

Introduction à la programmation  
Travaux pratiques: séance 4  
INFO0201-1

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



- Récapitulation des séances précédentes
  - Variables
  - Structures de contrôle et structures de contrôle itératives
  - Tableaux statiques

- Récapitulation des séances précédentes
  - Variables
  - Structures de contrôle et structures de contrôle itératives
  - Tableaux statiques
  
- Fonctions et procédures
  - séries d'instructions paramétrables et ré-utilisables
    - Structuration du programme
    - Ré-utilisation de code
    - Modularité du code

- Récapitulation des séances précédentes
  - Variables
  - Structures de contrôle et structures de contrôle itératives
  - Tableaux statiques
- Fonctions et procédures
  - séries d'instructions paramétrables et ré-utilisables
    - Structuration du programme
    - Ré-utilisation de code
    - Modularité du code
- Lecture/Ecriture dans un fichier
  - Chargement de plus de données qu'au clavier
  - (ré-)Utilisation de données plus complexes
  - Sauvegarde de données

Récapitulation des notions de base :

- Variables
  - déclaration, initialisation et portée

## Récapitulation des notions de base :

- Variables  
→ déclaration, initialisation et portée
- Structures de contrôle  
→ **if...else**, **switch case()**,...

## Récapitulation des notions de base :

- Variables  
→ déclaration, initialisation et portée
- Structures de contrôle  
→ **if...else**, **switch case()**,...
- Structures de contrôle itératives  
→ **while**, **do...while**, **for**

## Récapitulation des notions de base :

- Variables  
→ déclaration, initialisation et portée
- Structures de contrôle  
→ **if...else**, **switch case()**,...
- Structures de contrôle itératives  
→ **while**, **do...while**, **for**
- Tableaux statiques  
→ déclaration, initialisation, indices,...



# Variables : déclaration, initialisation et portée

- Déclaration : toute variable utilisée dans le programme doit **d'abord** être déclarée !  
Une bonne pratique qui permet de **structurer** un programme consiste à déclarer toutes les variables principales en début de programme, dans une "zone de déclaration". Les variables qui ne sont nécessaires que de manière temporaire continueront évidemment à être déclarées là où elles sont nécessaires.
- Initialisation : toute variable non initialisée possède une valeur indéterminée (pas nécessairement zéro)
- Portée des variables : une variable déclarée entre un groupe d'accolades `{...}` n'existe qu'entre ces accolades. Elle ne peut être utilisée ni avant ni après ces accolades !  
Par exemple, dans la boucle for

```
1 for(int i=0; i < 10; i++)
```

la variable *i* n'existe qu'à l'intérieur des accolades `{...}` de la boucle, et ne peut pas être utilisée après

- structure **if...else** :

```
1  if(condition1)
2  {
3      // instructions exécutées si condition1 est VRAI
4  }
5  else if(condition2)
6  {
7      // instructions exécutées si
8      // condition2 est VRAI ET condition1 est FAUX
9  }
10 else
11 {
12     // instructions exécutées dans les autres cas
13 }
```

- structure **switch case** :

```
1  switch(une_variable)
2  {
3      case 1:
4          // instructions exécutées si une_variable == 1
5          break;
6      case 3:
7      case 4:
8          // instructions exécutées si une_variable == 3
9          // OU une_variable == 4
10         break;
11     default:
12         //instructions exécutées dans les autres cas
13 }
```

- Bloc d'instructions : lorsque plusieurs instructions doivent être exécutées dans une structure **if**, il faut entourer ces instructions avec des accolades `{...}`
- Expression de la condition dans les structures **if...else** : **ATTENTION** à la différence entre opérateur d'affectation et opérateur de **comparaison** !

```
1 if(a == 2) // On compare les valeurs 'a' et '2'
```

```
1 if(a = 2) // On change la valeur de a !!!!!
```

- structure de boucle **while** :

```
1 while(condition_pour_continuer)
2 {
3     // instructions exécutées tant que
4     // 'condition_pour_continuer' est VRAI
5 }
```

- structure de boucle **do...while** :

```
1 do
2 {
3     // instructions exécutées tant que
4     // 'condition_pour_continuer' est VRAI
5 }while(condition_pour_continuer);
```

**ATTENTION** : dans la boucle **do...while**, la condition est vérifiée **après** chaque exécution de la série d'instructions.

- structure de boucle **for** :

```
1 for(int i=debut; i < fin; i++)
2 {
3     // instructions exécutées (fin-debut-1) fois,
4     // avec des valeurs de i allant de 'debut' à
5     // 'fin'-1
6 }
```

**ATTENTION** : pas de point-virgule après

```
1 for(int i=debut; i < fin; i++)
```

ni après

```
1 while(condition_pour_continuer)
```

## Quand utiliser chaque type de boucle ?

- Boucle **for** : Quand on connaît le nombre d'itérations (parcourir un tableau, répéter des instructions un nombre précis de fois, ...)
- Boucle **while** : Quand les instructions doivent se répéter tant qu'une condition logique n'est pas fausse (convergence d'une solution, etc...)
- Boucle **do...while** : Quand on devrait utiliser une boucle **while** mais que la condition de la boucle ne peut être évaluée avant d'avoir exécuté les instructions une première fois.

- Déclaration = réserver la place en mémoire

```
1 // tableau de 'taille' entiers
2 int nom_tableau[taille];
3 // tableau de taille_1 * taille_2 éléments double
4 double nom_tableau_2D[taille_1][taille_2];
```

- Initialisation = donner des valeurs de départ

```
1 int maurice[5] = {19, 42, 785, 361, 7};
```

**ATTENTION** : l'*initialisation* peut uniquement se faire avec des valeurs 'constantes', connues avant l'exécution du programme. Donc **pas des variables** !

Pour utiliser des variables, il faut le faire élément par élément :

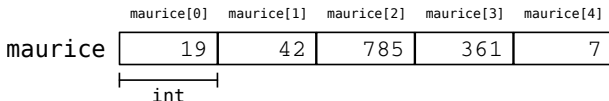
```
1 maurice[0] = a;
2 maurice[1] = b;
3 ...
```



- Numérotation des éléments : les **indices** pour accéder aux éléments du tableau vont de 0 à  $N - 1$ , où  $N$  est le nombre d'éléments du tableau.

**ATTENTION** : ne pas accéder à un élément hors du tableau !

```
1 int maurice[5] = {19, 42, 785, 361, 7};
```



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 !**

- **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 !

- **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.

Le **nombre de lignes** de code d'un programme moyennement complexe peut rapidement devenir très élevé. Pour structurer ce code on le découpe en sous-programmes appelés **fonctions**.

Ces **fonctions** peuvent être utilisées indépendamment du programme principal. On peut ainsi les utiliser à plusieurs endroits différents du même programme sans en ré-écrire le code (et éviter le copier/coller qui est source d'erreurs !). On peut même les ré-utiliser dans d'autres programmes (**modularité**).

Une fonction effectue une série d'instructions qui dépendent de la valeur de paramètres qu'on peut lui fournir. Elle retourne ensuite une valeur. Si elle ne retourne pas de valeur, on peut également lui donner le nom de **procédure** ou **routine**.

# Fonctions et procédures : déclaration

Comment déclarer une procédure/fonction :

```
type nom_fonction(type paramètre_1, type paramètre_2, ...);
```

→ Il s'agit du **prototype** de la fonction

Comment déclarer une procédure/fonction :

```
type nom_fonction(type paramètre_1, type paramètre_2, ...);
```

→ Il s'agit du **prototype** de la fonction

- **type** de la fonction : **int**, **float**, **double**, **char**, ...

Définit le type de valeur que retournera la fonction. Dans le cas particulier d'une fonction (procédure) qui ne retourne pas de valeur, on utilise le mot-clé **void** pour le signaler



Comment déclarer une procédure/fonction :

```
type nom_fonction(type paramètre_1, type paramètre_2, ...);
```

→ Il s'agit du **prototype** de la fonction

- **type** de la fonction : **int**, **float**, **double**, **char**, ...  
Définit le type de valeur que retournera la fonction. Dans le cas particulier d'une fonction (procédure) qui ne retourne pas de valeur, on utilise le mot-clé **void** pour le signaler
- **nom\_fonction** : le plus représentatif possible, il servira à appeler la fonction dans le programme

Comment déclarer une procédure/fonction :

```
type nom_fonction(type paramètre_1, type paramètre_2, ...);
```

→ Il s'agit du **prototype** de la fonction

- **type** de la fonction : **int**, **float**, **double**, **char**, ...  
Définit le type de valeur que retournera la fonction. Dans le cas particulier d'une fonction (procédure) qui ne retourne pas de valeur, on utilise le mot-clé **void** pour le signaler
- **nom\_fonction** : le plus représentatif possible, il servira à appeler la fonction dans le programme
- **type paramètre\_1** :
  - **type** : type de la variable qui est passée en paramètre
  - *paramètre\_1* : nom de celle-ci, qui sera utilisé pour y accéder dans le corps de la fonction

Comment déclarer une procédure/fonction :

```
type nom_fonction(type paramètre_1, type paramètre_2, ...);
```

→ Il s'agit du **prototype** de la fonction

- **type** de la fonction : **int**, **float**, **double**, **char**, ...  
Définit le type de valeur que retournera la fonction. Dans le cas particulier d'une fonction (procédure) qui ne retourne pas de valeur, on utilise le mot-clé **void** pour le signaler
- **nom\_fonction** : le plus représentatif possible, il servira à appeler la fonction dans le programme
- **type paramètre\_1** :
  - **type** : type de la variable qui est passée en paramètre
  - *paramètre\_1* : nom de celle-ci, qui sera utilisé pour y accéder dans le corps de la fonction
- **type paramètre\_2** : idem pour chaque paramètre, séparés par des virgules “,”

La **déclaration** d'une fonction signale au compilateur l'existence d'une telle fonction quelque part dans le code du programme.

→ Il est ensuite nécessaire de la définir. La **définition** de la fonction commence par le **prototype** de la fonction suivi des instructions que celle-ci doit accomplir, placées entre accolades {...}

```
1 // Définition de la fonction somme
2 double addition(double a, double b)
3 {
4     double somme;
5     somme = a + b;
6     return somme;
7 }
```

Le mot-clé **return** détermine la valeur retournée par la fonction. L'exécution de la fonction se termine dès que ce mot-clé est rencontré.

Toutes les fonctions utilisées dans le **main()** doivent être déclarées avant celui-ci.

Remarque : en effet, à un endroit donné du programme, n'existe que ce qui a été déclaré plus tôt dans le programme. Cela est valable pour les fonctions comme pour les variables.

Les fonctions peuvent donc être **définies** ailleurs (après le main) pour autant qu'elles aient été **déclarées** avant.

## Portée des variables :

Les variables déclarées dans une fonction n'existent que durant l'exécution de la fonction (elles sont dites **locales**).

Les valeurs des variables passées en paramètres sont copiées dans des variables locales de la fonction (qui portent le nom qui leur a été attribué lors de la définition de la fonction).

De ce fait, les variables qui ont servi à appeler la fonction ne sont pas modifiées par la fonction.

## Fonctions et procédures : exemple en pratique

```
1 // Définition de la fonction puissance entière > 0
2 double power(double base, int exposant)
3 {
4     double puissance = 1; // si exposant == 0, return 1
5     for( ; exposant > 0; exposant--) // pas d'init
6         puissance = puissance * base;
7     return puissance;
8 }
```

```
1 // Définition de la fonction affichagePuissance
2 void affichagePuissance(double base, double exposant,
3     double resultat)
4 {
5     cout << base << " à la puissance " << exposant << "
6     vaut " << resultat << endl;
7 }
8 }
```

## Fonctions et procédures : exemple en pratique

```
1 int main()
2 {
3     double a = 2;
4     double b = 4;
5     double resultat = 0;
6
7     // Enregistrement de la valeur retournée par la
8     // fonction puissance sur a, b dans resultat
9     resultat = power(a,b);
10    // --> a et b ne sont pas modifiés par la fonction
11
12    // Pas de variable de retour car
13    // affichagePuissance est une procédure
14    affichagePuissance(a,b,resultat);
15    return 0;
16 }
```

# Fonctions et procédures

Pour plus de clarté, il est recommandé d'utiliser un marquage séparateur entre les fonctions et de commenter chaque en-tête pour décrire l'utilité et/ou le fonctionnement de la fonction.

```
1  /*****  
2  /* La fonction addition calcule la somme de a et b */  
3  *****/  
4  double addition(double a, double b)  
5  {  
6      double somme;  
7      somme = a + b ;  
8      return (somme);  
9  }  
10 /*****  
11 /* affichageSomme affiche le resultat de la somme */  
12 /* de a et b                                     */  
13 *****/  
14 void affichageSomme(double a, double b, double somme)  
15 {  
16     cout << a << " + " << b << " = " << somme << endl;  
17 }
```



Il est possible de fournir un tableau en paramètre à une fonction. Il faut pour cela faire suivre le nom de la variable par des crochets []. La fonction ne connaît pas la taille du tableau : il faut un paramètre pour le préciser.

```
1 // Utiliser un tableau en argument
2 void AffichageTableau(double tab[], int Ntab)
3 {
4     for(int i = 0 ; i < Ntab ; i++)
5     {
6         cout << "tab[" << i << "] = " << tab[i] << endl;
7     }
8 }
```

**ATTENTION** : Dans ce cas, si les valeurs du tableau sont modifiées dans la fonction, la modification s'appliquera **aussi** aux valeurs du tableau qui a été passé en paramètre

NB : il s'agit en fait du même tableau, il n'a pas été copié.

Pour un tableau de dimension  $>1$ , il faut préciser la taille de chacune des dimensions.

```
1 void AffichageTableau2D(double tab[10][5])
2 {
3     for(int i = 0 ; i < 10 ; i++)
4         for(int j = 0 ; j < 5 ; j++)
5             {
6                 cout << "tab[" << i << "][" << j << "] = "
7                 << tab[i][j] << endl;
8             }
```

Une fonction est dite **récursive** lorsqu'elle fait appel à elle-même dans sa définition.

Cette possibilité est particulièrement utile dans le cas de fonctions mathématiques définies par **récurrence**.

Exemples : la fonction factorielle, la suite de Fibonacci, ...

```
1 // Définit la fonction factorielle par récurrence
2 int factorielle(int n)
3 {
4     if(n == 1)
5         return 1;
6     else
7         return n*factorielle(n-1);
8 }
```

Comme dans le cas des boucles, il est important de veiller à ce que les appels successifs ne se répètent pas de manière infinie !

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.

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.

Etapes pour l'écriture dans un fichier :

- 1 Ouvrir le fichier  
→ Permet de spécifier dans quel fichier écrire et de quelle manière.

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.

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.



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.

## Écrire 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 précédentes.

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);
```

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()
```

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();
```

## Écrire dans un fichier : exemple

```
1 #include <fstream>
2
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         // Le fichier est bien ouvert, on peut écrire
10        mon_fichier << "J'ai " << age << " ans." << endl;
11        mon_fichier.close(); // fermer le fichier
12    }
13    else
14    {
15        cout << "ERREUR: Impossible d'ouvrir le fichier";
16    }
17    return 0;
18 }
```

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

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");
```

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()
```



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()
```

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

## 1 Problème de cinématique simple - rebonds d'une balle :

Une balle lâchée d'une hauteur initiale  $h_0$  tombe et rebondit plusieurs fois. La vitesse atteinte lors de l'impact au sol est  $v_0 = \sqrt{2gh_0}$ , et on considère qu'elle rebondit avec une vitesse  $v_1 = \epsilon v_0$ , où  $\epsilon$  est le coefficient de restitution du sol. La nouvelle hauteur atteinte est alors  $h = v_1^2 / (2g)$ .

En fonction des paramètres que sont la hauteur initiale  $h_0$  et le coefficient  $\epsilon$ , écrire les fonctions :

- `double h_rebond(double h_start, double epsilon)` : calcule la hauteur atteinte après un rebond au départ d'une hauteur  $h_{start}$  ;
- `double h_final(double h_start, int n, double epsilon)` : calcule la hauteur atteinte après  $n$  rebonds ;
- `double n_rebonds(double h_start, double h_end, double epsilon)` : calcule le nombre de rebonds nécessaire avant que la hauteur atteinte soit inférieure à  $h_{end}$  mètres ;
- Pour une hauteur initiale de 10 m et un coefficient  $\epsilon = 0.85$ ,
  - calculer et afficher la hauteur atteinte après 4 rebonds ;
  - calculer et afficher le nombre de rebonds avant que la hauteur de rebond soit inférieure à 1 cm ;
  - écrire dans un fichier les hauteurs successives atteintes lors des 10 premiers rebonds.

- ② Modification du programme de calcul de  $\pi$  par la formule de Leibniz
  - Modifier le programme du TP précédent pour calculer le  $n^{\text{e}}$  terme de la série au moyen d'une fonction : `double terme_Leibniz(int n)`
- ③ Calculs avec des vecteurs dans une base orthonormée. Ecrire les fonctions
  - `double produit_scalaire(double x1, double y1, double z1, double x2, double y2, double z2)`, prenant en paramètres 6 nombres représentant respectivement les 3 composantes de deux vecteurs de l'espace et qui calcule leur produit scalaire ;
  - `void produit_vectoriel(double vecteur1[3], double vecteur2[3])`, prenant en paramètres 3 vecteurs de l'espace, qui calcule le produit vectoriel des deux premiers et place le résultat dans le troisième vecteur (remarque : utiliser des tableaux) ;
  - Appliquer ces fonctions aux vecteurs  $(10, 5, 5)$ ,  $(-2, 2, 2)$  et  $(3, 6, 1)$  deux à deux et afficher les résultats.
- ④ Evaluation de polynômes  $P(x)$  : évaluer  $P(x)$  à intervalles réguliers et écrire les données dans un fichier pour pouvoir en tracer un graphique.
  - Ecrire une fonction qui prend en paramètre un tableau avec les coefficients d'un polynôme  $P$ , une valeur  $x$  et qui retourne la valeur de  $P(x)$  ;
  - Evaluer le polynôme  $2x^4 - 3x^2 + x - 2$  pour des valeurs de  $x$  comprises dans l'intervalle  $[-3, 3]$  avec un pas de  $10^{-3}$  et écrire les couples  $(x, P(x))$  dans un fichier.

## 5 Calcul d'intégrale numérique :

Calculer numériquement l'intégrale au sens de Riemann : on utilisera pour cela une approximation de l'aire sous la courbe de la fonction par des rectangles de largeur uniforme  $\epsilon$ .

- Ecrire une fonction calculant l'intégrale de  $\cos(2x)$  dans l'intervalle  $[a, b]$  pour une valeur de  $\epsilon$  paramétrable ;
- Calculer l'intégrale de  $\cos(2x)$  dans l'intervalle  $[0, \pi/4]$  pour des valeurs de  $\epsilon$  égales à 0.1, 0.01, 0.001, 0.0001 et 0.00001 ;
- Ecrire une fonction évaluant la valeur  $P(x)$  d'un polynôme  $P$  de degré  $n$  de manière **efficace**. Pour ce faire, la fonction recevra en paramètres un tableau avec les coefficients du polynôme, le degré  $n$  du polynôme et le nombre  $x$  pour lequel évaluer le polynôme (voir exercice précédent) ;
- Ecrire une fonction calculant l'intégrale du polynôme  $P$  de degré  $n$  dans l'intervalle  $[a, b]$  pour une valeur de  $\epsilon$  paramétrable. Pour y parvenir, la fonction recevra en paramètres un tableau contenant les coefficients du polynôme, le degré du polynôme, l'intervalle dans lequel calculer l'intégrale et la largeur  $\epsilon$  des rectangles à utiliser ;
- Calculer l'intégrale du polynôme  $7x^5 + 3x^4 + 2x^2 + 5x + 7$  dans l'intervalle  $[0, 10]$  pour  $\epsilon = 0.001$ . Comparer en évaluant le polynôme qui correspond à la primitive de  $7x^5 + 3x^4 + 2x^2 + 5x + 7$  ;
- Utiliser les fonctions précédentes pour calculer  $I(t) = \int_0^t P(x)dx$  pour  $t \in [0, 10]$  avec un pas de 0.01 et écrire le résultat dans un fichier.

- 6 Chargement de données et distribution : le fichier data.dat disponible à l'adresse <http://spin02.phys.ulg.ac.be/> contient des nombres.
- Ecrire une fonction qui répartit des données en classes. La fonction prend en paramètres le nombre de classes, un tableau avec la valeur centrale de chaque classe, la largeur des classes et un tableau dans lequel sera totalisé le nombre d'éléments de chaque classe ;
  - Appliquer cette fonction au fichier data.dat pour un ensemble de 15 classes de largeurs identiques couvrant l'intervalle  $[-4.5 : 4.5]$  ;
  - Ecrire une fonction qui prend en paramètres un tableau contenant une distribution réparties en classes et la taille de celui-ci. La fonction affiche la distribution dans la console horizontalement (voir image) ;  
Remarque : la largeur des lignes dans la console est limitée à 80 caractères. Il est dès lors conseillé de ne pas dépasser 70 caractères de large pour l'affichage de la distribution.
  - Ecrire la distribution dans un fichier distribution.txt ;
  - Tester également le programme avec le fichier data2.dat



Fenêtre d'affichage d'une distribution normale de moyenne nulle dans la console :

