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

R. Chrétien, G. Vanhaele & B. Baert, X. Baumans
rchretien@ulg.ac.be - guillaume.vanhaele@ulg.ac.be

Programme de la séance

- Récapitulation des séances précédentes
  - Variables
  - Structures de contrôle et structures de contrôle itératives
  - Tableaux statiques
- Les tableaux dynamiques
  $\rightarrow$ Gestion de tableaux de taille variable avec l'objet *vector*
- Fonctions et procédures
  $\rightarrow$ séries d'instructions paramétrables et ré-utilisables
  - Structuration du programme
  - Ré-utilisation de code
  - Modularité du code

Récapitulation des trois premières séances

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

Structures de contrôle : if...else, switch case

- 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 }
```

Structures de contrôle : if...else, switch case

- 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 }
```

Structures de contrôle : if...else, switch case

- 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 !!!!!
```

Structures de contrôle itératives : while, do...while, for

- structure de boucle **while** :

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

- 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.

Structures de contrôle itératives : while, do...while, for

- structure de boucle **for** :

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

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

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

ni après

```
1 while(condition_pour_continuer)
```

(sauf dans un **do ... while()** ;)

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.

Les tableaux statiques

- Déclaration = réserver la place en mémoire

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

- Initialisation = donner des valeurs de départ

```
1 int tab[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** !

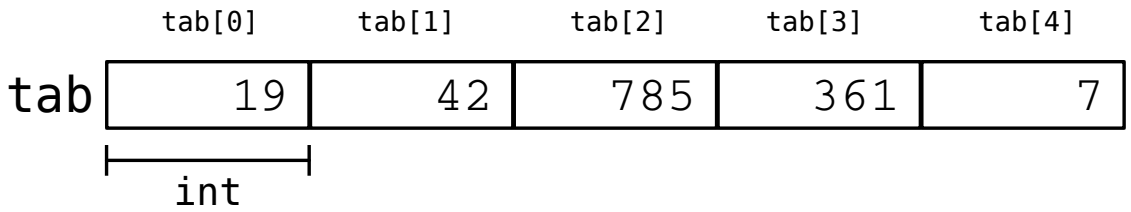
Pour utiliser des variables, il faut le faire élément par élément :

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

Les tableaux statiques

- 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 tab[5] = {19, 42, 785, 361, 7};
```



Règles de bonne pratique

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 !
- 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.

L'objet *vector*

- Pour gérer de manière pratique des tableaux dynamiques, c'est-à-dire des tableaux dont la taille peut changer pendant l'exécution du programme, il existe en C++ un objet spécifique : l'objet **vector**
- Les avantages principaux de cet objet sont les suivants :
  - Gestion de l'allocation et de la libération de mémoire sous-jacente
  - Tableaux redimensionnables lors de l'exécution
  - Copie de tableaux complets facilitée
- Pour pouvoir l'utiliser, il faut inclure la bibliothèque "vector" :

```
#include <vector>
```

L'objet *vector* : utilisation

- Les objets `vector` font appel à une notion, les *templates*, pour spécifier le type des variables stockées dans un tableau dynamique. La notation de la déclaration d'un objet `vector` est la suivante : `vector<type> mon_vector;`

```
1 vector<int> mon_tableau_dynamique;
```

- On peut également déclarer un objet `vector` d'une certaine taille (qui peut désormais être une variable qui ne prend une valeur connue qu'à l'exécution), et préciser avec quelle valeur remplir tous les éléments du tableau par défaut : `vector<type> mon_vector(int size, type value);`

```
1 int n, m;
2 cin >> n >> m;
3 vector<int> mon_tableau_dynamique(n, m);
4 // tableau dynamique de n éléments égaux à m
```

L'objet *vector* : utilisation

- On peut accéder aux éléments d'un objet `vector` comme à ceux d'un tableau normal. On utilise pour cela l'opérateur [ ] :

```
1 vector<double> mon_tableau(100, 0);
2 for(int i=0; i < 100; i++)
3 {
4   mon_tableau[i] = cos(i/100.);
5 }
```

ATTENTION : La valeur de l'index utilisé avec l'opérateur [ ] doit être inférieure à la taille de l'objet `vector`, comme pour les tableaux statiques. Dans le cas contraire, le résultat n'est pas défini, mais mènera certainement à une erreur !

Propriétés et fonctions membres de l'objet *vector*

L'objet *vector* contient également toute une série de fonctions permettant d'obtenir des informations ou d'effectuer des manipulations sur le tableau de données qu'il représente.

Toute la documentation concernant les fonctions liées à l'objet *vector* est disponible à l'adresse
`http://www.cplusplus.com/reference/vector/vector/`

Remarque : Dans certaines de ces fonctions, il est fait appel au concept d'*itérateur*. Cette notion ne fait pas partie du cadre de ce cours, mais est expliquée en détails sur le site de référence, avec de nombreux exemples explicites.

Propriétés et fonctions membres de l'objet *vector*

Parmi les fonctions principales, on trouve ainsi , avec `v` un objet de type vecteur :

- **int** `v.size()` : Renvoie la taille du tableau de données contenu dans le vector `v`
- **void** `v.resize(`**int** `n, `**type** `val)` : Redimensionne le vecteur à la taille *n*
  - Si *n* est plus grand que la taille précédente : le `vector` est agrandit à la taille *n* et les nouveaux éléments prennent la valeur *val* ;
  - Si *n* est plus petit que la taille précédente : le `vector` est réduit à la taille *n* et les éléments excédentaires sont supprimés (le paramètre *val* n'est dans ce cas pas nécessaire)
- **bool** `v.empty()` : Teste si le vector `v` est vide (taille = 0)

Propriétés et fonctions membres de l'objet *vector*

- **void** `v.push_back(`**type** `val)` : Ajoute un élément à la fin du `vector` avec la valeur *val*
- **void** `v.pop_back()` : Supprime le dernier élément du `vector`
- **type&** `v.front()` : Renvoie une *référence* vers le premier élément du `vector`, *i.e.* `v[0]`
- **type&** `v.back()` : Renvoie une *référence* vers le dernier élément du `vector`, *i.e.* `v[v.size() - 1]`
- **void** `v.clear()` : Supprime tous les éléments du `vector`, ramenant ainsi sa taille à 0
- **type*** `v.data()` : Renvoie un pointeur direct vers la zone mémoire contenant le tableau de données du `vector`

Opérateurs utilisables avec l'objet *vector*

Certains opérateurs peuvent être utilisés avec les objets de type `vector` :

- L'opérateur d'affectation "=" :
  Cet opérateur permet, contrairement à ce qui se produirait avec deux tableaux classiques, de **copier** les valeurs d'un `vector` dans un autre.

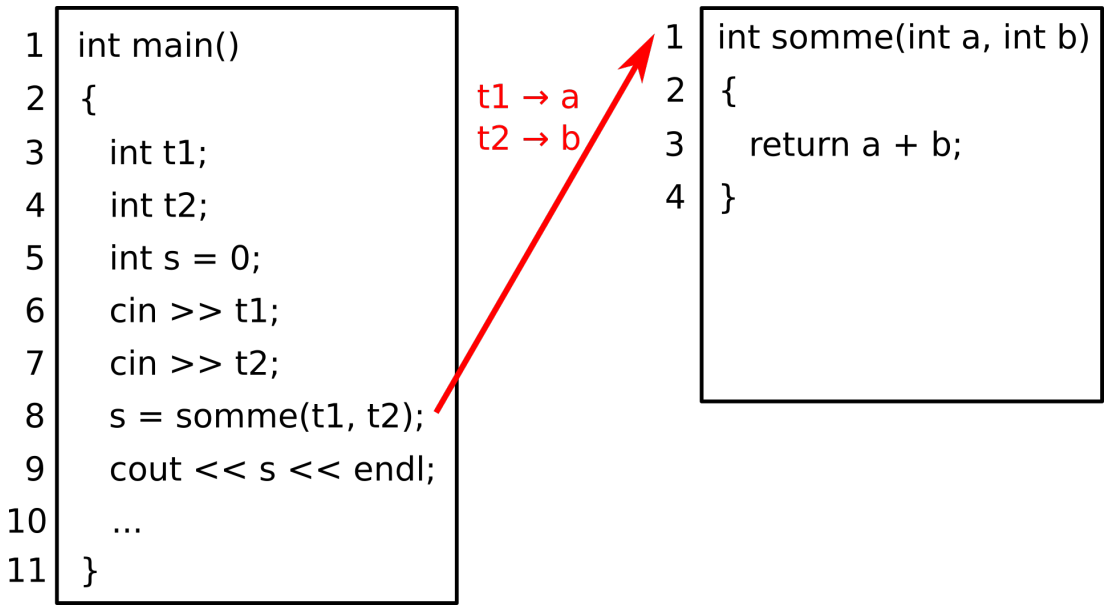
- L'opérateur de comparaison "==" :
  Cet opérateur vérifie d'abord que les deux objets `vector` ont la même taille (`.size()`) et si c'est le cas, les éléments respectifs des deux `vector` sont comparés deux à deux.
  La comparaison est donc **vraie** si le contenu des deux `vector` est strictement égal, et **faux** dans le cas contraire.

Fonctions et procédures

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

Ces **fonctions** peuvent être utilisées indépendamment du programme principal. On peut les utiliser à plusieurs endroits différents du même programme sans les ré-écrire (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

- **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 ","

Fonctions et procédures : déclaration et définition

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.

Fonctions et procédures

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,
      double resultat)
3 {
4   cout << base << " à la puissance " << exposant << "
      vaut " << resultat << endl;
5 }
```

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
       fonction puissance sur a, b dans resultat
8      resultat = power(a,b);
9      // --> a et b ne sont pas modifiés par la fonction
10
11     // Pas de variable de retour car
       affichagePuissance est une procédure
12     affichagePuissance(a,b,resultat);
13     return 0;
14 }
```

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 }
```

Fonctions et procédures : passage d'un tableau en argument

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 connait 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.

Fonctions et procédures : passage d'un tableau en argument

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 << "] = "
       << tab[i][j] << endl;
7    }
8  }
```

Fonctions et procédures : passage d'un vector en argument

Il est également possible de fournir un vector en paramètre à une fonction. Il faut pour cela faire précéder le nom de la variable par une esperluette &. Il n'est pas nécessaire de préciser en argument la taille du vector, celle-ci étant accessible via la méthode `v.size()`

```
1  // Utiliser un vector en argument
2   void AffichageVector(vector<double> &v)
3   {
4      for(int i = 0 ; i < v.size() ; i++)
5      {
6        cout << "v[" << i << "] = " << v[i] << endl;
7      }
8   }
```

ATTENTION : Dans ce cas-ci également, si les valeurs du vector sont modifiées dans la fonction, **la modification s'appliquera aussi aux valeurs du vector qui a été passé en paramètre**
NB : il s'agit en fait du même vector, il n'a pas été copié.

Fonctions et procédures : la récursivité

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 !

La programmation défensive : introduction

Principe

La programmation défensive est une façon de programmer qui permet d'assurer le bon comportement d'un logiciel dans des circonstances et utilisations imprévues.

- On souhaite lire/écrire dans un fichier non ouvert.
- On s'attend à une entrée de type **double** d'un utilisateur et celui-ci tape une chaîne de caractères.
- On attend (par exemple) un nombre entier positif donné par l'utilisateur et celui-ci tape un nombre négatif ou réel.

Ces comportements inattendus par le programmeur entraînent bien souvent une erreur du programme s'ils n'ont pas été anticipés.
$\rightarrow$ Toujours informer l'utilisateur de ce qu'on attend qu'il tape.
$\rightarrow$ Terminer l'exécution du programme dès qu'un comportement inattendu survient.

Exercices (1/5)

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 ϵ 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 ϵ, 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.

Exercices (2/5)

2. Modification du programme de calcul de π par la formule de Leibniz
   - Modifier le programme du TP précédent pour calculer le n^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], double vecteur[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.

Exercices (3/5)

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 ϵ. L'intégrale de f dans l'intervalle $[x_0; x_N]$ peut être approximée par $\sum_{n=0}^{N-1} \epsilon\ f(x_n + \epsilon/2)$.
   - Ecrire une fonction calculant l'intégrale de $\cos(2x)$ dans l'intervalle $[a, b]$ pour une valeur de ϵ paramétrable ;
   - Calculer l'intégrale de $\cos(2x)$ dans l'intervalle $[0, \pi/4]$ pour des valeurs de ϵ 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 ϵ 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 ϵ 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.

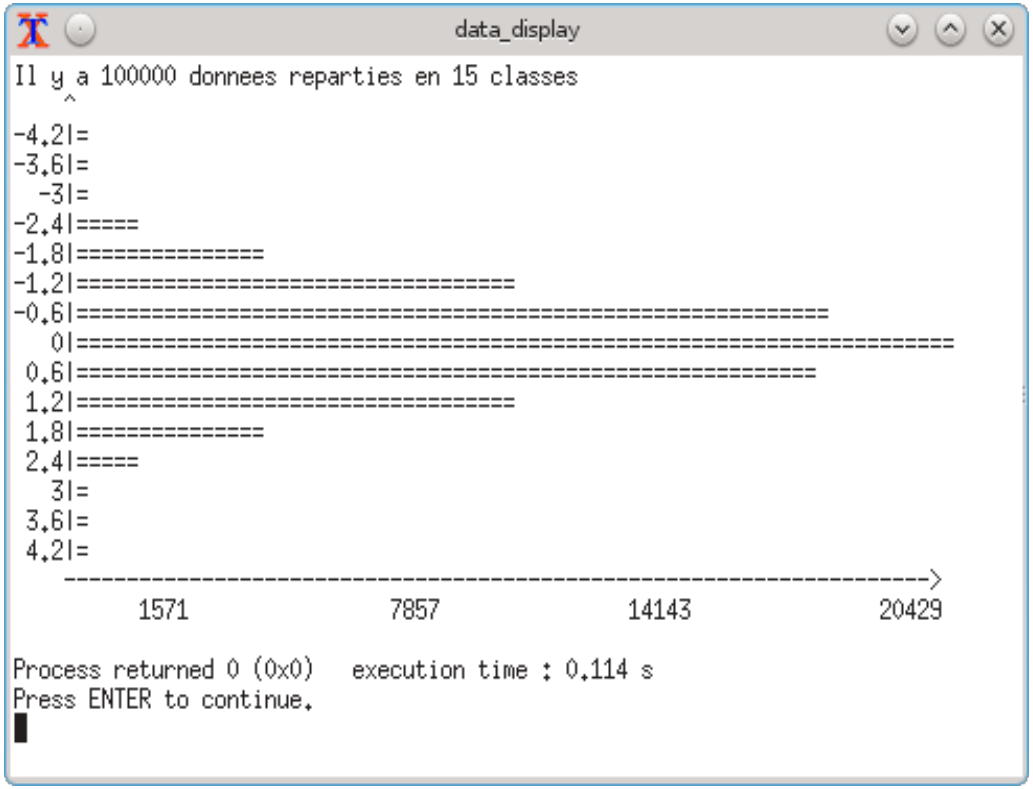
Exercices (4/5)

6. Suite de Fibonacci par récurrence
   - Ecrire un petit programme qui calcule la suite de Fibonacci avec une fonction récursive `int fibo(int n)`, où n est le nombre de termes de la suite que l'on souhaite obtenir.

7. Chargement de données et distribution : le fichier data.dat disponible sur le site `http://www.pqs.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

Exercices (5/5)

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



8. Réécrire tous vos programmes à l'aide de `vector` plutôt que des tableaux statiques.