Introduction à l'informatique
Travaux pratiques: séance 7
INFO0205-1

X. Baumans
(xavier.baumans@ulg.ac.be)

Programme de la séance

- Rappels sur les tableaux
  $\rightarrow$ tableau 2D : matrices

Programme de la séance

- Rappels sur les tableaux
  $\rightarrow$ tableau 2D : matrices

- Rappels sur les fonctions
  $\rightarrow$ séries d'instructions paramétrables et ré-utilisables

Programme de la séance

- Rappels sur les tableaux
  $\rightarrow$ tableau 2D : matrices

- Rappels sur les fonctions
  $\rightarrow$ séries d'instructions paramétrables et ré-utilisables

- Paramètres :
  $\rightarrow$ Passage par valeur / Passage par référence

  Travailler sur les variables initiales et pas des copies !

Les tableaux statiques : accès aux élements

Attention à la numérotation des indices !

```
1 int v[3] = {4, 6, 8};
2 cout << "Element 1 = " << v[0] << endl; // affiche 4
3 cout << "Element 2 = " << v[1] << endl; // affiche 6
4 cout << "Element 3 = " << v[2] << endl; // affiche 8
5 v[1] = 12;
6 cout << "Element 2 = " << v[1] << endl; // affiche 12
```

Tout comme pour les tableaux 1D, chaque dimension d'un tableau à D dimensions est numérotée de 0 à $N_D - 1$ où N_D est la taille de la D-ième dimension du tableau.

```
1 int v[3][4] = { {1, 2, 3,  4},
2                 {5, 6, 7,  8},
3                 {8, 9, 10, 11} };
4 cout << v[1][2] << endl; // affiche 7
5 cout << v[2][0] << endl; // affiche 8
```

Remplir un tableau

La plupart du temps, on utilise des boucles pour initialiser, traiter et afficher des tableaux. Dans ce cas, une variable entière est utilisée pour accéder aux éléments du tableaux.

```
1  const int n = 10;
2
3  int main()
4  {
5     int vect[n] = {0};  int matr[n][n] = {0};
6
7     for(int i = 0; i < n; i++){
8        v[i] = rand%6 ;
9     }
10
11    for(int i = 0; i < n; i++)
12     for(int j = 0; j < n; j++)
13       cin >> matr[i][j];
14
15    return 0;
16 }
```

Remplir un tableau

A 1D :

```
1    for(int i = 0; i < n; i++){
2
3        cout << v[i] << "\t";
4    }
```

A 2D :

```
1    for(int i = 0; i < n; i++){
2     for(int j = 0; j < n; j++){
3
4        cout << matr[i][j] << "\t";
5     }
6
7     cout << endl;
8    }
```

Fonctions et procédures

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

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

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

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

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.

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

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 de a et b*/
12 /*************************************************************/
13 void affichageSomme(double a, double b, double somme){
14
15   cout << a << " + " << b << " = " << somme << endl;
16 }
```

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      for(int i = 0 ; i < Ntab ; i++){
4
5        cout << "tab[" << i << "] = " << tab[i] << endl;
6      }
7   }
```

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 : 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 !

Fonction : paramètres par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

Fonction : paramètres par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.

Fonction : paramètres par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.
- **Passage par référence** : Le passage par référence stipule que la variable ne doit pas être copiée, mais que les modifications qui lui sont appliquées dans la fonction doivent l'être à la variable initiale. Il suffit pour cela de précéder le nom de la variable du caractère "&" : int fonction(int &a)

Fonction : paramètres par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.
- **Passage par référence** : Le passage par référence stipule que la variable ne doit pas être copiée, mais que les modifications qui lui sont appliquées dans la fonction doivent l'être à la variable initiale. Il suffit pour cela de précéder le nom de la variable du caractère "&" : int fonction(int &a)
- **Passage par adresse** : c'est un pointeur vers la variable utile qui est passé en paramètre. Il permet d'accéder à la valeur de cette variable mais aussi de modifier la valeur de la variable initiale.

  Remarque un tableau est passé par adresse

Fonction : paramètres par référence

Nous avons déjà vu la notion de fonction et de passage de paramètre par valeur. Il existe au total 3 moyens de passer des arguments à une fonction :

- **Passage par valeur** : la valeur de la variable passée en paramètre est copiée dans une nouvelle variable qui n'existe que dans la fonction. Les modifications appliquées à cette copie ne changent pas la valeur de la variable initiale.
- **Passage par référence** : Le passage par référence stipule que la variable ne doit pas être copiée, mais que les modifications qui lui sont appliquées dans la fonction doivent l'être à la variable initiale. Il suffit pour cela de précéder le nom de la variable du caractère "&" : int fonction(int &a)
- **Passage par adresse** : c'est un pointeur vers la variable utile qui est passé en paramètre. Il permet d'accéder à la valeur de cette variable mais aussi de modifier la valeur de la variable initiale.
  Remarque un tableau est passé par adresse

Avec ces deux dernières méthodes, seule l'adresse est passée en argument à la fonction. Il n'existe alors qu'une seule zone mémoire partagée entre la fonction et la partie du programme qui l'appelle. Toutes les modifications réalisées dans la fonction seront appliquées aux variables originales.

Fonction : passage par référence

```
 1  void minmax(int i, int j, int& min, int& max)
 2  {
 3      if(i<j)
 4      { min=i; max=j; }
 5      else
 6      { min=j; max=i; }
 7  }
 8  int main()
 9  {
10      int a, b, w, x;
11      cout << "Tapez la valeur de a : "; cin >> a;
12      cout << "Tapez la valeur de b : "; cin >> b;
13      minmax(a, b, w, x);
14      cout << "Le plus petit vaut : " << w << endl;
15      cout << "Le plus grand vaut : " << x << endl;
16      return 0;
17  }
```

Fonction : passage par référence

Explication de l'exemple :

- Au lieu d'utiliser un passage de paramètres par pointeur comme dans l'exemple précédent, on peut utiliser un passage de paramètres par référence.
- Dans cet exemple, la fonction minmax() possède 4 paramètres : 2 entiers 'i' et 'j' passés par valeur et 2 entiers 'min' et 'max' passés par référence. 'i' et 'j' sont les paramètres en entrée de la fonction minmax(). 'min' et 'max' sont les paramètres en sortie de cette fonction. Le passage par référence permet donc d'avoir plus d'une variable de retour dans une fonction.
- Lors de l'écriture de la fonction minmax(), on remarquera le symbole & placé après le type qui indique que le paramètre est passé par référence.
- Lors de l'appel de minmax(), on remarquera aussi qu'il s'écrit minmax(a,b,w,x) ; sans symbole particulier. 'a' et 'b' sont passés par valeur et 'w' et 'x' sont passés par référence.

L'exécution de ce programme affichera :

Tapez la valeur de a : 25
Tapez la valeur de b : 12
Le plus petit vaut : 12
Le plus grand vaut : 25

Exercices (1/2)

1. Passage par référence

   Ecrire une fonction qui prend en paramètre un nombre réel x et modifie sa valeur en $\sqrt[4]{x}$. La fonction retourne un entier qui vaut 0 si la fonction n'a pas rencontré de problème et 1 si la valeur de x était négative (dans ce cas, la valeur finale de x restera inchangée). La fonction doit utiliser un passage par référence : utilisez-la pour calculer $\sqrt[4]{65\ 536} = \sqrt[4]{2^{16}}$.

2. Gestion de tableau 2D

   Construire un programme qui :

   - Génère une matrice (tableau 2D) carrée remplie de nombre aléatoires.
   - Calcule sa trace (= somme des éléments diagonaux)
   - Teste sa symétrie (i.e. teste si tab[i][j]==tab[j][i] $\forall$ i,j)
   - Calcule son produit matriciel avec elle-même si elle ne l'est pas

Exercices (2/2)

3. Ecrire une fonction qui, sans retourner aucune valeur (procédure), va permettre de :
   - Transposer une matrice carrée de dimension n
   - Calculer la trace de matrice transposée

   Utiliser ensuite cette fonction dans un programme qui va remplir une matrice de dimension n (choisie par l'utilisateur) de nombre aléatoires, la transposer et calculer la trace de la transposée grâce à la fonction préalablement définie et afficher cette trace à l'écran **via la fonction principale (main)**