Introduction à la programmation
Travaux pratiques: séance 8
INFO0201-1

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

Programme de la séance

- Mesurer le temps d'exécution d'un programme
  $\rightarrow$ Comparer la performance des algorithmes
- Les pointeurs : suite
  - Opérations sur les pointeurs
  - Pointeurs de pointeurs
  - Pointeurs de fonctions
- Retour sur les fonctions : les fichiers `.h` et `.cpp`
- Les tableaux dynamiques à deux dimensions
  - Tableaux 2D arrangés en ligne (tableau 1D)

Temps d'exécution d'un programme

Pour comparer les performances de différents algorithmes, il est nécessaire de pouvoir calculer le temps d'exécution d'une partie spécifique du programme.

On utilise pour cela la fonction **clock**, qui retourne le nombre de "battements d'horloge" depuis un moment proche du début de l'exécution du programme.

Pour calculer un temps d'exécution, il suffit donc de calculer la différence entre le nombre de "battements d'horloge" à la fin de l'exécution d'une partie du programme et le nombre déjà écoulés au début.

Enfin, pour avoir une durée qui puisse être exprimée en secondes, il suffit de diviser ce nombre de "battements d'horloge" par le nombre de battements par seconde qui vaut **CLOCKS_PER_SEC**

Temps d'exécution d'un programme

Pour utiliser tout cela, il faut inclure la librairie **ctime**

```
1 #include <ctime>
```

La fonction **clock()** retourne une variable de type **clock_t** qui est en fait un simple nombre entier.

```
1 double x = 0;
2 clock_t debut = clock();
3 for(int a=0; a < 1000000; a++)
4   for(int b=0; b < 1000000; b++
5     x = a*b*(b+1.)*(a+1.);
6 clock_t fin = clock();
7 cout << "temps = " << (fin-debut)*1000/CLOCKS_PER_SEC
     << " ms (" << fin-debut << " ticks)" << endl;
```

Rappels : notion de pointeur

- Pointeur = variable qui contient l'adresse d'une autre variable
- Déclaration d'un pointeur : **type** *nom_du_pointeur ;
- Relations variables-pointeurs :
  - int a ;
    a est une variable entière
    &a est une adresse mémoire (pointeur)
  - int *b ;
    b est un pointeur (adresse mémoire)
    *b est une variable entière

Equivalence tableau - pointeur

Un **tableau/pointeur** peut être vu comme un **pointeur/tableau** !

En effet, le nom d'un tableau n'est rien d'autre qu'un pointeur vers le premier élément du tableau.
De même, un pointeur vers une série de variables contigues dans la mémoire correspond à un tableau.
Ainsi, on peut écrire

```
1 int tab1[10]; // 'tab1' est un pointeur
2 int* tab2 = tab1; // 'tab2' est un "tableau"
3
4 tab2[5] = 0; // ceci fonctionne
5
6 *(tab1 + 1) = 12; // ceci aussi ==> tab[1] = 12;
```

Opérations sur les pointeurs

Même si toutes les opérations arithmétiques de base ne sont pas permises avec les pointeurs, les opérations d'incrémentation et décrémentation restent permises, avec une signification légèrement différente, et permettent de parcourir les éléments d'un tableau.

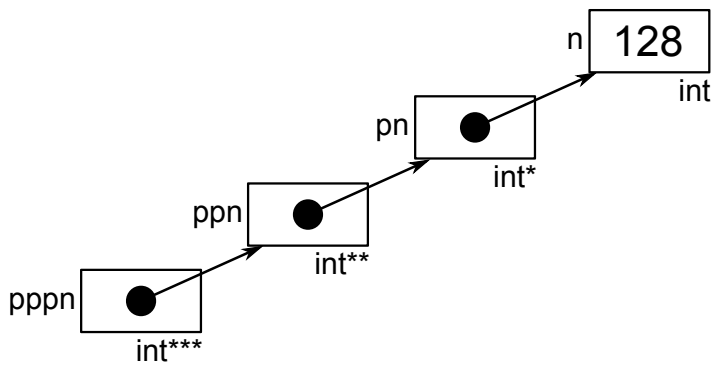
En effet, lorsque ces opérations sont appliquées à des pointeurs, l'ajout de **une unité** signifie "avancer dans la mémoire d'une quantité égale à la taille de l'objet vers lequel pointe le pointeur".

```
1  int tableau[100];
2  int* t = tableau; // 4325756
3
4  ++t; // 4325756 + 4 = 4325760
5  *t = 0; // t[1] = 0
6
7  t = t + 3; // 4325760 + 3*4 = 4325772
8  *t = 0; // t[1+3] = t[4] = 0
9
10 --t; // 4325772 - 4 = 4325768
11 *t = 0; // t[4-1] = t[3] = 0
```

Pointeurs de pointeurs

Un pointeur est un type de variable. On peut donc imaginer de créer **un pointeur qui pointe vers une variable qui est elle-même un pointeur** vers une autre variable.
On déclare ainsi un **pointeur de pointeur**. On peut même déclarer des pointeurs de pointeurs de pointeurs, et ainsi de suite.

```
1  int n = 128; // variable
2  int* pn = &n; // pointeur
3  int** ppn = &pn; // pointeur vers un pointeur
4  int*** pppn = &ppn; // pointeur de pointeur de pointeur
5  *pppn // --> = ppn
6  **ppn = 32; // n = 32;
7  ***pppn = 64; // --> n = 64;
8
```



Pointeurs de fonctions : introduction

On considère le problème suivant : calculer le minimum de $f(x)$ sur l'intervalle $[a, b]$

```
1  //double f(double x) { return x*x; }
2  double f(double x) { return 1/x; }
3  //double f(double x) { return sqrt(x); }
4  //double f(double x) { return exp(x); }
5
6  double minimum(double a, double b) {
7     double min = 1E308;
8     for(double x=a; x<b; x+= 0.01) {
9         if (f(x) < min) {
10            min = f(x);
11        }
12    }
13    return min;
14 }
```

Problème : si on veut calculer le minimum d'une autre fonction, il faut recompiler et relancer le programme.

Pointeurs de fonctions : déclaration et initialisation

Pointeurs de fonctions

Un **pointeur de fonction** est un pointeur qui contient l'adresse du début du code exécutable d'une fonction.

- Déclaration : **type** (*identificateur)(type_arg1, type_arg2, . . .) ;

```
1  int (*pf)( int, int );
```

Dans cet exemple, **pf** est un pointeur vers une fonction qui renvoie un entier et qui attend deux entiers en paramètre.

- Initialisation : affecter l'adresse d'une fonction correspondant au prototype attendu par le pointeur au moyen de l'opérateur de référencement & (**NE JAMAIS OUBLIER L'INITIALISATION**).

```
1  int somme( int a, int b ) { return a + b; }
2
3  int main() {
4     int (*pf)( int, int );   // Déclaration
5     pf = &somme;   // Initialisation. pf = somme OK aussi
6  }
```

Remarque : **pour des pointeurs de fonctions**, on peut omettre le &.

Pointeurs de fonctions : utilisation et intérêt

- Utilisation : (*identificateur)(arg1, arg2, ...)

```
1  int somme( int a, int b ) { return a + b; }
2
3  int main() {
4     int (*pf)( int, int );   // Déclaration
5     pf = &somme;   // Initialisation. pf = somme OK aussi
6     int resultat = (*pf) (3, 4);   // Utilisation. pf(3,4)
       OK aussi
7  }
```

Remarque : **pour des pointeurs de fonctions**, de même qu'à l'initialisation on peut omettre le &, on peut également omettre (*) à l'utilisation. C'est même la façon la plus courante de se servir des pointeurs de fonctions. Cela n'est par contre pas valable pour des pointeurs "classiques".

- Intérêt : très utile lorsque l'on souhaite coder une fonction (calcul d'un minimum/maximum, intégration, interpolation, ...) qui traite une fonction mathématique a priori inconnue. Cela permet d'implémenter le traitement voulu sans connaître l'expression de la fonction mathématique. $\rightarrow$ sera utilisé en bloc 2 lors du calcul de racines, de l'intégration d'une fonction ou de résolution d'EDOs (Runge-Kutta).

Pointeurs de fonctions : retour sur l'exemple initial

Calculer le minimum de $f(x)$ sur l'intervalle $[a, b]$.

```
 1  double carre(double x) { return x*x; }
 2  double inverse(double x) { return 1/x; }
 3  double racine(double x) { return sqrt(x); }
 4  double exponentielle(double x) { return exp(x); }
 5
 6  double minimum(double a, double b, double(*pf)(double)) {
 7      double min = 1E308;
 8      for(double x=a; x<b; x+= 0.01) {
 9          if ( (*pf)(x) < min ) min = (*pf)(x);
10      }
11      return min;
12  }
13
14  int main () {
15      double (*pf)( double );   // Déclaration
16      pf = &racine;   // Initialisation
17      cout << "Le minimum de sqrt(x) sur [0,2] est " << minimum(0,2,pf);
18      pf = &inverse;   // On change la fonction pointée
19      cout << "Le minimum de 1/x sur [-5,23] est " << minimum(-5,23,pf);
20  }
```

$\rightarrow$ Solution élégante, modulaire et générale.

Retour sur les fonctions : les fichiers `.h` et `.cpp`

Rappel : Pour faire appel à une fonction, il est obligatoire que celle-ci soit définie avant son appel.

```
1  int main() {
2     double a = 5., b = 10., c;
3     c = somme(a, b);
4     return 0;
5  }
6
7  double somme(double x, double y) {
8     return x + y;
9  }
```

Ce programme ne compilera pas, la fonction somme étant inconnue du compilateur lors de son appel dans le programme principal, comme ça serait également le cas pour une simple variable.

Retour sur les fonctions : les fichiers .h et .cpp

Solution :

```
1  double somme (double x, double y); // Déclaration de la fonction
2
3  int main() {
4     double a = 5., b = 10., c;
5     c = somme(a, b);
6     return 0;
7  }
8
9  double somme(double x, double y) { // Implémentation de la fonction
10    return x + y;
11 }
```

ou

```
1  double somme(double x, double y) { // Déclaration
2     return x + y; // et implémentation
3  } // de la fonction simultanément
4
5  int main() {
6     double a = 5., b = 10., c;
7     c = somme(a, b);
8     return 0;
9  }
```

Retour sur les fonctions : les fichiers `.h` et `.cpp`

Les 2 solutions précédentes, bien que commodes pour de petits programmes, peuvent devenir impossibles à maintenir dans un contexte plus réaliste où les fonctions sont nombreuses.

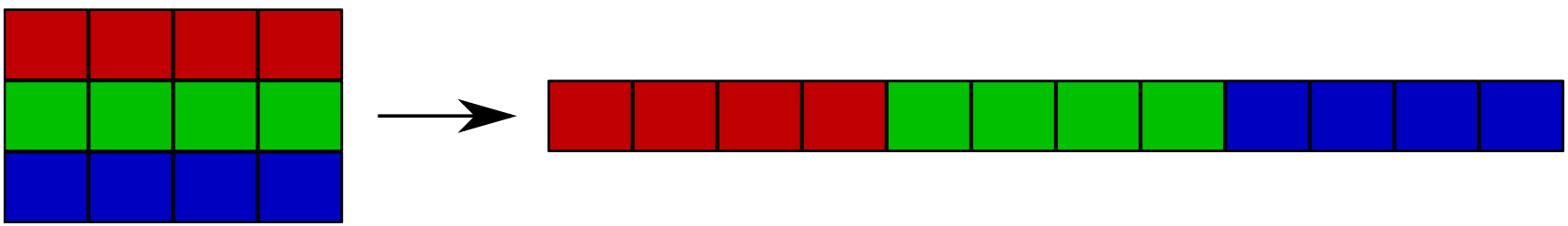
- En pratique, on définit les fonctions dans un (ou plusieurs) fichiers `.cpp` que l'on regroupe généralement selon le rôle des fonctions (ex : `integration.cpp`, `interpolation.cpp`, ...).
- À chaque fichier `.cpp`, on associe un fichier `.h` dans lequel on déclare comme précédemment chaque fonction incluse dans le fichier `.cpp` (ex : `integration.h`, `interpolation.h`, ...).
- Dans le programme principal, on spécifie au début du programme

```
1 #include "integration.h"
2 #include "interpolation.h"
```

les librairies ainsi définies entre guillemets (les chevrons <> sont réservés aux bibliothèques de la bibliothèque standard du C++).

Tableaux dynamiques 2D efficaces

Il est possible de déclarer des tableaux dynamiques à 2 dimensions. Cependant, ceux-ci sont construits sur base d'une série de tableaux 1D répartis à **différents endroits de la mémoire**, ce qui n'est pas efficace en terme d'accès à la mémoire (accéder à des endroits différents prend plus de temps qu'accéder à un seul endroit) et implique une **perte de performance**.
Pour remédier à ce problème, on préfèrera allouer un grand tableau 1D qui aura la même longueur que le nombre total de cellules du tableau 2D correspondant.



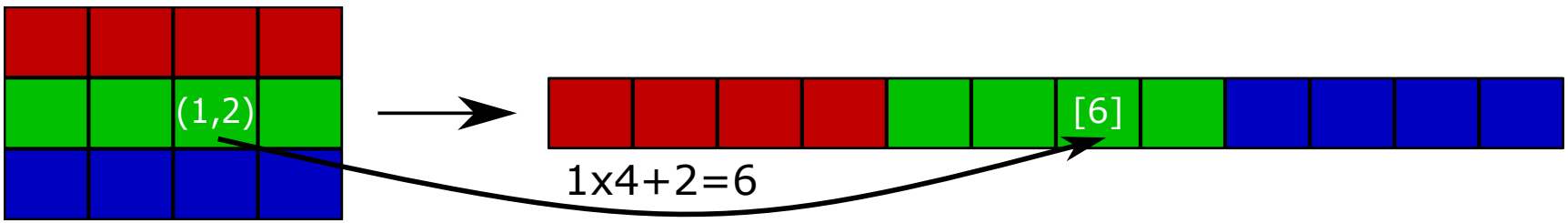
L'ordinateur gère ainsi un seul grand tableau (dynamique) continu dans la mémoire, mais l'utilisateur l'utilise comme un tableau 2D

Tableaux dynamiques 2D efficaces : accès aux éléments

Pour accéder aux éléments d'un tel tableau 2D remis en ligne, il faut convertir un couple (i, j) représentant l'élément de la ligne i et de la colonne j en un indice x du tableau 1D qui existe réellement. La relation entre les deux est la suivante :

$$\text{indice}_{\text{tableau1D}} = x = i * \text{NB\_COLONNES} + j$$

où NB_COLONNES est le nombre de colonnes du tableau 2D.



ATTENTION : L'indice du premier élément, de ligne ou de colonne, possède toujours la valeur zéro.

Exercices (1/4)

1. Transposition d'une matrice carrée :
   - Ecrire un programme qui demande à l'utilisateur de saisir une matrice carrée M de taille variable et la stocker dans un tableau (`vector`) 2D dynamique ;
   - Afficher la matrice transposée M^T dans la console.

2. Tirages aléatoires
   - Ecrire un programme qui réalise un grand nombre de tirages aléatoires de deux dés (valeurs de 1 à 6).
   - Calculer la probabilité d'obtenir chacune des valeurs possibles pour la somme des deux dés. Aide : ces valeurs sont $2, 3, 4, 5, 6, 7, 8, 9, 10, 11$ et 12
   - Quel est le résultat le plus probable ?

Exercices (2/4)

3. Produit tensoriel de deux vecteurs :
   Le produit tensoriel de deux tenseurs d'ordre 1 (vecteurs 1D) $u(m \times 1)$ et $v(1 \times n)$ correspond à la matrice $A(m \times n)$, résultat du produit matriciel de u et v^{T} :

$$A = u \otimes v = uv^{\mathsf{T}}$$

   Ecrire un programme qui
   - demande à l'utilisateur de saisir deux vecteurs de tailles variables (la taille est demandée à l'utilisateur lors de l'exécution) ;
   - calcule le produit tensoriel de ces deux vecteurs dans un tableau dynamique (2 dimensions) ;
   - affiche le résultat à la fois à l'écran et dans un fichier nommé "produit_tensoriel.txt" ;

Exercices (3/4)

4. Calcul d'une fonction par son développement en série :
   La fonction $\ln(1+x)$ a pour développement en série dans l'intervale $x \in]-1,1]$

$$\ln(1+x) = \sum_{n=1}^{+\infty} \alpha_n(x) = \sum_{n=1}^{+\infty} (-1)^{n-1} \frac{x^n}{n}$$

   - Ecrire une fonction **double** `alpha_n(`**double** `x,`**int** `n)` qui renvoit la valeur de $\alpha_n(x)$
   - Demander à une valeur de $x \in]-1,1]$ et une précision ϵ (< 1) à l'utilisateur
   - A l'aide de la fonction précédente, calculer $\ln(1+x)$ en additionnant les $\alpha_n(x)$ jusqu'à ce que $\alpha_n(x) < \epsilon$
   - Afficher le résultat et comparer avec la valeur calculée directement par `log(1+x)` de `<cmath>`
   - Pour $x = 0.5$, quelle valeur de ϵ faut-il utiliser pour obtenir une différence entre les deux résultats $< 10^{-6}$?

Exercices (4/4)

5. Crible d'Eratosthène
   Le crible d'Eratosthène est un procédé permettant de trouver les nombres premiers compris entre 2 et N. Pour ce faire, dans un tableau contenant tous les nombres de 2 à N, on supprime (= fixe à zéro) successivement tous les multiples d'un nombre entier. A la fin, il ne reste donc que les nombres qui ne sont multiples d'aucun autre entier. On commence ainsi par supprimer tous les multiples de 2. On supprime ensuite les multiples du plus petit entier restant suivant (*i.e.* 3) et ainsi de suite. On s'arrête lorsqu'un entier $>= \sqrt{N}$ est atteint.
   - Ecrire une fonction `vector<int> Eratosthene(int N)` qui crée un `vector Nombres` local contenant tous les nombres de 1 à N et, par la méthode du crible d'Eratosthène, ne laisse dans le tableau que les nombres premiers. La fonction retourne un `vector` qui ne contient que les éléments non nuls du vector `Nombres`, c-à-d les nombres premiers jusqu'à N ;
   - Le programme demande à l'utilisateur de choisir un nombre N. Au moyen de la fonction `Eratosthene`, calculer les nombres premiers compris entre 1 et N et les enregistrer dans un fichier ;
   - Calculer et afficher le temps d'exécution de la fonction de calcul des nombres premiers pour des nombres premiers compris entre 1 et 100, 1 et 10 000 et 1 et 100 000.