente à `while(expression logique)`.
Et :

```
for(;;) {
  ...
}
```

est une boucle infinie équivalente à `while(1)`, le `champ 2` vide étant considéré comme « vrai» . Pour en sortir il faut utiliser un `break`, étudié précédemment.

Remarque

L'initialisation peut contenir plusieurs instructions séparées par des virgules. Par exemple, dans le calcul des 10 premières puissances de 2 vu précédemment l'initialisation de `x` à 1. peut se faire dans le `for` :

```
for(x = 1., i = 1; i <= 10; i++) {
  x = x*2;
}
```

Remarque

Toute boucle `for` peut être écrite avec un `while` et inversement. On s'efforce de choisir la solution la plus simple et la plus lisible.

Exemples

1) Calcul des puissances de 2 jusqu'à 2^n.

```
#include<iostream>
using namespace std;
int main() {
  int i, i2, n;
  n = 10; i2 = 1;
  //-----------------------------------------------
  for(i = 0; i <= n; i++) {
    cout << "2 puissance " << i << " = " << i2 << endl;
    i2 *= 2;
  }
  //-----------------------------------------------
  return 0;
}
```

On obtient :
2 puissance 0 = 1
2 puissance 1 = 2
...
2 puissance 10 = 1024
Pour écrire le même programme avec un `while` on remplace ce qui est entre tirets dans le programme précédent par :

```
//-----------------------------------------------
i = 0;
while(i <= n) {
  cout << "2 puissance " << i << " = " << i2 << endl;
  i2 *= 2;
  i++;
}
//-----------------------------------------------
```

2) Calcul d'une intégrale par la méthode des rectangles.

$$\int_a^b f(x)\,dx \simeq h\sum_{i=1}^{n} f\left[a+(i-\frac{1}{2})h\right]$$

où n est le nombre de rectangles, h le pas d'intégration : $h = (b-a)/n$.
Appliquons cette formule à la fonction :

$$\frac{1}{1+x^2}$$

```
#include<iostream>
#include<math.h>
using namespace std;
int main() {
  double a = 1., b = 2., s = 0., h, som, u;
  int i, n = 1000;
  h = (b-a)/n;
  for(i = 1; i <= n; i++) {
    u = a + (i-0.5)*h;
    s += 1./(1.+u*u);
  }
  som = h*s;
  cout << "Valeur par les rectangles : " << som << endl;
  cout << "Valeur par la primitive : " << atan(b)-atan(a) << endl;
  return 0;
}
```

On obtient :
Valeur par les rectangles : 0.321750
Valeur par la primitive : 0.321751