Introduction à la programmation
Travaux pratiques: séance 2
INFO0201-1

B. Baert, X. Baumans & F. Ludewig
Bruno.Baert@ulg.ac.be - Xavier.Baumans@ulg.ac.be

Programme de la séance

- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  → expressions condensées, code plus concis

Programme de la séance

- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  → expressions condensées, code plus concis
- Structure de contrôle conditionnelle : l'instruction **switch**
  → nombreux cas différents (plus concis que le **if**)

Programme de la séance

- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  → expressions condensées, code plus concis
- Structure de contrôle conditionnelle : l'instruction **switch**
  → nombreux cas différents (plus concis que le **if**)
- Nombres aléatoires
  → génération de nombres pseudo-aléatoires

Programme de la séance

- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  → expressions condensées, code plus concis
- Structure de contrôle conditionnelle : l'instruction **switch**
  → nombreux cas différents (plus concis que le **if**)
- Nombres aléatoires
  → génération de nombres pseudo-aléatoires
- Structures de contrôle itératives : les boucles **for**, **while**, ...
  → exécution d'instructions répétitives

Programme de la séance

- Opérateurs d'affectation et unaires (+=, -=, ++, −−, ...)
  → expressions condensées, code plus concis
- Structure de contrôle conditionnelle : l'instruction **switch**
  → nombreux cas différents (plus concis que le **if**)
- Nombres aléatoires
  → génération de nombres pseudo-aléatoires
- Structures de contrôle itératives : les boucles **for**, **while**, ...
  → exécution d'instructions répétitives
- Précision limitée du stockage des nombres réels
  → précautions à prendre dans l'utilisation de ces nombres

Opérateurs d'affectation condensés

Il existe une série d'opérateurs qui permettent de réaliser une **opération** avec une variable et de remplacer immédiatement sa **valeur** par le **résultat** de l'opération.

Opérateurs d'affectation condensés

Il existe une série d'opérateurs qui permettent de réaliser une **opération** avec une variable et de remplacer immédiatement sa **valeur** par le **résultat** de l'opération.
Exemple avec l'addition :

```
1 int a = 2;
2 int b = 5;
3 a += b; // --> a = a + b
4 cout << "a = " << a << endl; // affiche 7
```

Opérateurs d'affectation condensés

Il existe une série d'opérateurs qui permettent de réaliser une **opération** avec une variable et de remplacer immédiatement sa **valeur** par le **résultat** de l'opération.
Exemple avec l'addition :

```
1 int a = 2;
2 int b = 5;
3 a += b; // --> a = a + b
4 cout << "a = " << a << endl; // affiche 7
```

De la même manière, on peut utiliser les opérateurs suivants :

+=	a+=b	$a = a + b$
-=	a-=b	$a = a - b$
*=	a*=b	$a = a * b$
/=	a/=b	$a = a/b$
%=	a%=b	$a = a\%b$

Opérateurs unaires

Il existe en C++ des **opérateurs** qui ne nécessitent qu'un seul **opérande** : ce sont les **opérateurs unaires** $++$ et $--$.

Ceux-ci servent respectivement à **incrémenter** et **décrémenter** la valeur d'une variable. Ceux-ci ont également un comportement différent selon qu'ils sont utilisés sous leur forme **préfixée** ou **suffixée**.

Opérateurs unaires

Il existe en C++ des **opérateurs** qui ne nécessitent qu'un seul **opérande** : ce sont les **opérateurs unaires** $++$ et $--$.

Ceux-ci servent respectivement à **incrémenter** et **décrémenter** la valeur d'une variable. Ceux-ci ont également un comportement différent selon qu'ils sont utilisés sous leur forme **préfixée** ou **suffixée**.

```
1 int a = 2, b = 0;
2 a++; // --> a = 3
3 b = a++; // --> b = 3, a = 4
4 b = ++a; // --> b = 5, a = 5
```

Opérateurs unaires

Il existe en C++ des **opérateurs** qui ne nécessitent qu'un seul **opérande** : ce sont les **opérateurs unaires** $++$ et $--$.

Ceux-ci servent respectivement à **incrémenter** et **décrémenter** la valeur d'une variable. Ceux-ci ont également un comportement différent selon qu'ils sont utilisés sous leur forme **préfixée** ou **suffixée**.

```
1 int a = 2, b = 0;
2 a++; // --> a = 3
3 b = a++; // --> b = 3, a = 4
4 b = ++a; // --> b = 5, a = 5
```

Lorsque l'opérateur est **préfixé**, l'opération est d'abord effectuée sur l'opérande et le résultat est utilisé dans l'expression correspondante.

Lorsque l'opérateur est **suffixé**, la valeur actuelle de l'opérande est utilisée dans l'expression puis l'opération est appliquée à l'opérande.

Le type char

Le type **char** permet de stocker un **caractère** dans une variable.

En interne, il s'agit en fait d'un petit entier, permettant de stocker des valeurs comprises entre -128 et $+127$. Les valeurs numériques correspondent au code ASCII du caractère et peuvent ainsi représenter les différents caractères.
Exemple : le caractère "A" possède le code ASCII 64.

```
1 char caractere = 'A'; // caractère A == 64
2 char c = 97; // caractère a == 97
```

ATTENTION : L'affectation à un type **char** s'effectue avec des guillemets simples ('), contrairement au texte (chaînes de caractères) qui s'écrit avec des guillemets doubles (").

```
1 char g = "B"; // ERREUR, cela ne compilera pas !
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (1/4)

L'instruction **switch** teste plusieurs valeurs d'une expression de manière plus concise qu'avec de nombreux **if...else** imbriqués.

```
1  int a = 3;
2  switch(a)
3  {
4    case 1: // si a == 1
5      cout << "a vaut 1" << endl;
6      break;
7    case 2: // si a == 2
8    case 3: // si a == 3
9      cout << "a vaut 2 ou 3" << endl;
10     break;
11   default: // dans tous les autres cas
12     cout << "a ne vaut pas 1, 2 ou 3" << endl;
13     break;
14 }
```

Structure de contrôle conditionnelle : switch...case (2/4)

- Chaque valeur à tester est introduite par le mot-clé **case** :

```
1   case 5:
```

Structure de contrôle conditionnelle : switch...case (2/4)

- Chaque valeur à tester est introduite par le mot-clé **case** :

```
1   case 5:
```

- La fin des instructions à exécuter après vérification d'un cas est signalée par le mot-clé **break**

```
1   case 5:
2     // instructions
3     break;
```

Structure de contrôle conditionnelle : switch...case (2/4)

- Chaque valeur à tester est introduite par le mot-clé **case** :

```
1   case 5:
```

- La fin des instructions à exécuter après vérification d'un cas est signalée par le mot-clé **break**

```
1   case 5:
2     // instructions
3     break;
```

En l'absence de **break**, l'exécution des instructions suivantes se poursuit.

```
1   case 5:
2     // instructions
3   case 6:
4     // instructions exécutées dans les cas 6 ET 5
5     break;
```

Structure de contrôle conditionnelle : switch...case (3/4)

Un mot-clé spécifique, **default**, permet de couvrir tous les cas restants :

```
1  switch(a)
2  {
3    case 1:
4      // instructions pour a == 1
5      break;
6    case 2:
7      // instructions pour a == 2
8      break;
9    default:
10     // instructions pour les autres cas
11     // (i.e. a!=1 && a!=2)
12     break;
13 }
```

Structure de contrôle conditionnelle : switch...case (4/4)

On peut aussi tester des caractères alphanumériques (type **char**).

```
1  switch(a) // a est de type 'char'
2  {
3    case 'a':
4      cout << "ceci est un a" << endl;
5      break;
6    case 'b':
7      cout << "ceci est un b" << endl;
8      break;
9    default:
10     cout << "une lettre indéterminée" << endl;
11     break;
12 }
```

Nombres aléatoires

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>
```

Nombres aléatoires : manipulations

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

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

Nombres aléatoires : manipulations

- 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 libraire <**ctime**>

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Le premier type de boucle est la boucle **while**.
Sa syntaxe est la suivante : **while**(*condition*) {*instructions ;*}

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Le premier type de boucle est la boucle **while**.
Sa syntaxe est la suivante : **while**(*condition*) {*instructions ;*}

- *condition* est une **expression booléenne** qui est évaluée ;

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Le premier type de boucle est la boucle **while**.
Sa syntaxe est la suivante : **while**(*condition*) {*instructions ;*}

- *condition* est une **expression booléenne** qui est évaluée ;
- si le résultat est **true**, les *instructions* sont exécutées ;

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Le premier type de boucle est la boucle **while**.
Sa syntaxe est la suivante : **while**(*condition*) {*instructions ;*}

- *condition* est une **expression booléenne** qui est évaluée ;
- si le résultat est **true**, les *instructions* sont exécutées ;
- ensuite, *condition* est à nouveau évaluée ;

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Le premier type de boucle est la boucle **while**.
Sa syntaxe est la suivante : **while**(*condition*) {*instructions ;*}

- *condition* est une **expression booléenne** qui est évaluée ;
- si le résultat est **true**, les *instructions* sont exécutées ;
- ensuite, *condition* est à nouveau évaluée ;
- si le résultat est encore **true**, les *instructions* sont à nouveau exécutées ;

Structure de contrôle itérative : boucle while (1/4)

Il est parfois nécessaire d'**exécuter plusieurs fois** une même instruction. Il existe pour cela des structures de contrôles itératives, les **boucles**.

Le premier type de boucle est la boucle **while**.
Sa syntaxe est la suivante : **while**(*condition*) {*instructions ;*}

- *condition* est une **expression booléenne** qui est évaluée ;
- si le résultat est **true**, les *instructions* sont exécutées ;
- ensuite, *condition* est à nouveau évaluée ;
- si le résultat est encore **true**, les *instructions* sont à nouveau exécutées ;
- cette situation se répète jusqu'à ce que *condition* soit évalué à la valeur **false**.

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (2/4)

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 }
```

Ce code va afficher :

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
```

Structure de contrôle itérative : boucle while (3/4)

Autre exemple : calculer la série (qui tend vers 2 pour $k \to \infty$)

$$\sum_{k=0}^{\infty} \frac{1}{2^k} \text{ tant que } \frac{1}{2^k} > 0.05$$

Structure de contrôle itérative : boucle while (3/4)

Autre exemple : calculer la série (qui tend vers 2 pour $k \to \infty$)

$$\sum_{k=0}^{\infty} \frac{1}{2^k} \text{ tant que } \frac{1}{2^k} > 0.05$$

```
1 double somme = 0.;
2 double i = 1.;
3 while(i > 0.05)
4 {
5   somme += i;
6   i/=2.;
7   cout << "valeur = " << somme << endl;
8 }
```

```
valeur = 1
valeur = 1.5
valeur = 1.75
valeur = 1.875
```

Structure de contrôle itérative : boucle while (4/4)

ATTENTION : Si la condition est mal exprimée, la boucle peut tourner un nombre infini de fois ! Le programme est alors bloqué.

```
1 int i = 5;
2 while(i != 2) // mal exprimé !
3                // "i <= 2" aurait évité l'erreur
4 {
5  i = i + 1; // Cette boucle va se répéter sans fin
6 }
```

Structure de contrôle itérative : boucle while (4/4)

ATTENTION : Si la condition est mal exprimée, la boucle peut tourner un nombre infini de fois ! Le programme est alors bloqué.

```
1 int i = 5;
2 while(i != 2) // mal exprimé !
3                // "i <= 2" aurait évité l'erreur
4 {
5  i = i + 1; // Cette boucle va se répéter sans fin
6 }
```

Lorsque l'on se trouve dans une telle situation, où le programme est bloqué par une **boucle infinie**, la combinaison de touches **Ctrl-C** permet d'interrompre brutalement le programme et de résoudre le problème.

Structure de contrôle itérative : boucle do...while (1/3)

Exemple : saisir une valeur positive au clavier et le vérifier

```
1  int nombre;
2  cout << "Entrez un nombre positif :" << endl;
3  cin >> nombre;
4  while(nombre < 0)
5  {
6    cout << "Entrez un nombre positif :" << endl;
7    cin >> nombre;
8  }
9  cout << "Le nombre positif est " << nombre << endl;
```

Pour pouvoir écrire la condition “nombre < 0”, il faut d'abord saisir une valeur au clavier, ce qui n'est pas pratique ni élégant. En effet, les mêmes instructions sont écrites deux fois alors que le but des structures de boucles est justement d'éviter cela.

Structure de contrôle itérative : boucle do...while (2/3)

Pour remédier à ce problème, on peut utiliser une structure de boucle légèrement différente : la boucle **do...while**

Sa syntaxe est la suivante : **do** {*instructions ;* } **while** (*condition*) ;

Structure de contrôle itérative : boucle do...while (2/3)

Pour remédier à ce problème, on peut utiliser une structure de boucle légèrement différente : la boucle **do...while**

Sa syntaxe est la suivante : **do** {*instructions ;* } **while** (*condition*) ;

Avec cette structure de boucle, les *instructions* sont d'abord exécutées. Ensuite la *condition* est évaluée, et si elle est évaluée à la valeur **true**, les *instructions* sont exécutées à nouveau.

Structure de contrôle itérative : boucle do...while (2/3)

Pour remédier à ce problème, on peut utiliser une structure de boucle légèrement différente : la boucle **do...while**

Sa syntaxe est la suivante : **do** {*instructions ;* } **while** (*condition*) ;

Avec cette structure de boucle, les *instructions* sont d'abord exécutées. Ensuite la *condition* est évaluée, et si elle est évaluée à la valeur **true**, les *instructions* sont exécutées à nouveau.

Cette structure implique que les *instructions* sont toujours exécutées **au moins une fois**.

Structure de contrôle itérative : boucle do...while (3/3)

L'exemple précédent peut ainsi se ré-écrire

```
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;
```

Les instructions effectuant la saisie du nombre au clavier sont ainsi exécutées une première fois avant l'évaluation de la condition.

Ce code a exactement la même fonction que le précédent, mais il est plus concis et plus lisible.

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

La séquence d'évènements se déroule ainsi :

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

La séquence d'évènements se déroule ainsi :

1. *initialisation* est évaluée ;

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

La séquence d'évènements se déroule ainsi :

1. *initialisation* est évaluée ;
2. *condition* est évaluée. Si le résultat est **false**, fin de la boucle ;

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

La séquence d'évènements se déroule ainsi :

1. *initialisation* est évaluée ;
2. *condition* est évaluée. Si le résultat est **false**, fin de la boucle ;
3. sinon, les *instructions* sont exécutées ;

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

La séquence d'évènements se déroule ainsi :

1. *initialisation* est évaluée ;
2. *condition* est évaluée. Si le résultat est **false**, fin de la boucle ;
3. sinon, les *instructions* sont exécutées ;
4. *itération* est évaluée ;

Structure de contrôle itérative : boucle for (1/2)

Dernier type de boucle : la **boucle for** est utilisée pour répéter des instructions un certain nombre déterminé de fois.

Son utilisation (contrairement aux autre boucles **while** et **do...while**) est particulièrement indiquée lorsque le nombre d'itérations est connu ou peut être calculé facilement.

Sa syntaxe est la suivante :
for(*initialisation* ; *condition* ; *itération*) { *instructions ;* }

La séquence d'évènements se déroule ainsi :

1. *initialisation* est évaluée ;
2. *condition* est évaluée. Si le résultat est **false**, fin de la boucle ;
3. sinon, les *instructions* sont exécutées ;
4. *itération* est évaluée ;
5. retour au point 2.

Structure de contrôle itérative : boucle for (2/2)

Exemple : compter jusque 10

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

Structure de contrôle itérative : boucle for (2/2)

Exemple : compter jusque 10

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

Autre exemple : calculer la factorielle d'un nombre

```
1 int factorielle = 1;
2 int nombre = 5; // calculer la factorielle de 5
3 for(int i=nombre; i > 1; i--)
4 {
5   factorielle *= i;
6 }
7 cout << "La factorielle de " << nombre;
8 cout << " vaut " << factorielle << endl;
```

Equivalence des types de boucles

Toutes les structures itératives (**while**, **do..while** et **for**) sont interchangeables. Certaines sont cependant plus faciles ou intuitives à utiliser dans certains cas.
Exemple : compter de 2 à 7

Equivalence des types de boucles

Toutes les structures itératives (**while**, **do..while** et **for**) sont interchangeables. Certaines sont cependant plus faciles ou intuitives à utiliser dans certains cas.

Exemple : compter de 2 à 7

```
1 for(int a=2; a <= 7; a++)
2 {
3   cout << a << endl;
4 }
```

Equivalence des types de boucles

Toutes les structures itératives (**while**, **do..while** et **for**) sont interchangeables. Certaines sont cependant plus faciles ou intuitives à utiliser dans certains cas.

Exemple : compter de 2 à 7

```
1 for(int a=2; a <= 7; a++)
2 {
3   cout << a << endl;
4 }
```

```
1 int a = 2;
2 while(a <= 7)
3 {
4   cout << a << endl;
5   a++;
6 }
```

Equivalence des types de boucles

```
1 int a = 2;
2 do
3 {
4   cout << a << endl;
5   a++;
6 } while (a <= 7);
```

Equivalence des types de boucles

```
1 int a = 2;
2 do
3 {
4   cout << a << endl;
5   a++;
6 } while (a <= 7);
```

Ces trois exemples de code aboutissent au même résultat, pour autant que l'initialisation et les conditions soient adaptées au type de boucle utilisée.

- La **boucle for** est particulièrement indiquée lorsque l'on connait le nombre d'itérations, puisque les initialisations et incrémentations sont prévues ;
- La boucle **do...while** permet de pallier à certaines difficultés qui peuvent être rencontrées avec la boucle **while** lorsqu'une première itération est nécessaire avant de pouvoir évaluer la condition.

Précision limitée des nombres réels (float, double)

Les nombres stockés dans une variable de type réel ont une précision limitée ($\sim$ 6 chiffres significatifs pour les **float** et $\sim$ 15 pour les **double**). Ainsi, même si l'on peut par exemple stocker des nombres très **grands**, ils sont en fait **approximés**.

Précision limitée des nombres réels (float, double)

Les nombres stockés dans une variable de type réel ont une précision limitée ($\sim$ 6 chiffres significatifs pour les **float** et $\sim$ 15 pour les **double**). Ainsi, même si l'on peut par exemple stocker des nombres très **grands**, ils sont en fait **approximés**.

Exemple : $10^{10} + 1 - 10^{10}$

Avec une précision totale : $(10^{10} - 10^{10}) + 1 = 0 + 1 = 1$

En C++, avec des floats : $(10^{10} + 1) - 10^{10} = 10^{10} - 10^{10} = 0$

$$\underbrace{1.000000}_{\text{6 chiffres}}0000 \times 10^{10}$$

$$\underbrace{1.000000}_{\text{6 chiffres}}0001 \times 10^{10}$$

Précision limitée des nombres réels (float, double)

Les nombres stockés dans une variable de type réel ont une précision limitée ($\sim$ 6 chiffres significatifs pour les **float** et $\sim$ 15 pour les **double**). Ainsi, même si l'on peut par exemple stocker des nombres très **grands**, ils sont en fait **approximés**.
Exemple : $10^{10} + 1 - 10^{10}$
Avec une précision totale : $(10^{10} - 10^{10}) + 1 = 0 + 1 = 1$
En C++, avec des floats : $(10^{10} + 1) - 10^{10} = 10^{10} - 10^{10} = 0$

$$\underbrace{1.000000}_{\text{6 chiffres}}0000 \text{ x } 10^{10}$$

$$1.000000\overbrace{0001}^{\text{6 chiffres}} \text{ x } 10^{10}$$

$\rightarrow 10^{10} + 1$ est donc approximé à la même valeur que 10^{10} lorsqu'il est stocké dans une variable de type **float**.

Précision limitée des nombres réels (float, double)

Les nombres stockés dans une variable de type réel ont une précision limitée ($\sim$ 6 chiffres significatifs pour les **float** et $\sim$ 15 pour les **double**). Ainsi, même si l'on peut par exemple stocker des nombres très **grands**, ils sont en fait **approximés**.
Exemple : $10^{10} + 1 - 10^{10}$
Avec une précision totale : $(10^{10} - 10^{10}) + 1 = 0 + 1 = 1$
En C++, avec des floats : $(10^{10} + 1) - 10^{10} = 10^{10} - 10^{10} = 0$

$$\underbrace{1.000000}_{\text{6 chiffres}}0000 \text{ x } 10^{10}$$

$$\underbrace{1.000000}_{\text{6 chiffres}}0001 \text{ x } 10^{10}$$

$\rightarrow 10^{10} + 1$ est donc approximé à la même valeur que 10^{10} lorsqu'il est stocké dans une variable de type **float**.
Dans ce cas précis, des variables de type **double** auraient permis de stocker ces nombres avec la précision suffisante. La limite du type **double** est cependant facilement atteinte également $(10^{20} + 1)$

Précision limitée des nombres réels (float, double)

La précision limitée des nombres réels implique également que les **comparaisons d'égalité entre ceux-ci sont à éviter**.
En effet, deux nombres qui seraient égaux avec une précision totale peuvent ne pas l'être à cause des erreurs d'approximation dûes à la précision limitée.

Précision limitée des nombres réels (float, double)

La précision limitée des nombres réels implique également que les **comparaisons d'égalité entre ceux-ci sont à éviter**.
En effet, deux nombres qui seraient égaux avec une précision totale peuvent ne pas l'être à cause des erreurs d'approximation dûes à la précision limitée.
Pour comparer deux nombres réels, il est donc préférable de vérifier que leur différence est inférieure à une limite que l'on se fixe

```
1 double a = 1./3.;
2 double b = 0.3333;
3
4 if(a == b) // dangereux !!!
5
6 if(fabs(a-b) < 1e-12) // OK
```

Exercices (1/4)

1. Détection de lettres et voyelles :
   - Saisir un caractère alphanumérique (type **char**) au clavier ;
   - S'il s'agit d'un "a", afficher "première lettre de l'alphabet" ;
   - S'il s'agit d'un "z", afficher "dernière lettre de l'alphabet" ;
   - S'il s'agit d'une voyelle, afficher "ceci est une voyelle" ;
   - Sinon, afficher "ceci est une consonne".
2. Code ASCII
   - Afficher les lettres de 'a' à 'z' en minuscules et en majuscules suivies de leur code ASCII entre parenthèses ;
   - Afficher les caractères correspondants aux codes ASCII compris entre 32 et 90.
3. Nombres aléatoires
   - Demander à l'utilisateur le nombre n de nombres aléatoires à générer ;
   - Générer n nombres entiers compris entre 0 et 1000, les afficher et les additionner successivement dans une variable `somme` ;
   - Afficher la valeur moyenne des nombres générés, qui vaut $\frac{somme}{n}$.

     Vers quelle valeur cette moyenne tend-elle lorsque n devient grand ?

Erreurs numériques avec des nombres réels :

4. Calcul de $[(10/3) - 3] \times 3 - 1 = 0$ en plusieurs étapes :
   - Calculer $a = 10.0/3.0$;
   - Calculer $b = a - 3.$;
   - Calculer $c = b * 3.$;
   - Calculer $d = c - 1.$;

   Quel est le résultat ? Est-il conforme à ce qui était attendu ?
5. Boucles de comptage. Répéter les deux exercices ci-dessous en utilisant successivement les 3 types de boucles.
   - Ecrire un programme qui compte de 0 à 100 par incréments de 1. Vérifier que le résultat final est bien égal à 100 ;
   - Ecrire un programme qui compte de 0 à 1 par incréments de 0.1. Vérifier que le résultat final est bien égal à 1 ;
6. Précision numérique :
   - Déclarer trois nombres $a = 10^{30}$, $b = -10^{30}$ et $c = 1$;
   - Calculer $(a + b) + c$ et $a + (b + c)$ et comparer les résultats.

Exercices (3/4)

7. Suite de Fibonacci et nombre d'or :

   La suite de Fibonacci est une suite de nombres entiers dont chaque terme est la somme des deux précédents : $1, 1, 2, 3, 5, 8, 13, 21, \ldots$ Le rapport de deux termes consécutifs de cette suite tend vers le nombre d'or Φ.

   - Saisir au clavier la précision e avec laquelle on veut calculer Φ ;
   - Calculer les termes de la suite de Fibonacci jusqu'à ce que la différence entre deux valeurs calculées consécutives de Φ soit inférieure à la précision e. Tester avec $e = 0.0001$;
   - Afficher le nombre d'or Φ calculé et le nombre d'itérations nécessaires pour l'obtenir.

   Pour ce faire :

   - Soient a et b les deux derniers termes actuels de la suite de Fibonacci. La valeur actuelle de ϕ est donc b/a ;
   - Calculer le terme suivant de la suite : $c = a + b$;
   - Calculer la nouvelle valeur de ϕ_{new} : celle-ci vaut c/b ;
   - Continuer à procéder de la sorte tant que $|\phi_{new} - \phi| > e$, avec e la précision voulue.

   Etape 1 : `a=1, b=1` $\rightarrow \phi$ `= b/a = 1`
   Etape 2 : `a=1, b=1` $\rightarrow$ `c=2` $\rightarrow \phi_{new}$ `= c/b = 2 et` $|\phi_{new} - \phi| = 1 > e$
   Etape 3 : `a=1, b=2` $\rightarrow$ `c=3` $\rightarrow \phi_{new}$ `= c/b = 1.5 et` $|\phi_{new} - \phi| = 0.5 > e$
   Etape 4 : ...

Exercices (4/4)

8. Le juste prix :
   - Faire générer par l'ordinateur un prix qui vaut 1000 € au maximum, avec deux chiffres derrière la virgule ;
   - Demander au second joueur d'essayer de trouver cette valeur ;
   - Si la valeur donnée est la bonne, féliciter le joueur !
   - Si la valeur est erronée, dire au joueur si la valeur cherchée est plus petite ou plus grande que la proposition et recommencer ;
   - Si le joueur n'a pas trouvé la bonne solution après 10 essais, lui annoncer qu'il a perdu et terminer le programme.

   Remarque : Pour éviter les erreurs de comparaison entre nombres réels, stocker le prix en *centimes* d'euro.

9. Mastermind avec des nombres :
   - Faire générer de manière aléatoire par l'ordinateur une suite de 4 chiffres ;
   - Demander au second joueur une suite de 4 chiffres ;
   - Si la proposition est correcte (chiffres et positions identiques), féliciter le joueur !
   - Si la proposition est incorrecte
     - Mentionner le nombre de chiffres corrects mais mal placés ;
     - Mentionner le nombre de chiffres correctement placés ;
     - Demander au joueur de ré-essayer.
   - Après 10 essais infructueux, annoncer au joueur qu'il a perdu et terminer le programme.