Introduction à la Programmation des Algorithmes

4. Langage de programmation C

Semaine 2 : Bases

Automne 2025

Dr Julia Buwaya

UNIVERSITÉ
DE GENÈVE

Semaine 2 : Bases

1 Syntax et Types de base

Nous nous rappelons notre premier programme C minimal du dernier cours :

```c
#include <stdio.h>

int main(void) {
  printf("Bonjour\n");
  return 0;
}
```

En particulier :

- Chaque programme C possède une fonction `int main()` principale qui est exécutée lors du lancement du programme (lors de l'exécution du fichier binaire, après compilation du programme).
- `int` est la valeur de retour de la fonction `main()`, généralement l'entier 0.
- Les lignes du programme sont lues et exécutées dans l'ordre. En particulier les lignes qui se terminent par `;` dans le `main`.

On peut rajouter dans un fichier source des **commentaires** que le compilateur ignore.

Il suffit de les placer entre les symboles `/*` et `*/`. Un commentaire peut être sur plusieurs lignes.

Ou sur une seule ligne, précédé de `//`.

```c
#include <stdio.h>

int main(void) {
  /* Ceci est un commentaire.
  Deuxième ligne Commentaire.*/

  printf("Bonjour\n");

  // Ceci est également un commentaire.

  return 0;
}
```

Un **type** est soit pré-défini dans le langage lui-même (“type de base”), soit défini dans le programme.

Il y a de nombreux types de base en C, mais nous n’en utiliserons que trois :

- Le type `int`
- Le type `float`
- Le type `char`

Nous verrons plus tard comment nous pouvons créer nos propres types en combinant des types existants.

Avant d'être utilisée dans un programme une variable doit être **déclarée** en indiquant son type puis son identifiant.

Par exemple dans ce programme

```c
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier;
  return 0;
}
```

Plusieurs variables du même type peuvent être déclarées d'un coup en séparant leurs identifiants par des virgules

```c
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier, jackUnAutreEntier, n;
  return 0;
}
```

Lorsqu'une variable a été **déclarée**, on peut ensuite lui **affecter** une valeur avec le symbole '='.

```c
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier;
  bob_le_nombre_entier = 1025;
  return 0;
}
```

 En C une variable déclarée a une valeur indéfinie tant qu'une valeur ne lui a pas été explicitement affectée.

On peut également déclarer une ou des variables et faire les affectations de valeurs en même temps :

```
int nb1 = 5, nb2 = 27;
```

 Le symbole `'='` a donc un sens très différent de celui qu'il a en mathématique. Il ne définit pas une propriété qui est et sera toujours vraie. Il indique simplement de changer quelque chose en mémoire : "copie la valeur spécifiée à droite dans la variable spécifiée à gauche."

Ce programme est parfaitement correct, bien qu'il ne produise aucun résultat :

```c
#include <stdio.h>

int main(void) {
  int bob_le_nombre_entier;
  bob_le_nombre_entier = 10042;
  bob_le_nombre_entier = 12;
  bob_le_nombre_entier = 553;
  return 0;
}
```

2 printf

Un autre concept clé en programmation est celui de **fonction**, qui permet de donner un nom à un bout de programme que l'on veut pouvoir facilement ré-utiliser.

Pour exécuter une fonction, il suffit d'indiquer son identifiant, suivi entre parenthèses de valeurs qui modulent son fonctionnement et que l'on appelle ses **arguments**.

Par convention du langage, quand un programme écrit en C est exécuté, c'est la fonction `main` qui est appelée.

La fonction classique pour faire un affichage en C est `printf`.

On doit indiquer entre parenthèses les **arguments** qui définissent quoi afficher.

S'il y a un seul argument cela doit être une **array de caractères** entre ", et elle est affichée telle quelle.

Les caractères `\n` indique de revenir à la ligne.

```c
#include <stdio.h>

int main(void) {
  printf("Bonjour\n");
  return 0;
}
```

```c
#include <stdio.h>

int main(void) {
  printf("Et de un,\net de deux,\net de trois!\n");
  return 0;
}
```

affiche

```
Et de un,
et de deux,
et de trois!
```

Pour que l'affichage puissent dépendre du fonctionnement du programme, on doit passer plusieurs arguments à `printf`.

Le premier reste une **array de caractères** qui indique un "format" :

1. des caractères à afficher tels quels, et
2. des emplacements spécifiés avec `%d`, `%f` et `%c` qui indiquent où afficher des valeurs entières, réelles, et des caractères respectivement.

Les autres arguments indiquent les valeurs à mettre à la place des `%`.

Remarque : Il existe de nombreux autres formats spécifiables avec `%`.

Par exemple :

- `printf("toto\n")` affiche `'toto'`
- `printf("la valeur est %d\n", 3)` affiche `'la valeur est 3'`
- `printf("pi=%f\n", 3.1415926)` affiche `'pi=3.1415926'`
- `printf("1/%d=%f\n", 50, 0.02)` affiche `'1/50=0.02'`

On peut donc en particulier afficher les valeurs de variables.

```c
#include <stdio.h>

int main(void) {
  int a, b;
  a = 10;
  b = 20;
  printf("a=%d b=%d\n", a, b);
  b = 21;
  printf("a=%d b=%d\n", a, b);
  a = 11;
  printf("a=%d b=%d\n", a, b);
  return 0;
}
```

affiche

```
a=10 b=20
a=10 b=21
a=11 b=21
```

3 String remplacée par Array (tableau) de char

En C, il n'existe notamment pas de type **string** (comme en Python), mais un **array** (tableau) de caractères est créé pour remplir cette fonction. Les arrays correspondent à peu près aux listes en Python :

```c
#include <stdio.h>

int main() {

    // declaring and initializing a string
    char str[] = "Julia";

    // printing the string
    printf("The string is: %s\n", str);

    return 0;
}
```

Pour `printf` : emplacement spécifié avec %s qui indique où afficher la string.

4 Opérateurs

Les **opérateurs** sont les composants de base pour décrire des opérations à exécuter. Nous pouvons les grouper en six sous-familles :

- Opérateurs arithmétiques.
- Opérateur d'affectation.
- Opérateurs d'incrément, de décrément, et d'affectation composée.
- Opérateurs relationnels.
- Opérateurs logiques booléens.
- Opérateurs logiques bit à bit.

Opérateurs arithmétiques

Les **opérateurs arithmétiques** incluent le – unaire ainsi que les opérateurs binaires classiques :

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	modulo

La division `/` de deux entiers est une division euclidienne, elle retourne le quotient entier. En revanche si une des opérandes est une valeur à virgule, le résultat sera à virgule.

L'opérateur de modulo `%` calcule le reste de la division euclidienne.

Les opérateurs `+-` ont une priorité inférieure à `*/%`. Dans le cas d'opérateurs de même priorité, le calcul est effectué de gauche à droite.

```
printf("%d\n", 3 + 4 * 5);
```

affiche

23

```
printf("%d\n", 12 % 5);
```

affiche

2

```
printf("%d\n", 3 * (1 + 2) );
```

affiche

```
9
```

```
printf("%d\n", 15 / 20);
```

affiche

```
0
```

```
printf("%f\n", 15 / 20.0);
```

affiche

```
0.750000
```

```
printf("%d\n", 3 * 5 / 5);
```

affiche

3

```
printf("%d\n", 3 / 5 * 5);
```

affiche

0

Opérateurs d'incrément et de décrément

Les opérateurs d'**incrément**, et de **décrément**, sont des opérateurs unaires qui **modifient leur opérande.**

	Nom	Opération	Valeur
`n++`	post-incrément	ajoute 1 à `n`	`n` avant le changement
`++n`	pré-incrément	ajoute 1 à `n`	`n` après le changement
`n--`	post-décrément	soustrait 1 à `n`	`n` avant le changement
`--n`	pré-décrément	soustrait 1 à `n`	`n` après le changement

Par ex.

```
j = i++;
```

copie dans `j` la valeur courante de `i` puis incrémente `i` de 1. Équivalent à

```
j = i;
i = i + 1;
```

Alors que

```
j = ++i;
```

incrémente `i` de 1 puis copie dans `j` la nouvelle valeur de `i`. Équivalent à

```
i = i + 1;
j = i;
```

```c
int i, j;
i = 0;
j = i++;
printf("%d\n", j);
j = i++;
printf("%d\n", j);
j = ++i;
printf("%d\n", j);
j = ++i;
printf("%d\n", j);
```

affiche

0
1
3
4

Opérateurs d'affectation composée

Les opérateurs d'**affectation composée**, sont des opérateurs binaires qui modifient leur opérande de gauche.

	Opération	Valeur
`n += k`	`n = n + k`	`n` après le changement
`n -= k`	`n = n - k`	`n` après le changement
`n *= k`	`n = n * k`	`n` après le changement
`n /= k`	`n = n / k`	`n` après le changement
`n %= k`	`n = n % k`	`n` après le changement

Par ex.

`i *= 15;`

multiplie `i` par 15.

Et

```
1  int a = 9;
2  int b = 4 + (a /= 3);
3  printf("a=%d b=%d\n", a, b);
```

affiche

```
a=3 b=7
```

La ligne 2 divise `a` par 3, puis copie dans `b` la valeur de `4 + a`.

Il est extrêmement périlleux d'utiliser la valeur d'une affectation :

```
1  int n;
2
3  n = 2;
4  printf("%d\n", n = n + 1);
5
6  n = 2;
7  printf("%d\n", n += 1);
8
9  n = 2;
10 printf("%d\n", n++);
```

affiche

```
3
3
2
```

Opérateurs relationnels et opérateurs booléens

 Jusqu'à récemment, le langage C n'avait pas de type de donnée spécifique pour les valeurs booléennes et utilise pour cela seulement le type `int`.

Depuis octobre 2024 (C standard C23), il existe également les mots-clés `true` et `false`. Pour cela, il faut inclure `<stdbool.h>` et utiliser les types `bool`.

Mais un opérateur qui calcule un résultat booléen produira toujours la valeur 0 (false) ou la valeur 1 (true), et un opérande booléen peut être de n'importe quel type numérique et sera interprété comme "faux" s'il est nul et "vrai" sinon.

Le C met à notre disposition deux groupes d'opérateurs pour exprimer des conditions booléennes :

- Les **opérateurs relationnels** combinent des opérandes de types numériques et produisent des résultats booléens, et
- les **opérateurs booléens** combinent des opérandes booléens et produisent des résultats booléens.

Les **opérateurs relationnels** sont

`>`	strictement supérieur
`>=`	supérieur ou égal
`<`	strictement inférieur
`<=`	inférieur ou égal
`==`	égal
`!=`	différent

et sont tous de priorité identique.

```c
int a, b;
a = 3 < 4;
printf("a=%d\n", a);
b = 3 > 4;
printf("b=%d\n", b);
```

affiche

```
a=1
b=0
```

Les **opérateurs booléens** sont, par ordre de priorités :

`!`	négation logique (opérateur unaire)
`&&`	et logique
`||`	ou logique

```c
int x, y;
x = 0;
y = 1;
printf("%d\n", x && y);
printf("%d\n", x || y);
printf("%d\n", x && !y);
printf("%d\n", !x && y);
```

affiche

```
0
1
0
1
```

Ces opérateurs peuvent être combinés avec les opérateurs arithmétiques, d'affectation, et d'affectations composées que nous avons vus précédemment.

Nous pouvons résumer les priorités des opérateurs que nous avons déjà vus en les rangeant par ordre décroissant. Deux opérateurs sur la même ligne ont même priorité, auquel cas ils sont évalués de gauche à droite.

```
()

* / %

+ -

< <= > >=

== !=

&&

||

= += -= *= /= %=
```

 Comme indiqué précédemment pour les opérateurs booléens : **tout opérande nul est interpreté comme “faux” et tout opérande non-nul comme “vrai”.**

Ceci est donc valide en C :

```
float a = 3.1415926, b = -5;
int q = a && b;
```

La variable q vaudra 1.

Le laxisme du C permet aussi

```
int ouch = 3 < 4 < 2;

printf("%d\n", ouch);
```

qui affiche 1.

En effet l'évaluation de gauche à droite donne

```
ouch = 3 < 4 < 2

ouch = 1 < 2

ouch = 1
```

5 Conversions de types

Le langage C est très laxiste avec les types, et fait en particulier des **conversions implicites** : lorsque deux opérandes de types numériques différents sont combinées avec un opérateur, l'opérande avec la plus petite précision est convertie au type de l'autre avant de faire l'opération.

Par ex.

```
1  float x, y;
2  int z;
3
4  x = 3.1415926;
5  z = 2;
6  y = x + z; /* la valeur de z est convertie en float */
7  printf("%f\n", y);
```

compile sans erreur et affiche

```
5.141593
```

 Le compilateur ne signale pas de problème si de l'information est perdue lors d'une affectation.

```
1  float x, y;
2  int z;
3
4  x = 4;
5  y = 1/x;
6  printf("%f\n", x * y);
7
8  x = 4;
9  z = 1/x; /* ouch */
10 printf("%f\n", x * z);
```

compile sans erreur et affiche

```
1.000000
0.000000
```

Il est possible de forcer une conversion en indiquant un type entre parenthèses avant une expression. Cet opérateur a priorité sur les opérateurs arithmétiques.

```
1  int q = 9;
2  float r;
3
4  r = 1 / q;
5  printf("%f\n", r);
6
7  r = 1 / (float) q;
8  printf("%f\n", r);
```

affiche

```
0.000000
0.111111
```

6 Contrôle du flux

Les langages de programmation offrent des mécanismes pour contrôler dynamiquement la séquence d'instructions à exécuter.

On fait référence à cette séquence comme étant le **flux** du programme, et à sa modulation comme le **contrôle du flux.**

Nous allons voir dans la suite des **instructions de contrôle du flux** en C qui permettent de spécifier dynamiquement si certaines parties du programme doivent être ignorées ou répétées.

Instruction `if/else`

L'instruction principale de contrôle de flux qui permet de définir un comportement **conditionnel** en C est le `if`, qui a la forme suivante :

```
if(condition)
  clause_si_vrai
```

où `condition` définit la condition qui doit être vraie, et calcule un résultat qui prend la valeur "vrai" ou "faux".

Si ce résultat est "vrai" alors `clause_si_vrai` est évaluée.

La condition doit impérativement être entre parenthèses.

Remarques : Nous appellerons **clause** un bout de programme qui est :

- une expression terminée par un `;`, ou
- une instruction de contrôle du flux, ou
- un bloc entre `{}`, contenant plusieurs clauses.

Cette définition est donc **récursive**.

Traditionnellement, on indente un programme en décalant le contenu d'une clause de plusieurs caractères vers la gauche pour faciliter la lecture.

Par exemple

```
1  if(n < 0)
2    n = -n;
```

ou

```
1  if(x == 0) {
2    printf("x est nul!");
3    x = 1;
4    nb_de_trucs_nuls++;
5  }
```

Il est possible de rajouter dans un `if` une clause à évaluer dans le cas où la condition est fausse avec `else` :

```
if(condition)
  clause_si_vrai
else
  clause_si_faux
```

Par exemple si nous voulons programmer

$$\forall n \in \mathbb{N}, f(n) = \begin{cases} \frac{1}{2}n & \text{si } n \in 2\mathbb{N} \\ 3n+1 & \text{sinon} \end{cases}$$

nous pouvons faire

```
if(n % 2 == 0)
  f_de_n = n / 2;
else
  f_de_n = 3 * n + 1;
```

Il est aussi possible de rajouter autant de clauses additionnelles si la clause du `else` est elle même un `if` : :

```
if(condition_1)
  clause_1
else if(condition_2)
  clause_2
  ...
else if(condition_k)
  clause_k
else
  clause_defaut
```

Une clause est exécutée si et seulement si toutes les conditions qui la précèdent sont fausses à l'exception de celle juste avant qui doit être vraie.

La clause par défaut après le `else` final est évaluée si toutes les conditions sont fausses.

Par exemple

nous pouvons faire

```
if (n == 0)
  printf("n est nul");
else if(n % 2 == 0)
  printf("n est pair et non-nul");
else
  printf("n est impair");
```

qui est équivalent à

```
if (n == 0)
  printf("n est nul");
else {
  if(n % 2 == 0)
    printf("n est pair et non-nul");
  else
    printf("n est impair");
}
```

Instructions `while`, `do`/`while` et `for`

Une première instruction de contrôle du flux qui permet de répéter l'évaluation d'une clause est le `while`, qui a la forme suivante :

```
while(condition)
  clause_a_repeter
```

où `condition` définit la condition qui doit être vraie, et calcule un résultat qui prend la valeur "vrai" ou "faux".

Si résultat est "vrai" alors `clause_a_repeter` est évaluée, et le programme retourne avant le `while`.

Donc `clause_a_repeter` sera évaluée encore et encore, **tant que** `condition` est vraie.

Comme pour `if` la condition doit impérativement être entre parenthèses.

```c
int n = 0;

while(n < 6) {
  printf("%d\n", n);
  n++;
}
```

affiche

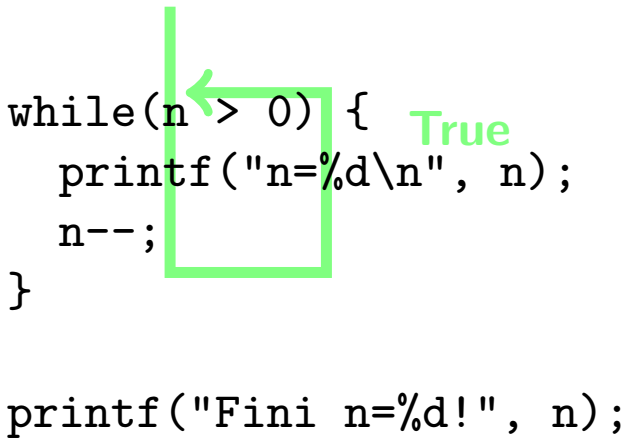
```
0
1
2
3
4
5
```

Il est important de remarquer que si la condition n'est pas vraie initialement, la clause n'est même pas exécutée une fois :

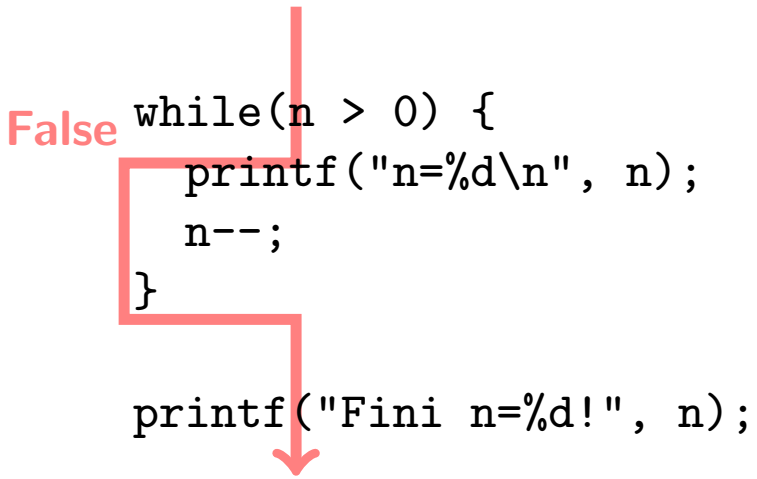
```
while(n > 0) {
  printf("n=%d\n", n);
  n--;
}

printf("Fini n=%d!", n);
```

Il est important de remarquer que si la condition n'est pas vraie initialement, la clause n'est même pas exécutée une fois :



Il est important de remarquer que si la condition n'est pas vraie initialement, la clause n'est même pas exécutée une fois :



```c
int n = 0;

while(n > 0) {
  printf("%d est strictement positif\n", n);
  n--;
}

while(n < 3) {
  printf("%d est strictement plus petit que 3\n", n);
  n++;
}
```

affiche

```
0 est strictement plus petit que 3
1 est strictement plus petit que 3
2 est strictement plus petit que 3
```

Une deuxième instruction de contrôle du flux qui permet de répéter l'évaluation d'une clause et le `do-while`, qui a la forme suivante :

```
do {
  clause_a_repeter
} while(condition);
```

`clause_a_repeter` est évaluée, puis, si la condition est "vraie", le programme retourne avant le `do`.

Donc `clause_a_repeter` sera évaluée encore et encore, **tant que** `condition` est vraie, mais **la clause sera toujours évaluée au moins une fois.**

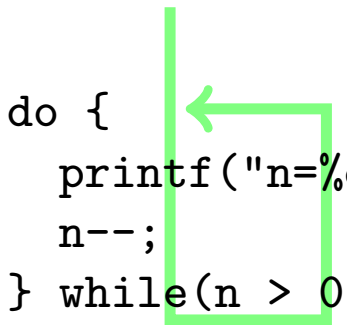
- La condition doit impérativement être entre parenthèses,
- `clause_a_repeter` peut être une seule ou plusieurs expressions terminées par un `;`, mais **est forcément entre** `{}`.

Donc, contrairement à ce qui se passe avec `while`, avec `do ... while` la clause est toujours exécutée une fois :

```
do {
  printf("n=%d\n", n);
  n--;
} while(n > 0);

printf("Fini n=%d!", n);
```

Donc, contrairement à ce qui se passe avec `while`, avec `do ... while` la clause est toujours exécutée une fois :

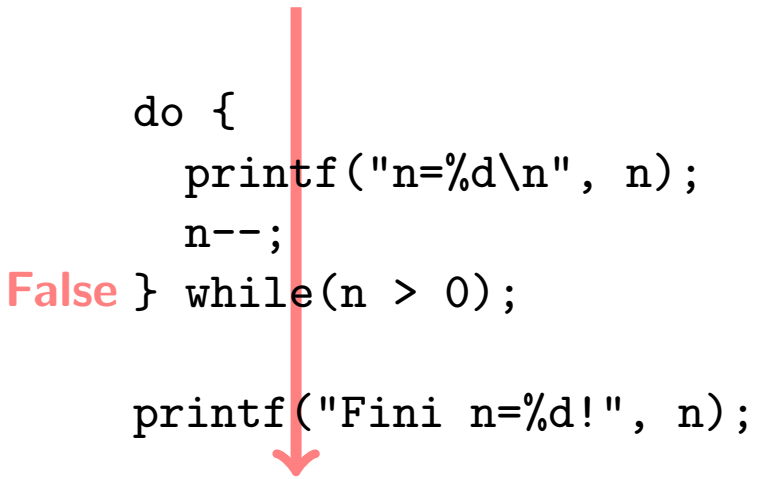


```
do {
  printf("n=%d\n", n);
  n--;
} while(n > 0);
```

True

```
printf("Fini n=%d!", n);
```

Donc, contrairement à ce qui se passe avec `while`, avec `do ... while` la clause est toujours exécutée une fois :



```c
int n = 0;

do {
  printf("%d est strictement positif\n", n);
  n--;
} while(n > 0);

do {
  printf("%d est strictement plus petit que 5\n", n);
  n++;
} while(n < 5);
```

affiche

```
0 est strictement positif
-1 est strictement plus petit que 5
0 est strictement plus petit que 5
1 est strictement plus petit que 5
2 est strictement plus petit que 5
3 est strictement plus petit que 5
4 est strictement plus petit que 5
```

La majorité des exécutions répétées sont des boucles du type

```
1  k = 0;
2
3  while(k < 10) {
4    printf("k=%d\n", k);
5    k++;
6  }
```

avec :

- une initialisation (ici `k = 0`),
- une condition (ici `k < 10`),
- une mise à jour (ici `k++`), et
- une clause à répéter (ici `printf("k=%d\n", k)`).

Une troisième instruction de contrôle du flux spécifique pour ce cas est le `for`, qui a la forme suivante :

```
for(initialisation; condition; mise_a_jour)
  clause_a_repeter
```

qui est exactement équivalent à

```
initialisation
while(condition) {
  clause_a_repeter
  mise_a_jour
}
```

Comme avec `while`, la clause peut ne jamais être exécutée si la condition n'est jamais vraie.

```c
int n;

for(n = 0; n < 5; n++)
  printf("n=%d\n", n);
```

affiche

```
n=0
n=1
n=2
n=3
n=4
```

La condition peut être une expression complexe.

```
1  int i = 0, j = 0;
2
3  for(i = 0; i + 25 >= i * i; i++) {
4    printf("i=%d j=%d\n", i, j);
5    j += i;
6  }
```

affiche

```
i=0 j=0
i=1 j=0
i=2 j=1
i=3 j=3
i=4 j=6
i=5 j=10
```

L'algorithme pour calculer $\sqrt{q}$ consiste à résoudre $x^2 - q = 0$ par dichotomie (binary search) :

1. définir $a_0 = 0$, $b_0 = q + 1$
2. itérer tant que $b_n - a_n \geq \epsilon$:
   - $c_n = \frac{a_n + b_n}{2}$
   - Si $c_n^2 - q \geq 0$ alors $a_{n+1} = a_n$ et $b_{n+1} = c_n$,
   - sinon $a_{n+1} = c_n$ et $b_{n+1} = b_n$.

```
1  float epsilon = 1e-6, q = 2;
2  float a = 0, b = 1 + q, c;
3
4  while(b - a >= epsilon) {
5    c = (a + b) / 2;
6    if (c * c - q >= 0) {
7      b = c;
8    } else {
9      a = c;
10   }
11 }
12
13 printf("sqrt(%f) ~ %f\n", q, c);
```

affiche

```
sqrt(2) ~ 1.414213
```

Instructions `break`, `continue`, `switch`, et `goto`

Des instructions que nous utiliserons très (très !) peu sont `break`, `continue`, `case` et `goto`, qui ne respectent pas la structure modulaire des clauses et ne sont utiles que dans des cas très particuliers.

Il arrive parfois que l'on veuille interrompre une boucle sans même finir la clause. L'instruction `break` permet de sortir immédiatement de la boucle dans laquelle elle se trouve.

```
1  n = 0;
2
3  while(n < 10000) {
4    n++;
5    if(n%17 == 0 && n%3 == 0) break;
6    n++;
7  }
8
9  printf("n=%d\n", n);
```

affiche

```
n=51
```

L'instruction `continue` ignore la suite dans l'itération courante de la boucle et va directement au début de la prochaine itération.

```
1  for(int n = 0; n < 6; n++) {
2    printf("n=%d\n", n);
3    if (n % 2 == 1) continue;
4    printf("pair!\n");
5  }
```

affiche

```
n=0
pair!
n=1
n=2
pair!
n=3
n=4
pair!
n=5
```

Il arrive assez fréquemment que l'on veuille exécuter des clauses associées à des valeurs spécifiques d'une expression.

L'instruction `switch` permet d'évaluer une expression et de continuer le programme à un endroit correspondant à la valeur obtenue spécifiée à l'aide du mot clé `case` ou `default`..

```
switch(expression) {
case valeur_1:
  expression;
  ...

case valeur_k:
  expression;

default:
  expression;
};
```

L'utilisation de `break` permet de continuer l'exécution du programme après le bloc du `switch`.

```
1  n = 2;
2
3  switch(n) {
4  case 0:
5    printf("zéro\n");
6    break;
7  case 1:
8    printf("un\n");
9    break;
10 case 2:
11   printf("deux\n");
12   break;
13 default:
14   printf("beaucoup\n");
15   break;
16 }
```

affiche

deux

Une instruction de contrôle du flux rarement utilisée est `goto`. Elle permet de faire revenir/continuer le programme à un endroit arbitraire spécifié par un **label**. Ce dernier est un identifiant suivi du symbole `:`.

```
1  #include <stdio.h>
2
3  int main(void) {
4
5    int n = 0;
6
7   debut:
8    printf("n=%d\n", n);
9    n++;
10   if(n < 10) goto debut;
11
12   return 0;
13 }
```

 Il est difficile d'écrire un programme modulaire et sans erreur avec des `goto`. Cette instruction est réservée à des situations exotiques.

La principale utilisation de `goto` consiste à sortir directement de plusieurs boucles imbriquées.

```
for(...) {
  for(...) {
    for(...) {
      ...
      if (...) goto mon_label_place_apres;
      ...
    }
  }
}
mon_label_place_apres:
```

```
1    int a, b, c;
2
3    for(a = 1; a < 1000; a++) {
4      for(b = 1; b < 1000; b++) {
5        for(c = 1; c < 1000; c++) {
6          if((a + b) * (b + c) == 4 * (a * b + b * c)) goto mon_label_plac
7        }
8      }
9    }
10   mon_label_place_apres:
11
12   printf("a=%d b=%d c=%d\n", a, b, c);
```

affiche

```
a=4 b=1 c=11
```

Fin

julia.buwaya@unige.ch