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

G.Allemand, A.Wafflard & R. Chrétien, G. Vanhaele & B. Baert, X. Baumans
guillaume.allemand@uliege.be - adrien.wafflard@uliege.be

Programme de la séance

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

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type
- Lecture/Ecriture dans un fichier
  $\rightarrow$ Chargement de plus de données qu'au clavier
  $\rightarrow$ (ré-)Utilisation de données plus complexes
  $\rightarrow$ Sauvegarde de données

Programme de la séance

- Tableaux statiques (à 1 et 2 dimensions)
  $\rightarrow$ stockage d'une série d'éléments d'un même type
- Lecture/Ecriture dans un fichier
  $\rightarrow$ Chargement de plus de données qu'au clavier
  $\rightarrow$ (ré-)Utilisation de données plus complexes
  $\rightarrow$ Sauvegarde de données
- Erreurs et débogage d'un programme
  $\rightarrow$ Comment repérer et corriger les erreurs d'un programme
  - Erreurs de compilation
  - Erreurs sémantiques
  - Utilisation du débogueur

Les tableaux statiques

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 :

Les tableaux statiques

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, ...

Les tableaux statiques

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
```

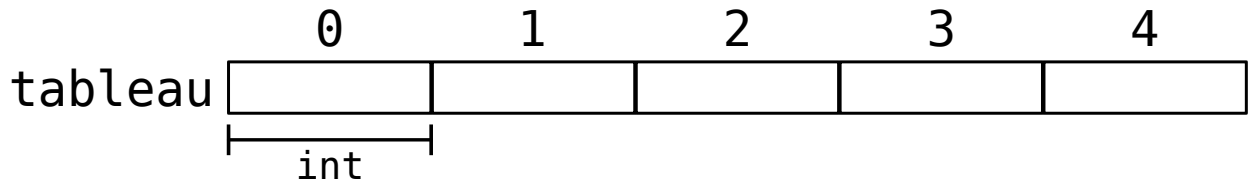
- **Type** de variable : **int**, **float**, **double**, **char**, ...
- **Nom** du tableau : le plus **représentatif** et **concis** possible
- **[...]** : autant de paires de crochets que de dimensions
- **Taille** : entre chaque crochet préciser la taille de la dimension correspondante du tableau

Les tableaux statiques : déclaration et initialisation

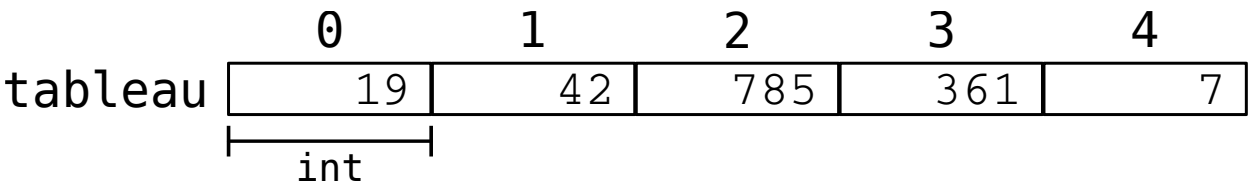
Déclaration d'un tableau = **réservation** de la mémoire

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

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




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

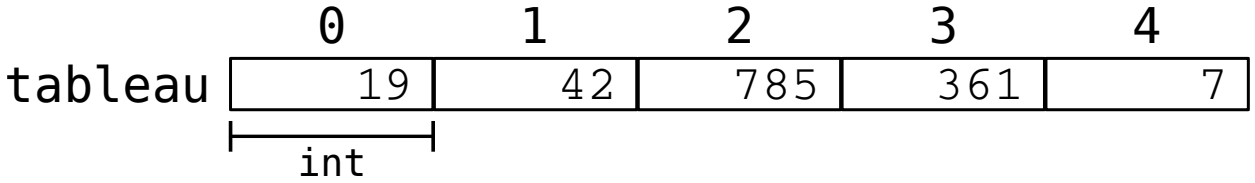


Si on veut initialiser tout le tableau à zéro :

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

Attention au volume de mémoire !

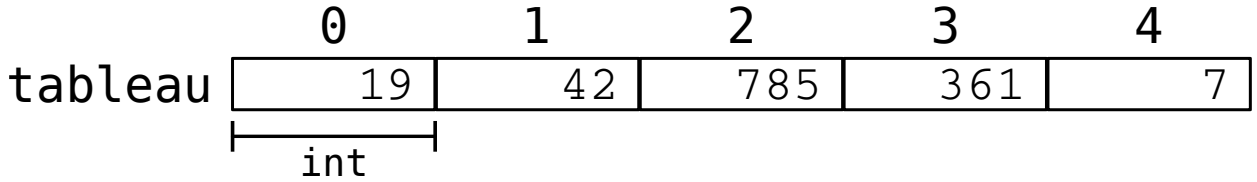
Les tableaux statiques : accès aux élements



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.

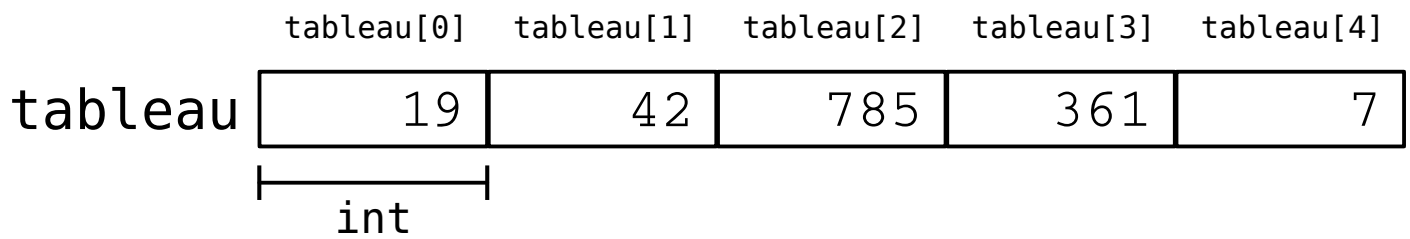
Les tableaux statiques : accès aux élements



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 élements 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 élements

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
```

Les tableaux statiques : accès aux élements

- 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 :

Les tableaux statiques : accès aux élements

- 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)

Les tableaux statiques : accès aux élements

- 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 apparaitront dans ce cas : ce sont les plus difficiles à détecter.

Exemple en pratique

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 }
```

La programmation orientée objet

C++ est un langage de programmation orienté objet.
Une classe est un modèle permettant de créer des objets. Par exemple, **cin** et **cout** sont des objets des classes istream et ostream respectivement.

Instanciation d'une classe : création d'un objet relatif à cette classe.
Exemple de création d'un objet appelé mon_fichier appartenant à la classe ofstream :

```
1 ofstream mon_fichier("data.txt");
```

Une classe est une variable qui contient

- d'autres variables (attributs de la classe). Ex : une chaîne de caractère, "data.txt".
- des fonctions (fonctions membres ou méthodes). Ex. : mon_fichier.is_open()

ofstream, ifstream, vector et string sont les classes qui seront abordées dans le cadre de ce cours.

Lire/Ecrire dans un fichier

Rappel : Pour la saisie au clavier et l'affichage à l'écran, il existe deux objets **cin** et **cout** :

- Ce sont en fait des variables un peu particulières : des "objets", qui représentent la console et le clavier ;
- On utilise les opérateurs "«" et "»" pour lire et écrire dans ces objets.

Lire/Ecrire dans un fichier

Rappel : Pour la saisie au clavier et l'affichage à l'écran, il existe deux objets **cin** et **cout** :

- Ce sont en fait des variables un peu particulières : des "objets", qui représentent la console et le clavier ;
- On utilise les opérateurs "«" et "»" pour lire et écrire dans ces objets.

Pour écrire et lire dans des fichiers, il existe des types d'objets similaires, **ifstream** et **ofstream**. Pour les utiliser, il est nécessaire d'inclure la librairie **fstream** :

```
1 #include <fstream>
```

- **ifstream** : "*input-file-stream*" est un "flux de fichier en entrée" qui permet de lire un fichier et d'en placer le contenu dans des variables du programme ;
- **ofstream** : "*output-file-stream*" est un "flux de fichier en sortie" qui permet d'écrire des données dans un fichier.

Ecrire dans un fichier (1/4)

Etapes pour l'écriture dans un fichier :

1. Ouvrir le fichier
   $\rightarrow$ Permet de spécifier dans quel fichier écrire et de quelle manière.

Ecrire dans un fichier (1/4)

Etapes pour l'écriture dans un fichier :

1. Ouvrir le fichier
   $\rightarrow$ Permet de spécifier dans quel fichier écrire et de quelle manière.
2. Vérifier que le fichier est correctement ouvert
   $\rightarrow$ Si un problème est survenu lors de l'ouverture du fichier, on ne pourra pas écrire dedans. Il faut donc vérifier d'abord.

Ecrire dans un fichier (1/4)

Etapes pour l'écriture dans un fichier :

1. Ouvrir le fichier
   → Permet de spécifier dans quel fichier écrire et de quelle manière.
2. Vérifier que le fichier est correctement ouvert
   → Si un problème est survenu lors de l'ouverture du fichier, on ne pourra pas écrire dedans. Il faut donc vérifier d'abord.
3. Ecrire dans le fichier
   → Ecrire toutes les données souhaitées dans le fichier.

Ecrire dans un fichier (1/4)

Etapes pour l'écriture dans un fichier :

1. Ouvrir le fichier
   → Permet de spécifier dans quel fichier écrire et de quelle manière.
2. Vérifier que le fichier est correctement ouvert
   → Si un problème est survenu lors de l'ouverture du fichier, on ne pourra pas écrire dedans. Il faut donc vérifier d'abord.
3. Ecrire dans le fichier
   → Ecrire toutes les données souhaitées dans le fichier.
4. Fermer le fichier
   → Lorsque les opérations d'écriture sont terminées, il est nécessaire de fermer le fichier pour signaler au système qu'on n'y accèdera plus. Si l'écriture des données n'était pas terminée, cela force également le système à le faire à ce moment-là.

Ecrire dans un fichier (2/4)

La première chose à faire est d'ouvrir le fichier. Pour cela, il faut déclarer un objet **ofstream** avec les paramètres suivants :

ofstream nom_variable("*chemin_vers_le_fichier*", *mode*) ;

- "*chemin_vers_le_fichier*" : une chaîne de caractères qui indique le nom du fichier à ouvrir ;
- *mode* : paramètres qui indiquent la manière d'ouvrir le fichier :
  - **ios**::**trunc** (truncate / tronquer) : spécifie que si le fichier existe déjà, son contenu précédent doit être écrasé ;
  - **ios**::**app** (append / ajouter) : si le fichier existe déjà, ajouter les nouvelles données à la suite sans effacer les précécentes.

Dans les 2 cas, si le fichier n'existe pas encore, un nouveau fichier vide est créé automatiquement.

```
1 // Ouvre un fichier "data.txt" en écriture, auquel on
     pourra accéder par l'objet mon_fichier
2 ofstream mon_fichier("data.txt", ios::trunc);
```

Ecrire dans un fichier (3/4)

Vérifier l'ouverture du fichier : il faut ensuite vérifier que le fichier est correctement ouvert. Il existe pour cela une fonction **is_open()** qui retourne une valeur

- **true** si le fichier est correctement ouvert ;
- **false** si une erreur s'est produite lors de l'ouverture du fichier.

Cette fonction ne prend pas d'argument, et elle s'utilise avec un objet fichier précédemment déclaré de la manière suivante :

```
1 mon_fichier.is_open()
```

S'il s'avère que cette fonction échoue, le signaler à l'utilisateur au moyen d'un message et terminer le programme au moyen de

```
1 cout << "ERREUR: Impossible d'ouvrir le fichier" << endl;
2 return EXIT_FAILURE;
```

La constante EXIT_FAILURE est contenue dans **cstdlib**.

Ecrire dans un fichier (4/4)

Ecrire dans le fichier : l'écriture dans le fichier s'effectue au moyen de l'opérateur "«", comme pour cout.

```
1 mon_fichier << "Hello" << endl;
2 mon_fichier << "5 + 4 = " << 5 + 4 << endl;
```

De la même manière qu'avec cout, on peut fixer le nombre de chiffres significatifs à afficher avec

```
1 mon_fichier.precision(valeur);
```

Fermer le fichier : pour fermer le fichier on utilise la fonction **close()**, qui ne prend pas d'argument :

```
1 mon_fichier.close();
```

Ecrire dans un fichier : exemple

```
1  #include <fstream>
2  #include <cstdlib>
3  int main()
4  {
5    int age = 19;
6    ofstream mon_fichier("data.txt", ios::trunc);
7    if(!mon_fichier.is_open()) // fichier bien ouvert ?
8    {
9     cout << "ERREUR: Impossible d'ouvrir le fichier" << endl;
10    return EXIT_FAILURE;
11   }
12   // Le fichier est bien ouvert, on peut écrire
13   mon_fichier << "J'ai " << age << " ans." << endl;
14   mon_fichier.close(); // fermer le fichier
15
16
17   return 0;
18 }
```

Lire dans un fichier

Pour lire un fichier, la méthode est similaire.
Un objet de type **ifstream** est cette fois-ci utilisé, et le sens “»” des chevrons utilisés change, comme pour l'utilisation de cout et cin.
Les étapes pour lire dans un fichier sont

1. Ouvrir le fichier
2. Vérifier que le fichier est correctement ouvert
3. Lire les données du fichier
4. Fermer le fichier

Lire dans un fichier

Pour ouvrir un fichier en lecture, on déclare un objet **ifstream** avec les paramètres suivants :

ifstream nom_variable(“*chemin_vers_le_fichier*”) ;

- “*chemin_vers_le_fichier*” : une chaîne de caractères qui indique le nom du fichier à ouvrir ;

```
1 // Ouvre un fichier "data_input.txt" en lecture,
     auquel on pourra accéder par l'objet mon_fichier
2 ifstream mon_fichier("data_input.txt");
```

Lire dans un fichier

Pour ouvrir un fichier en lecture, on déclare un objet **ifstream** avec les paramètres suivants :

ifstream nom_variable(“*chemin_vers_le_fichier*”) ;

- “*chemin_vers_le_fichier*” : une chaîne de caractères qui indique le nom du fichier à ouvrir ;

```
1 // Ouvre un fichier "data_input.txt" en lecture,
     auquel on pourra accéder par l'objet mon_fichier
2 ifstream mon_fichier("data_input.txt");
```

Il faut ensuite vérifier que le fichier est correctement ouvert, grâce à la même fonction **is_open()** que pour les fichiers en écriture.

```
1 mon_fichier.is_open()
```

Il faut, le cas échéant, également avertir l'utilisateur d'un soucis à l'ouverture du fichier et terminer le programme.

Lire dans un fichier

Avant chaque opération de lecture, il faut ensuite vérifier que le fichier est toujours prêt à être lu grâce à la fonction **good()**. Cette fonction retourne une valeur

- **true** : si on peut encore lire dans le fichier ;
- **false** : si on ne peut plus lire dans le fichier, que ce soit parce qu'une erreur s'est produite, parce que la fin du fichier a été atteinte, etc...

```
1 mon_fichier.good()
```

Lire dans un fichier

Avant chaque opération de lecture, il faut ensuite vérifier que le fichier est toujours prêt à être lu grâce à la fonction **good()**. Cette fonction retourne une valeur

- **true** : si on peut encore lire dans le fichier ;
- **false** : si on ne peut plus lire dans le fichier, que ce soit parce qu'une erreur s'est produite, parce que la fin du fichier a été atteinte, etc...

```
1 mon_fichier.good()
```

On peut ensuite lire les données au moyen de l'opérateur “»” comme avec cin :

```
1 int a;
2 mon_fichier >> a;
```

Lire dans un fichier

Avant chaque opération de lecture, il faut ensuite vérifier que le fichier est toujours prêt à être lu grâce à la fonction **good()**. Cette fonction retourne une valeur

- **true** : si on peut encore lire dans le fichier ;
- **false** : si on ne peut plus lire dans le fichier, que ce soit parce qu'une erreur s'est produite, parce que la fin du fichier a été atteinte, etc...

```
1 mon_fichier.good()
```

On peut ensuite lire les données au moyen de l'opérateur "»" comme avec cin :

```
1 int a;
2 mon_fichier >> a;
```

Et pour finir, le fichier doit être fermé avec la fonction **close()** :

```
1 mon_fichier.close();
```

Lire dans un fichier : exemple

```
 1  #include <fstream>
 2  #include <cstdlib>
 3  int main()
 4  {
 5    ifstream mon_fichier("data_input.txt"); //Ouverture du fichier
 6    if(!mon_fichier.is_open()) // fichier ouvert ?
 7    {
 8      cout << "ERREUR: Impossible d'ouvrir le fichier" << endl;
 9      return EXIT_FAILURE;
10    }
11
12    while(mon_fichier.good()) // encore des données à lire ?
13    {
14       int a = 0;
15       mon_fichier >> a;
16       cout << "Le fichier contient " << a << endl;
17    }
18    mon_fichier.close(); // fermer le fichier
19    return 0;
20  }
```

Erreurs et débogage

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

Erreurs et débogage

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
  $\rightarrow$ Ce sont les erreurs de déclaration, de notations des instructions (points-virgules), ...

Erreurs et débogage

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
  $\rightarrow$ 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) $\rightarrow$ 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**).

Erreurs courantes

- 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

Erreurs courantes

- 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

Erreurs courantes

- 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

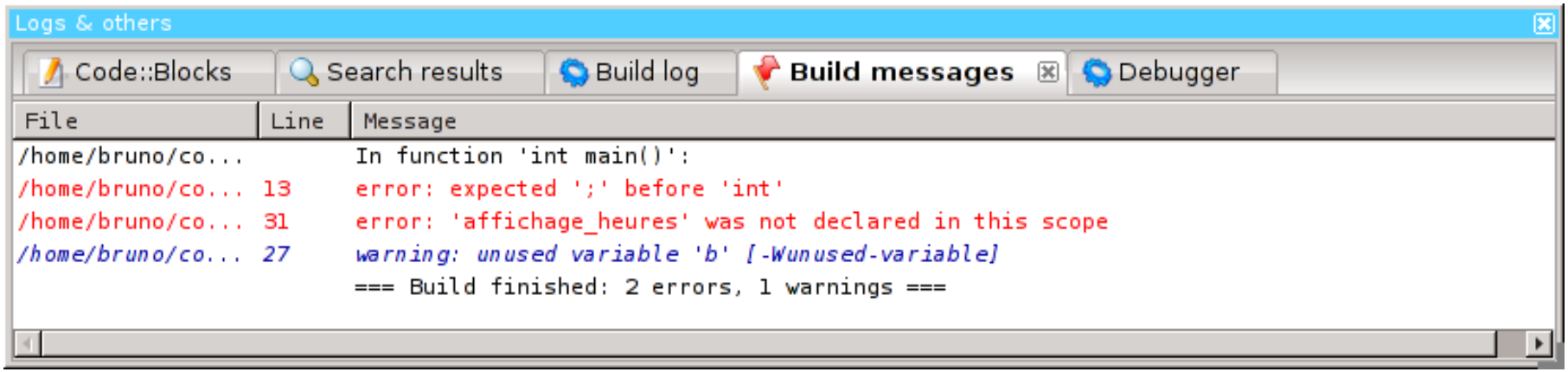
- Expression de comparaison :
  utiliser = (affectation) à la place de == (comparaison)
  → erreur sémantique

- 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 recontré.

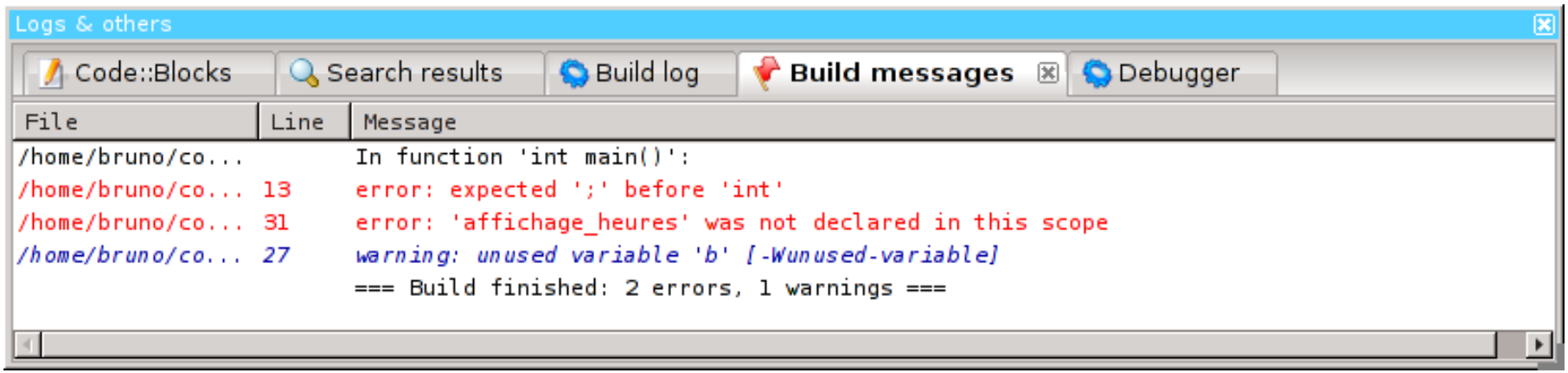
Il est important de commencer par lire et résoudre **les premières erreurs** pour commencer, car les erreurs suivantes peuvent être provoquées par les précédentes.



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 recontré.

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



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.

Erreurs sémantiques et débogage

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 ;

Erreurs sémantiques et débogage

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 ;

Erreurs sémantiques et débogage

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.

Déboguer un programme

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

Déboguer un programme

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

Utilisation du débogueur avec Code::Blocks (1/3)

Pour déboguer un programme :

- Placer le projet en configuration “Debug”

Utilisation du débogueur avec Code::Blocks (1/3)

Pour déboguer un programme :

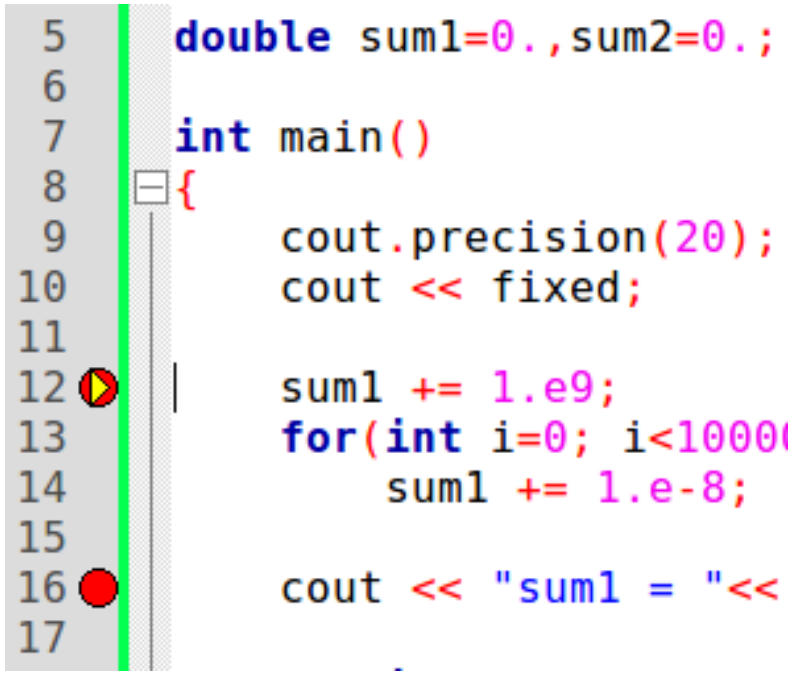
- Placer le projet en configuration “Debug”

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

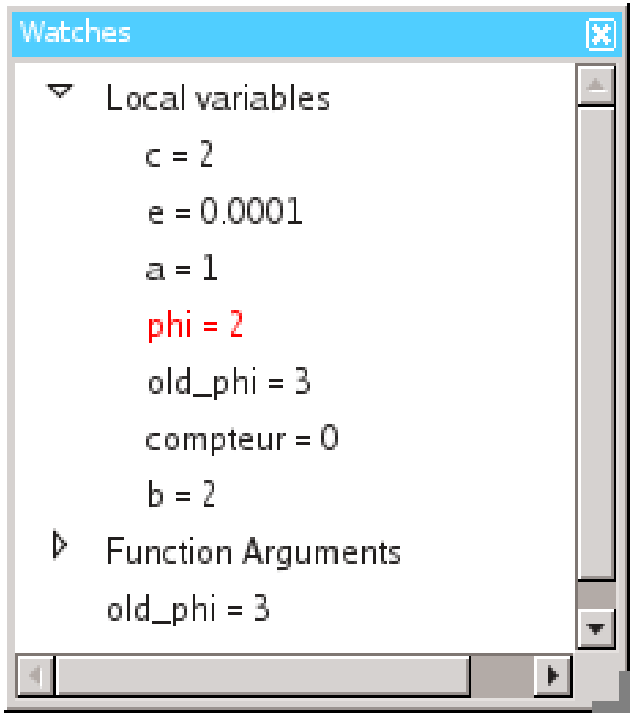
  - Le premier bouton sert à exécuter le programme jusqu'au point d'arrêt suivant
  - Le deuxième exécute le programme jusqu'à la position actuelle du curseur
  - Le troisième exécute la ligne de code suivante
  - etc...

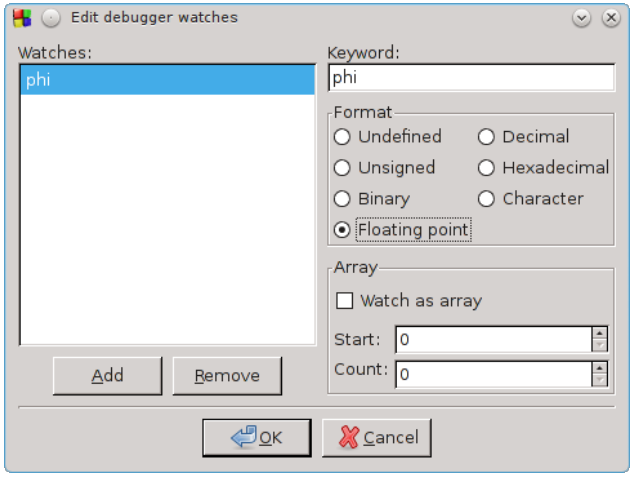
Utilisation du débogueur avec Code::Blocks (2/3)

- 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.



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 signifique que Code::Blocks affichera constamment sa valeur dans la fenêtre *Watches*.

Exercices

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 pouvant contenir 10 nombres entiers (les cotes) ;
   - Générer aléatoirement et stocker dans le tableau 10 cotes entre 0 et 20 ;
   - Afficher le tableau des cotes ;
   - Afficher la meilleure cote, la moins bonne cote et la moyenne ;
   - Demander à l'utilisateur une cote entre 0 et 20, afficher toutes les cotes supérieures à cette valeur et indiquer quel pourcentage des résultats cela représente.
3. Recherche dans un tableau
   - Créer un tableau de 50 éléments de type **double** ;
   - Charger les nombres contenus dans le fichier "`nombres.txt`" et les stocker dans le tableau ;
   - Afficher le tableau ;
   - Déterminer l'indice du plus petit élément du tableau ;
   - Afficher la valeur de cet élément et l'indice correspondant.

Exercices

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
   - Afficher le tableau non trié ;
   - Parcourir le tableau à la recherche du plus petit élément ;
   - Permuter cet élément avec le premier élément du tableau ;
   - 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) ;
   - Permuter cet élément avec le second du tableau ;
   - Poursuivre avec le 3^e, le 4^e élément, etc... jusqu'à ce que le tableau soit trié ;
   - Afficher le tableau trié.

5. Calcul d'un déterminant :
   - Charger le fichier "`matrice.txt`" contenant une matrice 3×3 dans un tableau de nombres réels ;
   - Afficher la matrice (sous forme carrée) dans la console ;
   - Calculer le déterminant de cette matrice ;
   - Afficher le résultat du calcul dans la console.

Exercices

6. Calcul de π par une série (formule de Leibniz) :
   Soit la série qui converge vers $\frac{\pi}{4}$,

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

   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} puis 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 ;
   - Ecrire chacun de ces résultats dans un fichier "`approx_pi.txt`", ligne par ligne, sous la forme "PI = 3.1415... (avec x termes)".

Exercices

7. Calcul de π par dichotomie :
   Soit la fonction $f(x) = \cos(x)$ considéréé 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, déterminer dans lequel des deux se trouve le zéro, et ainsi de suite 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 :

   7.1 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 $[c; b]$

   7.2 Recommencer l'étape (1) avec le nouvel intervalle à considérer ($[a; c]$ ou $[c; b]$) ;

   7.3 Procéder ainsi jusqu'à ce que l'intervalle $[a; b]$ soit réduit à la précision voulue e. Tester avec $e = 10^{-9}$;

   7.4 En déduire la valeur de π ;

   7.5 Ouvrir à nouveau le précédent fichier "`approx_pi.txt`" et y ***ajouter*** une ligne contenant "PI = 3.1415... (dichotomie)" à la suite des résultats précédents.