

Introduction à la programmation
Travaux pratiques: séance 3
INFO0201-1

B. Baert & F. Ludewig
Bruno.Baert@ulg.ac.be - F.Ludewig@ulg.ac.be



- Tableaux statiques (à 1 et 2 dimensions)
→ stockage d'une série d'éléments d'un même type

- Tableaux statiques (à 1 et 2 dimensions)
→ stockage d'une série d'éléments d'un même type
- Nombres aléatoires
→ génération de nombres pseudo-aléatoires

- Tableaux statiques (à 1 et 2 dimensions)
 - stockage d'une série d'éléments d'un même type
- Nombres aléatoires
 - génération de nombres pseudo-aléatoires
- Erreurs et débogage d'un programme
 - Comment repérer et corriger les erreurs d'un programme
 - Erreurs de compilation
 - Erreurs sémantiques
 - Utilisation du débogueur

Lorsque nous devons utiliser une suite ou série de données, il peut être plus commode d'utiliser un tableau. Celui-ci permet de centraliser les données sous un même nom de variable et l'accès aux différents éléments est réalisé par l'utilisation d'un indice.

Un tableau permet de mémoriser une suite ou série de valeurs d'un même type. La taille d'un tableau statique est fixée pour toute l'exécution du programme.

Exemples :

Lorsque nous devons utiliser une suite ou série de données, il peut être plus commode d'utiliser un tableau. Celui-ci permet de centraliser les données sous un même nom de variable et l'accès aux différents éléments est réalisé par l'utilisation d'un indice.

Un tableau permet de mémoriser une suite ou série de valeurs d'un même type. La taille d'un tableau statique est fixée pour toute l'exécution du programme.

Exemples :

- 1D : un vecteur, une liste, un ensemble de données, ...

Lorsque nous devons utiliser une suite ou série de données, il peut être plus commode d'utiliser un tableau. Celui-ci permet de centraliser les données sous un même nom de variable et l'accès aux différents éléments est réalisé par l'utilisation d'un indice.

Un tableau permet de mémoriser une suite ou série de valeurs d'un même type. La taille d'un tableau statique est fixée pour toute l'exécution du programme.

Exemples :

- 1D : un vecteur, une liste, un ensemble de données, ...
- 2D : une matrice, ...

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

```
type nom_tableau[taille_tableau];
```

Un tableau 2D :

```
type nom_tableau[taille_1][taille_2];
```

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```


Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

```
type nom_tableau[taille_tableau];
```

Un tableau 2D :

```
type nom_tableau[taille_1][taille_2];
```

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

- **Type de variable : int, float, double, char, ...**

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

```
type nom_tableau[taille_tableau];
```

Un tableau 2D :

```
type nom_tableau[taille_1][taille_2];
```

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

- **Type** de variable : `int`, `float`, `double`, `char`, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

```
type nom_tableau[taille_tableau];
```

Un tableau 2D :

```
type nom_tableau[taille_1][taille_2];
```

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

- **Type** de variable : `int`, `float`, `double`, `char`, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible
- `[...]` : autant de paires de crochets que de dimensions

Les tableaux statiques : déclaration

Comment déclarer un tableau 1D :

```
type nom_tableau[taille_tableau];
```

Un tableau 2D :

```
type nom_tableau[taille_1][taille_2];
```

Exemple :

```
1 int v[3]; // tableau de 3 éléments entiers
2 double tableau_2D[10][20]; // tableau de 10x20 double
```

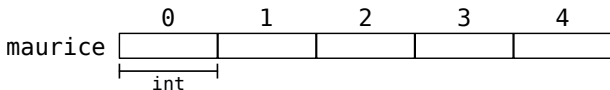
- **Type** de variable : `int`, `float`, `double`, `char`, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible
- `[...]` : autant de paires de crochets que de dimensions
- **Taille** : entre chaque crochet préciser la taille de la dimension correspondante du tableau

Les tableaux statiques : déclaration et initialisation

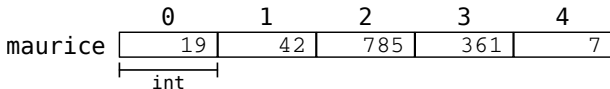
Déclaration d'un tableau = **réservation** de la mémoire

Les éléments (cases) d'un tableau sont contigus dans la mémoire

```
1 int maurice[5]; // déclaration d'un tableau de 5 int
```



```
1 // déclaration et initialisation d'un tableau de 5 int  
2 int maurice[5] = {19, 42, 785, 361, 7};
```



Si on veut initialiser tout le tableau à zéro :

```
1 int peter[5] = {0}; // équivalent à {0, 0, 0, 0, 0}
```

Les tableaux statiques : accès aux éléments

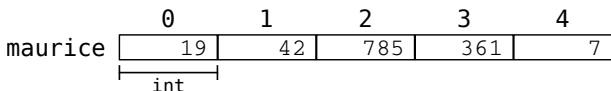
	0	1	2	3	4
maurice	19	42	785	361	7

int

Les éléments du tableau sont numérotés par des **indices** de **0** à **$N - 1$** , où N est la taille du tableau.

Ils représentent l'**offset** (décalage) de l'élément à partir du début du tableau : c'est pour cela que le premier élément possède l'indice 0.

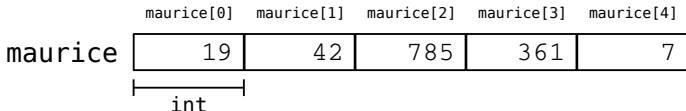
Les tableaux statiques : accès aux éléments



Les éléments du tableau sont numérotés par des **indices** de 0 à $N - 1$, où N est la taille du tableau.

Ils représentent l'**offset** (décalage) de l'élément à partir du début du tableau : c'est pour cela que le premier élément possède l'indice 0 .

L'accès aux éléments du tableau s'effectue au moyen de crochets [...] entre lesquels on place l'indice de l'élément auquel on veut accéder.



L'indice doit toujours être un **entier** !

Il peut être une valeur **constante** ou la valeur d'une **variable**.

Les tableaux statiques : accès aux éléments

Attention à la numérotation des indices !

```
1 int v[3] = {4, 6, 8};
2 cout << "Element 1 = " << v[0] << endl; // affiche 4
3 cout << "Element 2 = " << v[1] << endl; // affiche 6
4 cout << "Element 3 = " << v[2] << endl; // affiche 8
5 v[1] = 12;
6 cout << "Element 2 = " << v[1] << endl; // affiche 12
```

Tout comme pour les tableaux 1D, chaque dimension d'un tableau à D dimensions est numérotée de 0 à $N_D - 1$ où N_D est la taille de la D -ième dimension du tableau.

```
1 int v[3][4] = { {1, 2, 3, 4},
2                {5, 6, 7, 8},
3                {8, 9, 10, 11} };
4 cout << v[1][2] << endl; // affiche 7
5 cout << v[2][0] << endl; // affiche 8
```


- Erreur d'accès à un tableau

L'utilisation d'un indice non compris entre 0 et $N - 1$ ne sera pas détectée par le compilateur (C/C++).

Le programme tentera donc d'accéder à une partie de la mémoire qui n'a pas été réservée pour le tableau. Dans ce cas :

- Erreur d'accès à un tableau

L'utilisation d'un indice non compris entre 0 et $N - 1$ ne sera pas détectée par le compilateur (C/C++).

Le programme tentera donc d'accéder à une partie de la mémoire qui n'a pas été réservée pour le tableau. Dans ce cas :

- Le programme peut s'arrêter avec une erreur du type "segmentation fault" (accès à une zone mémoire interdite par le système)

- Erreur d'accès à un tableau

L'utilisation d'un indice non compris entre 0 et $N - 1$ ne sera pas détectée par le compilateur (C/C++).

Le programme tentera donc d'accéder à une partie de la mémoire qui n'a pas été réservée pour le tableau. Dans ce cas :

- Le programme peut s'arrêter avec une erreur du type "segmentation fault" (accès à une zone mémoire interdite par le système)
- Parfois l'exécution du programme ne s'arrêtera pas. On accède alors à de la mémoire non allouée, qui peut correspondre à d'autres variables et être modifiée pendant l'exécution. Seules des erreurs de valeur et donc de fonctionnement du programme apparaîtront dans ce cas : ce sont les plus difficiles à détecter.

La plupart du temps, on utilise des boucles pour initialiser, traiter et afficher des tableaux. Dans ce cas, une variable entière est utilisée pour accéder aux éléments du tableaux.

```
1  int main()
2  {
3      int v[1000] = {0};
4      // initialiser avec des multiples de 2
5      for(int i = 0; i < 1000; i++)
6      {
7          v[i] = i*2;
8      }
9      return 0;
10 }
```

Il est fréquent en programmation d'avoir besoin de générer des **nombres aléatoires**. Pour ce faire, il existe une fonction

```
rand();
```

Cette fonction retourne un entier positif pseudo-aléatoire compris dans l'intervalle [0 :RAND_MAX].

Pour l'utiliser, il faut inclure la bibliothèque **cstdlib** :

```
1 #include <cstdlib>
```

- Pour définir les limites de l'intervalle des nombres générés
→ Utilisation de la fonction modulo

```
1 int a = rand()%100; // nombre entre 0 et 99
```

- Pour définir les limites de l'intervalle des nombres générés
→ Utilisation de la fonction modulo

```
1 int a = rand()%100; // nombre entre 0 et 99
```

- Obtenir un nombre réel
→ Convertir en réel et diviser

```
1 // nombre entre 0 et 1
2 double a = (double)rand()/RAND_MAX;
3
4 // nombre entre 10 et 20
5 double b = 10. + 10.*(double)rand()/RAND_MAX;
```

Nombres aléatoires : initialisation du générateur

Le générateur produira toujours la même suite de nombres (pseudo-aléatoires). Pour obtenir des nombres aléatoires qui ne soient pas identiques d'une exécution à l'autre, on utilise la fonction **srand(x)** qui permet de commencer la série à un endroit déterminé.

Cette valeur de départ x est appelée *seed* (graine) et doit être différente à chaque exécution.

On utilise généralement pour cette valeur le nombre de secondes écoulées depuis le 1^{er} Janvier 1970 (qui est donc différent à chaque exécution). Ce nombre est obtenu par la fonction **time(NULL)**.

```
1 // initialisation:
2 srand(time(NULL));
3 // nombre aléatoire différent à chaque exécution:
4 int a = rand();
```

La fonction **time** nécessite d'inclure la librairie **<ctime>**

Lors de l'implémentation d'un code, différents types d'erreurs peuvent survenir. Il y a les

Lors de l'implémentation d'un code, différents types d'erreurs peuvent survenir. Il y a les

- erreurs **syntaxiques** : elles sont détectées par le compilateur car elles ne respectent pas la syntaxe (manière d'écrire les instructions) prévue par le langage
→ Ce sont les erreurs de déclaration, de notations des instructions (points-virgules), ...

Lors de l'implémentation d'un code, différents types d'erreurs peuvent survenir. Il y a les

- erreurs **syntaxiques** : elles sont détectées par le compilateur car elles ne respectent pas la syntaxe (manière d'écrire les instructions) prévue par le langage
→ Ce sont les erreurs de déclaration, de notations des instructions (points-virgules), ...
- erreurs **sémantiques** : ces erreurs ne sont **pas** détectées par le compilateur ! Elles correspondent à des erreurs **logiques** dans la **signification** de la suite des instructions (ce que **fait** le programme) → Le compilateur ne connaît pas l'objectif du programme et ne peut donc pas les détecter.

Deux cas sont alors possibles :

- Le programme s'arrête avec un message d'erreur comme "Segmentation Fault" ;
- Le programme ne fait pas ce qu'il devrait (**ATTENTION**).

- Diviser deux entiers :
quand on divise deux **int** l'un par l'autre, on obtient un nombre entier, même si l'on place le résultat dans un **double**.
Pour obtenir un résultat non-entier, il faut écrire

```
1 int a;  
2 int b;  
3 double c = (double)a/b;
```

→ erreur sémantique

- Diviser deux entiers :
quand on divise deux **int** l'un par l'autre, on obtient un nombre entier, même si l'on place le résultat dans un **double**.
Pour obtenir un résultat non-entier, il faut écrire

```
1 int a;  
2 int b;  
3 double c = (double)a/b;
```

→ erreur sémantique

- Expression de comparaison :
utiliser `=` (affectation) à la place de `==` (comparaison)
→ erreur sémantique

- Diviser deux entiers :
quand on divise deux **int** l'un par l'autre, on obtient un nombre entier, même si l'on place le résultat dans un **double**.
Pour obtenir un résultat non-entier, il faut écrire

```
1 int a;  
2 int b;  
3 double c = (double)a/b;
```

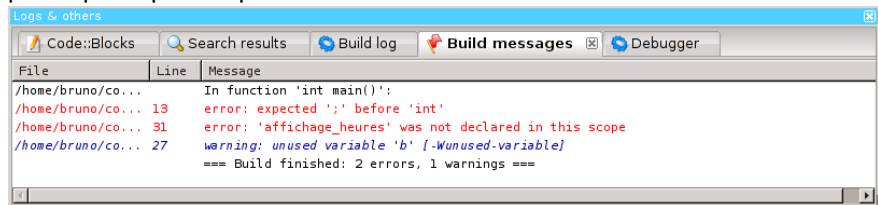
→ erreur sémantique

- Expression de comparaison :
utiliser `=` (affectation) à la place de `==` (comparaison)
→ erreur sémantique
- Oublier un point-virgule :
toutes les instructions se terminent par un point-virgule.
→ erreur syntaxique

Erreurs syntaxiques

Lorsque des erreurs **syntaxiques** sont présentes, le compilateur les énumère, fournit le numéro des lignes qui présentent des erreurs, et un petit message décrivant brièvement le type de problème rencontré.

Il est important de commencer par lire et résoudre **les premières erreurs** pour commencer, car les erreurs suivantes peuvent être provoquées par les précédentes.

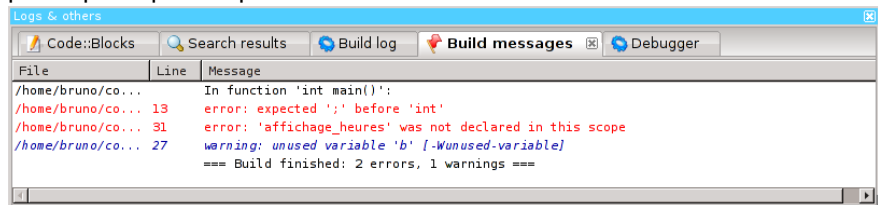


The screenshot shows a window titled "Logs & others" with several tabs: "Code::Blocks", "Search results", "Build log", "Build messages", and "Debugger". The "Build messages" tab is active and displays the following output:

File	Line	Message
/home/bruno/co...		In function 'int main()':
/home/bruno/co...	13	error: expected ';' before 'int'
/home/bruno/co...	31	error: 'affichage_heures' was not declared in this scope
/home/bruno/co...	27	warning: unused variable 'b' [-Wunused-variable]
		=== Build finished: 2 errors, 1 warnings ===

Lorsque des erreurs **syntaxiques** sont présentes, le compilateur les énumère, fournit le numéro des lignes qui présentent des erreurs, et un petit message décrivant brièvement le type de problème rencontré.

Il est important de commencer par lire et résoudre **les premières erreurs** pour commencer, car les erreurs suivantes peuvent être provoquées par les précédentes.



The screenshot shows a window titled "Logs & others" with several tabs: "Code::Blocks", "Search results", "Build log", "Build messages", and "Debugger". The "Build messages" tab is active, displaying a table with columns "File", "Line", and "Message".

File	Line	Message
/home/bruno/co...		In fonction 'int main()':
/home/bruno/co...	13	error: expected ';' before 'int'
/home/bruno/co...	31	error: 'affichage_heures' was not declared in this scope
/home/bruno/co...	27	warning: unused variable 'b' [-Wunused-variable]
		=== Build finished: 2 errors, 1 warnings ===

Le compilateur fournit également des messages d'alerte ("warnings"). Ceux-ci n'empêchent pas de compiler le programme, mais mettent en lumière des manières de faire non recommandées. Il est donc **fortement conseillé** de les corriger de manière à n'avoir ni erreurs, ni *warnings* lors de la compilation.

Les erreurs sémantiques ne sont pas directement visibles et sont donc **difficiles à détecter**.

Il est important de pouvoir les repérer et les corriger.

Conseils :

- Approcher petit-à-petit de l'erreur en vérifiant d'abord les grandes étapes du programme puis en réduisant progressivement la zone de vérification ;

Les erreurs sémantiques ne sont pas directement visibles et sont donc **difficiles à détecter**.

Il est important de pouvoir les repérer et les corriger.

Conseils :

- Approcher petit-à-petit de l'erreur en vérifiant d'abord les grandes étapes du programme puis en réduisant progressivement la zone de vérification ;
- Tester le programme avec des valeurs simples pour lesquelles la solution est connue. De cette manière, il est possible de contrôler efficacement les variables car les valeurs attendues sont connues ;

Les erreurs sémantiques ne sont pas directement visibles et sont donc **difficiles à détecter**.

Il est important de pouvoir les repérer et les corriger.

Conseils :

- Approcher petit-à-petit de l'erreur en vérifiant d'abord les grandes étapes du programme puis en réduisant progressivement la zone de vérification ;
- Tester le programme avec des valeurs simples pour lesquelles la solution est connue. De cette manière, il est possible de contrôler efficacement les variables car les valeurs attendues sont connues ;
- Structurer son code dès le début, avec des noms de variables explicites et des commentaires, ce qui permet de contrôler aisément le programme étape par étape.

Deux manières principales de procéder :

- 1 Afficher les valeurs de certaines variables dans la console
Simple et rapide pour contrôler le déroulement de certains points clés
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et erreurs simples

Deux manières principales de procéder :

- 1 Afficher les valeurs de certaines variables dans la console
Simple et rapide pour contrôler le déroulement de certains points clés
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et erreurs simples

- 2 Utiliser le débogueur

Code::Blocks dispose d'une interface intégrée avec un débogueur. Il permet :

- d'exécuter le code source étape par étape
- de contrôler directement les valeurs des variables du programme à chaque instant
- d'interrompre le programme à un moment précis pour vérifier son état

Pour déboguer un programme :

- Placer le projet en configuration “Debug”



Pour déboguer un programme :

- Placer le projet en configuration “Debug”



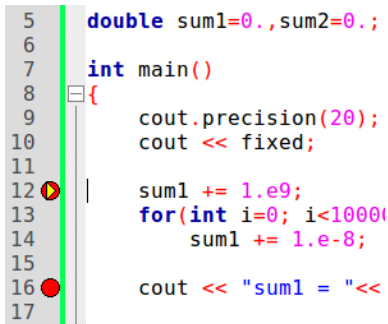
- Compiler le programme normalement, mais ne pas l'exécuter de la manière habituelle. La barre d'outil *Debug* sera utilisée à la place :



- Le premier bouton sert à exécuter le programme jusqu'au point d'arrêt suivant
- Le deuxième exécute le programme jusqu'à la position actuelle du curseur
- Le troisième exécute la ligne de code suivante
- etc...

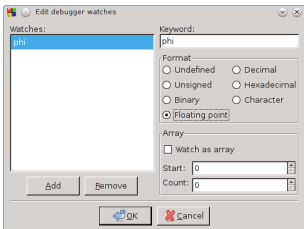
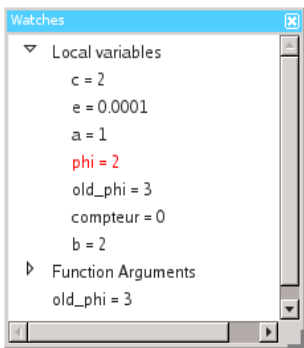
Utilisation du débogueur avec Code::Blocks (2/3)

- Pour ajouter un *point d'arrêt*, il suffit de cliquer dans la gouttière (entre le code et le numéro de ligne) à côté de la ligne souhaitée;
- En cours de débogage, Code::Blocks indique la ligne en cours d'exécution par une petite flèche jaune.



```
5 double sum1=0.,sum2=0.;
6
7 int main()
8 {
9     cout.precision(20);
10    cout << fixed;
11
12    sum1 += 1.e9;
13    for(int i=0; i<10000; i++)
14        sum1 += 1.e-8;
15
16    cout << "sum1 = " <<
17
```


Utilisation du débogueur avec Code::Blocks (3/3)



- La fenêtre *Watches* permet d'afficher les valeurs actuelles de toutes les variables locales.
- Lorsque la valeur d'une variable a été modifiée lors de la dernière instruction, elle s'affiche en rouge.
- Il est possible de suivre la valeur d'une variable spécifique non affichée par défaut, en ajoutant un 'espion'. Cela signifie que Code::Blocks affichera constamment sa valeur dans la fenêtre *Watches*.

- 1 Débogage d'un programme :
 - Ouvrir le projet fourni lors du TP et corriger les erreurs à l'aide des informations données par le compilateur et le débogueur.
- 2 Traitement des cotes d'un examen avec un tableau :
 - Déclarer un tableau pour 10 cotes ;
 - Faire saisir les valeurs des cotes par l'utilisateur ;
 - Afficher le tableau des cotes ;
 - Afficher la meilleure cote et la moins bonne cote ;
 - Calculer la moyenne et l'afficher.
- 3 Recherche dans un tableau
 - Créer un tableau de 50 éléments de type **double** ;
 - Le remplir de nombres aléatoires ;
 - Afficher le tableau ;
 - Déterminer l'indice du plus petit élément du tableau ;
 - Afficher la valeur de cet élément et l'indice correspondant.

- ④ Tri de tableau :
Modifier l'exercice précédent pour trier les éléments du tableau du plus petit au plus grand. Les étapes pour y parvenir sont
 - ① Afficher le tableau non trié ;
 - ② Parcourir le tableau à la recherche du plus petit élément ;
 - ③ Permuter cet élément avec le premier élément du tableau ;
 - ④ Chercher le plus petit élément parmi ceux restants (Remarque : les éléments restants sont tous les éléments sauf le premier, dont on sait désormais qu'il est le plus petit de tout le tableau) ;
 - ⑤ Permuter cet élément avec le second du tableau ;
 - ⑥ Poursuivre avec le 3^e, le 4^e élément, etc... jusqu'à ce que le tableau soit trié ;
 - ⑦ Afficher le tableau trié.
- ⑤ Calcul d'un déterminant :
 - Demander à l'utilisateur d'encoder une matrice 3x3 ;
 - Afficher la matrice dans la console ;
 - Calculer le déterminant de cette matrice ;
 - Afficher le résultat du calcul dans la console.

- 7 Calcul de π par une série (formule de Leibniz) :

Soit la série qui converge vers $\frac{\pi}{4}$,

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots = \frac{\pi}{4}$$

Ecrire un programme qui calcule les termes de cette série ainsi que la somme des termes.

- Calculer la série jusqu'à ce que le dernier terme soit successivement plus petit que 10^{-3} , 10^{-6} et 10^{-9} . Afficher le nombre de termes contenus dans chacune de ces sommes, ainsi que la valeur de π que l'on peut en déduire.

7 Calcul de π par dichotomie :

Soit la fonction $f(x) = \cos(x)$ considérée sur l'intervalle $[0; 3]$, le zéro de $f(x)$ est $x = \pi/2$. Calculer le zéro de $\cos(x)$ permet donc de déterminer la valeur de π . Ecrire un programme qui permet de calculer ce zéro par la méthode de la dichotomie. Cette méthode consiste à diviser de manière répétée l'intervalle sur lequel le zéro est recherché (ici, $[0; 3]$) par 2 jusqu'à obtenir un intervalle de la précision voulue autour de ce zéro. Pour ce faire, si $[a; b]$ est l'intervalle sur lequel le zéro est recherché :

- 1 Calculer $f(a)$, $f(b)$ et $f(c)$, avec $c = (a + b)/2$;
 - Si $f(a) \times f(c) < 0$, on sait que le zéro se trouve dans $[a; c]$
 - Si $f(b) \times f(c) < 0$, on sait que le zéro se trouve dans $[c; b]$
- 2 Recommencer l'étape (1) avec le nouvel intervalle à considérer ($[a; c]$ ou $[c; b]$) ;
- 3 Procéder ainsi jusqu'à ce que l'intervalle $[a; b]$ soit réduit à la précision voulue e . Tester avec $e = 10^{-9}$;
- 4 En déduire la valeur de π .