

Introduction à la programmation

Travaux pratiques: séance 5

INFO0201-1

X. Baumans

(xavier.baumans@ulg.ac.be)

[Copyright © F. Ludewig & B. Baert, ULg]



Programme de la séance

- Rappels structures de contrôle itératives
→ **while**, **do...while**, **for**

Programme de la séance

- Rappels structures de contrôle itératives
→ **while**, **do...while**, **for**
- Rappel et synthèse des règles de bonne pratique

Programme de la séance

- Rappels structures de contrôle itératives
→ **while, do...while, for**
- Rappel et synthèse des règles de bonne pratique
- Erreurs et débogage d'un programme
→ comment repérer et corriger les erreurs d'un programme

- Rappels structures de contrôle itératives
→ **while, do...while, for**
- Rappel et synthèse des règles de bonne pratique
- Erreurs et débogage d'un programme
→ comment repérer et corriger les erreurs d'un programme
 - Erreurs syntaxiques

Programme de la séance

- Rappels structures de contrôle itératives
→ **while, do...while, for**
- Rappel et synthèse des règles de bonne pratique
- Erreurs et débogage d'un programme
→ comment repérer et corriger les erreurs d'un programme
 - Erreurs syntaxiques
 - Erreurs sémantiques

Programme de la séance

- Rappels structures de contrôle itératives
→ **while, do...while, for**
- Rappel et synthèse des règles de bonne pratique
- Erreurs et débogage d'un programme
→ comment repérer et corriger les erreurs d'un programme
 - Erreurs syntaxiques
 - Erreurs sémantiques
 - Utilisation du débogueur

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```


Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```

Structure de contrôle itérative : boucle while

```
1 while(condition)
2 {
3     instructions;
4 }
```

Boucle réalisée "tant que" la condition est vérifiée (vraie).
Attention à bien influencer sur la condition dans les instructions sinon :
boucle infinie!

Un exemple simple :

```
1 int i = 0;
2 while(i < 5)
3 {
4     cout << "La valeur de i est : " << i << endl;
5     i++; // --> i = i + 1;
6 }
```


Structure de contrôle itérative : boucle do...while

```
1 do
2 {
3     instructions;
4
5 }while(condition);
```

Structure indiquée quand on veut exécuter le bloc d'instruction au moins une fois avant de vérifier la condition :

```
1 int nombre;
2 do
3 {
4     cout << "Entrez un nombre positif :" << endl;
5     cin >> nombre;
6 } while(nombre < 0);
7 cout << "Le nombre positif est " << nombre << endl;
```

Structure de contrôle itérative : boucle for

```
1 for(initialisation; condition; itération)
2 {
3     instructions;
4 }
```

Son utilisation est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Exemple simple : compter jusque 10

```
1 for(int c=0; c <= 10; c++)
2 {
3     cout << "La valeur de c est " << c << endl;
4 }
```

Règles de bonne pratique :

- **Réfléchir à la structure du programme avant de coder !**
Réfléchir d'abord aux différentes étapes nécessaires pour résoudre le problème : quelles variables seront nécessaires, faut-il des boucles, des tests conditionnels, etc...
Idéalement, écrire le plan du programme sur papier !

Règles de bonne pratique :

- **Réfléchir à la structure du programme avant de coder !**
Réfléchir d'abord aux différentes étapes nécessaires pour résoudre le problème : quelles variables seront nécessaires, faut-il des boucles, des tests conditionnels, etc...
Idéalement, écrire le plan du programme sur papier !
- **Indenter correctement le code !**
L'alignement des blocs de code rend le programme beaucoup plus lisible et structuré. Il permet aussi de repérer facilement des accolades “{” manquantes. Utiliser la commande automatique “Plugins/Source code formatter” de Code::Blocks.
Cela est valable pour l'examen aussi !

Règles de bonne pratique

- **Commenter le code un maximum !**

Tout le monde doit pouvoir comprendre à quoi sert le code sans refaire le raisonnement complet.

Cela est valable pour l'examen aussi !

Règles de bonne pratique

- **Commenter le code un maximum !**

Tout le monde doit pouvoir comprendre à quoi sert le code sans refaire le raisonnement complet.

Cela est valable pour l'examen aussi !

- **Lire les erreurs du compilateur**

Lorsqu'une erreur se produit à la compilation, le compilateur renvoie un message d'erreur contenant la ligne du code source à laquelle l'erreur a été détectée et la raison de l'erreur.

Attention de toujours commencer par la première erreur, la corriger, puis tenter de recompiler. En effet, les erreurs suivantes peuvent découler de la première et disparaître lorsque celle-ci a été corrigée.

→ La moindre des choses est de rendre un programme qui compile !

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 (erreurs de compilation) 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 (erreurs de compilation) 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 possibilités alors :

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 (erreurs de compilation) 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 possibilités alors :
 - Le programme s'arrête avec un message d'erreur du type "Segmentation Fault" ;

Erreurs et débogages

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 (erreurs de compilation) 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 possibilités alors :
 - Le programme s'arrête avec un message d'erreur du type "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

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

Erreurs syntaxiques

Lorsque des erreurs **syntaxiques** sont présentes, le compilateur

- les énumère en fournissant le numéro de la ligne correspondante

Erreurs syntaxiques

Lorsque des erreurs **syntaxiques** sont présentes, le compilateur

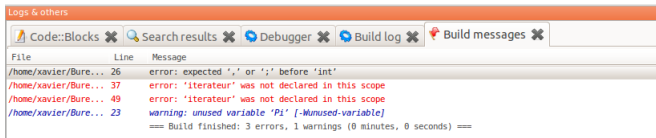
- les énumère en fournissant le numéro de la ligne correspondante
- fournit également un bref descriptif du problème rencontré

Erreurs syntaxiques

Lorsque des erreurs **syntaxiques** sont présentes, le compilateur

- les énumère en fournissant le numéro de la ligne correspondante
- fournit également un bref descriptif du problème rencontré

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



The screenshot shows a 'Logs & others' window with several tabs: 'Code::Blocks', 'Search results', 'Debugger', 'Build log', and 'Build messages'. The 'Build log' tab is active, displaying a table of build output. The table has two columns: 'File' and 'Message'. The messages include three errors and one warning, all related to syntax in a file at '/home/xavier/Bure...'. The errors are 'error: expected ',' or ';' before 'int'', 'error: 'iterateur' was not declared in this scope', and another 'error: 'iterateur' was not declared in this scope'. The warning is 'warning: unused variable 'Pi' [-Wunused-variable]'. At the bottom of the log, it says '=== Build finished: 3 errors, 1 warnings (0 minutes, 0 seconds) ==='.

File	Line	Message
/home/xavier/Bure...	26	error: expected ',' or ';' before 'int'
/home/xavier/Bure...	37	error: 'iterateur' was not declared in this scope
/home/xavier/Bure...	49	error: 'iterateur' was not declared in this scope
/home/xavier/Bure...	23	warning: unused variable 'Pi' [-Wunused-variable]

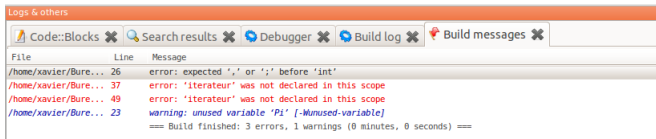
=== Build finished: 3 errors, 1 warnings (0 minutes, 0 seconds) ===

Erreurs syntaxiques

Lorsque des erreurs **syntaxiques** sont présentes, le compilateur

- les énumère en fournissant le numéro de la ligne correspondante
- fournit également un bref descriptif du problème rencontré

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



```
Logs & others
Code::Blocks x Search results x Debugger x Build log x Build messages x
File Line Message
/home/xavier/Bure... 26 error: expected ',' or ';' before 'int'
/home/xavier/Bure... 37 error: 'iterateur' was not declared in this scope
/home/xavier/Bure... 49 error: 'iterateur' was not declared in this scope
/home/xavier/Bure... 23 warning: unused variable 'Pi' [-Wunused-variable]
== Build finished: 3 errors, 1 warnings (0 minutes, 0 seconds) ==
```

Le compilateur fournit également des messages d'alerte ("warnings"). Ceux-ci n'empêchent pas la compilation mais mettent en lumière des pratiques non recommandées. Il est donc **fortement conseillé** de les corriger car ils pourraient mener à des erreurs.

Erreurs sémantiques et débogage

Les erreurs sémantiques ne sont pas directement visibles et sont donc **difficiles à détecter**. Personne n'est à l'abri d'en commettre, l'important est de pouvoir les repérer et les corriger.
Conseils :

Les erreurs sémantiques ne sont pas directement visibles et sont donc **difficiles à détecter**. Personne n'est à l'abri d'en commettre, l'important est 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**. Personne n'est à l'abri d'en commettre, l'important est 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 affichages des 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**. Personne n'est à l'abri d'en commettre, l'important est 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 affichages des 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 clefs

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 clefs
 - Etablir un affichage clair et précis

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 clefs
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles

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 clefs
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et aux 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 clefs
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et aux erreurs simples
- 2 Utiliser le débogueur
Code : :Blocks dispose d'une interface intégrée avec un débogueur. Il permet :

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 clefs
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et aux 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

Débugger 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 clefs
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et aux 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

Débugger 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 clefs
 - Etablir un affichage clair et précis
 - Ne pas multiplier les informations inutiles
 - Technique limitée à de petits programmes et aux 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

Exemple de détection par affichage

- Erreurs de comparaison : afficher la variable avant et après le test car celle-ci ne doit pas être modifiée par le test de comparaison (`==`) mais en cas d'utilisation d'un mauvais opérateur (`=`), elle le sera ;

Exemple de détection par affichage

- Erreurs de comparaison : afficher la variable avant et après le test car celle-ci ne doit pas être modifiée par le test de comparaison (`==`) mais en cas d'utilisation d'un mauvais opérateur (`=`), elle le sera ;
- Erreur de type de variable : le plus souvent un nombre à virgule est attendu mais le programme retourne un nombre entier ou zéro.
Par exemple : $1/2=0$, $5/2 = 2$ mais $5./2 = 2.5$

Exemple de détection par affichage

- Erreurs de comparaison : afficher la variable avant et après le test car celle-ci ne doit pas être modifiée par le test de comparaison (`==`) mais en cas d'utilisation d'un mauvais opérateur (`=`), elle le sera ;
- Erreur de type de variable : le plus souvent un nombre à virgule est attendu mais le programme retourne un nombre entier ou zéro.
Par exemple : $1/2=0$, $5/2 = 2$ mais $5./2 = 2.5$
- Les boucles infinies : afficher la condition de contrôle de la boucle (et les éléments qui la constituent si nécessaire) à chaque itération. De cette manière, il est possible de rechercher à quel moment la condition aurait dû faire aboutir la boucle.

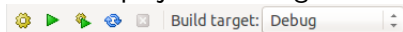
Exemple de détection par affichage

- Erreurs de comparaison : afficher la variable avant et après le test car celle-ci ne doit pas être modifiée par le test de comparaison (`==`) mais en cas d'utilisation d'un mauvais opérateur (`=`), elle le sera ;
- Erreur de type de variable : le plus souvent un nombre à virgule est attendu mais le programme retourne un nombre entier ou zéro.
Par exemple : $1/2=0$, $5/2 = 2$ mais $5./2 = 2.5$
- Les boucles infinies : afficher la condition de contrôle de la boucle (et les éléments qui la constituent si nécessaire) à chaque itération. De cette manière, il est possible de rechercher à quel moment la condition aurait dû faire aboutir la boucle.
- Les boucles jamais exécutées : vérifier si la boucle est exécutée en ajoutant l'affichage d'un message à chaque itération de la boucle. Si le message n'apparaît jamais, c'est que la boucle n'est pas exécutée.

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

Pour déboguer un programme :

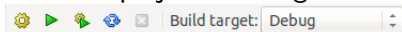
- Placer le projet en configuration "Debug"



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 :

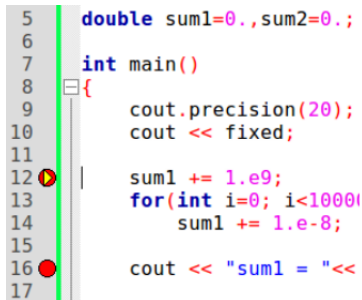


- Le premier bouton sert à exécuter le programme jusqu'au point d'arrêt suivant
- Le deuxième exécute le programme jusqu'à la position actuelle du curseur
- Le troisième exécute la ligne de code suivante ;
- etc...

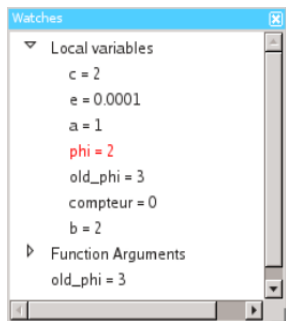
Utilisation du débogueur avec Code : :Blocks (2/3)

- Pour ajouter un *point d'arrêt*, il suffit de cliquer dans la gouttière (entre le code et le numéro de la 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
14        sum1 += 1.e-8;
15
16    cout << "sum1 = "<<
17
```

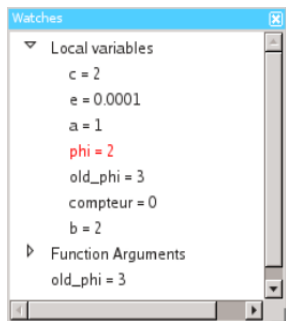


Utilisation du débogueur avec Code : :Blocks (3/3)



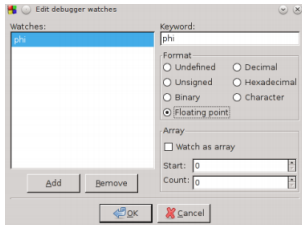
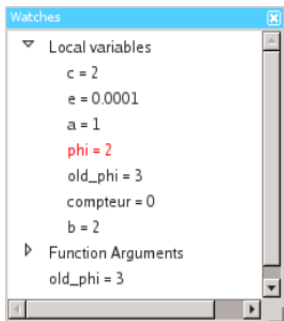
- La fenêtre *Watches* permet d'afficher les valeurs actuelles de toutes les variables locales.

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.

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 :
 - Charger le fichier main.cpp fourni avec le TP ;
 - L'inclure dans un nouveau projet ;
 - Lire et s'attacher à comprendre ce que le programme tente de faire ;
 - Corriger les erreurs à l'aide des informations données par le compilateur et le débogueur.
- 2 Suite des exercices de la séance 4... (terminer au moins le 4 et le 5!)