*Chapitre 5*

# SOUS-PROGRAMMES

POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

# Les fonctions

- Réduit la complexité des programmes.
- Ensemble d'instructions servant à exécuter une tâche particulière dans la résolution d'un problème.
- Chaque fonction peut être testée et mise au point indépendamment des autres.
- Un programme peut se résumer à une séquence d'appels à des fonctions.

# Fonction

```
Type_resultat Nom(liste des types et  paramètres)
{
    partie déclarations de la fonction < au besoin >

    instructions de la fonction

    return expression_de_retour;   < au besoin >
}
```

# Fonction

```
                                        // Ordre d'exécution:
void afficherNombresImpairs()           //
{                                       //
    for (int i = 11; i < 30; i += 2)    // 4
        cout << setw(4) << i;           // 5 (cette ligne est exécutée 10 fois)
}                                       // 6 La fin de la fonction est atteinte ->
                                        //   retourne où la fonction à été appelée.
                                        //
int main()                              Début de l'exécution
{                                       //
    cout << "Nombres impairs de 10 à 30"; // 1
    cout << endl << endl;               // 2
    afficherNombresImpairs();           // 3 -> Va exécuter la fonction.
    cout << endl << endl;               // 7
    cout << "Affichage terminé";        // 8
}                                       // 9 -> La fin du programme est atteinte. Il est
possible de retourner un code d'erreur au programme qui a appelé notre programme; par défaut
le code est zéro, signifiant "pas d'erreur".
```

V_fonction.cpp

# Fonction: Les paramètres

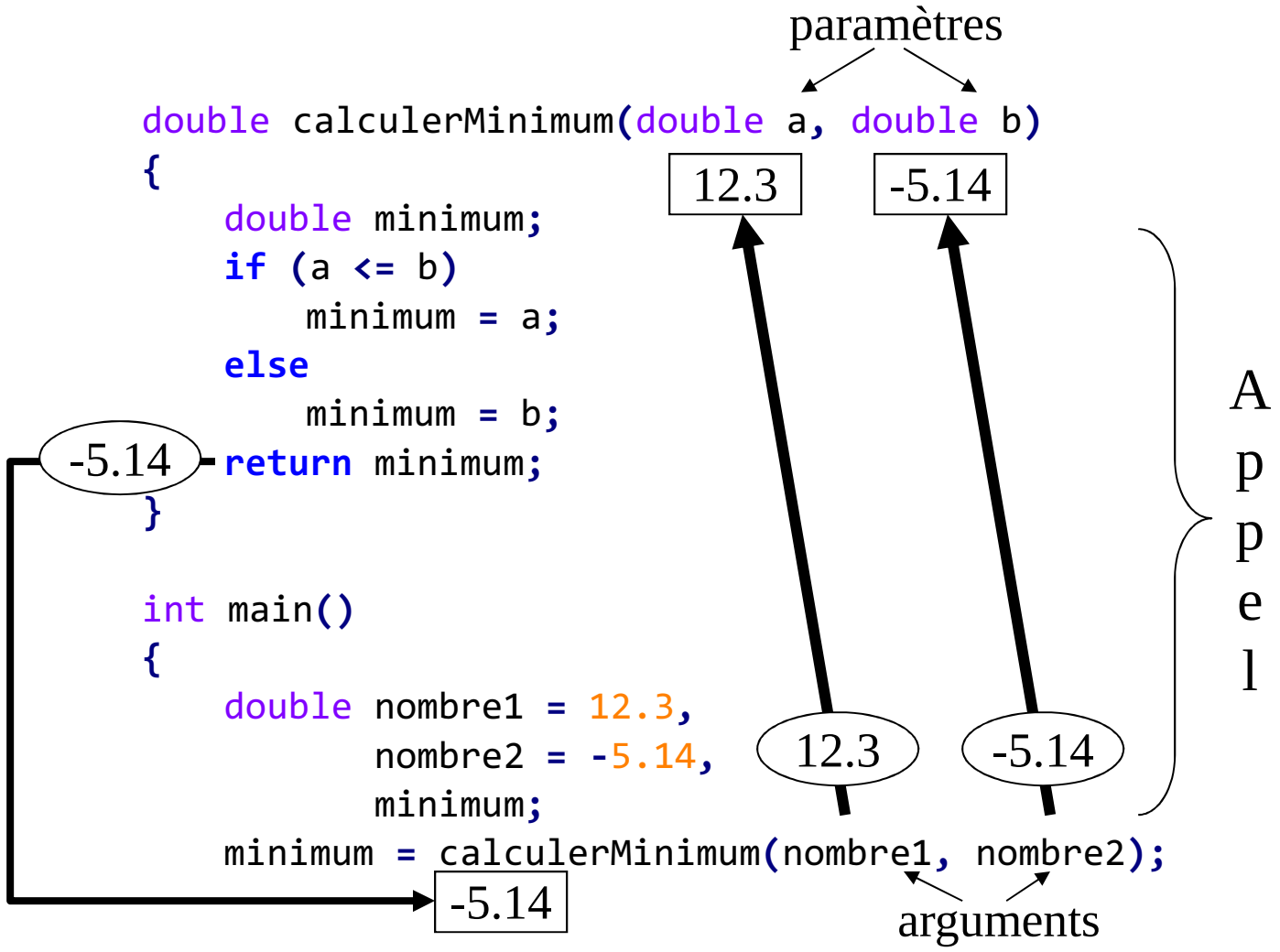
```
double calculerMinimum(double a, double b)
{
    double minimum;
    if (a <= b)
        minimum = a;
    else
        minimum = b;
    return minimum;
}

int main()
{
    double nombre1 = 12.3,
           nombre2 = -5.14,
           minimum;
    minimum = calculerMinimum(nombre1, nombre2);


    cout << "Le plus petit est " << minimum << endl;
}
```

V_parametres.cpp

# Fonction: Les paramètres




```
double calculerMinimum(double a, double b)
{
    double minimum;
    if (a <= b)
        minimum = a;
    else
        minimum = b;
    return minimum;
}

int main()
{
    double nombre1 = 12.3,
           nombre2 = -5.14,
           minimum;
    minimum = calculerMinimum(nombre1, nombre2);

    cout << "Le plus petit est " << minimum << endl;
}
```

V_parametres.cpp

# Fonction: les paramètres

- Les paramètres servent à augmenter l'utilisation de la fonction en obtenant de l'information de l'extérieur lors de l'appel.
- Il doit y avoir le même nombre d'arguments lors de l'appel que de paramètres définis dans la fonction.
- L'ordre des arguments dans l'appel de la fonction est important puisque c'est la position de l'argument à l'appel qui détermine le paramètre associé.

# Transmission de paramètres

## PAR VALEUR

- La valeur de l'argument est transmise au paramètre correspondant.
- Le paramètre est initialisé à la valeur reçue en effectuant une copie de la valeur.
- Lorsque le paramètre est modifié, l'argument n'est pas modifié.

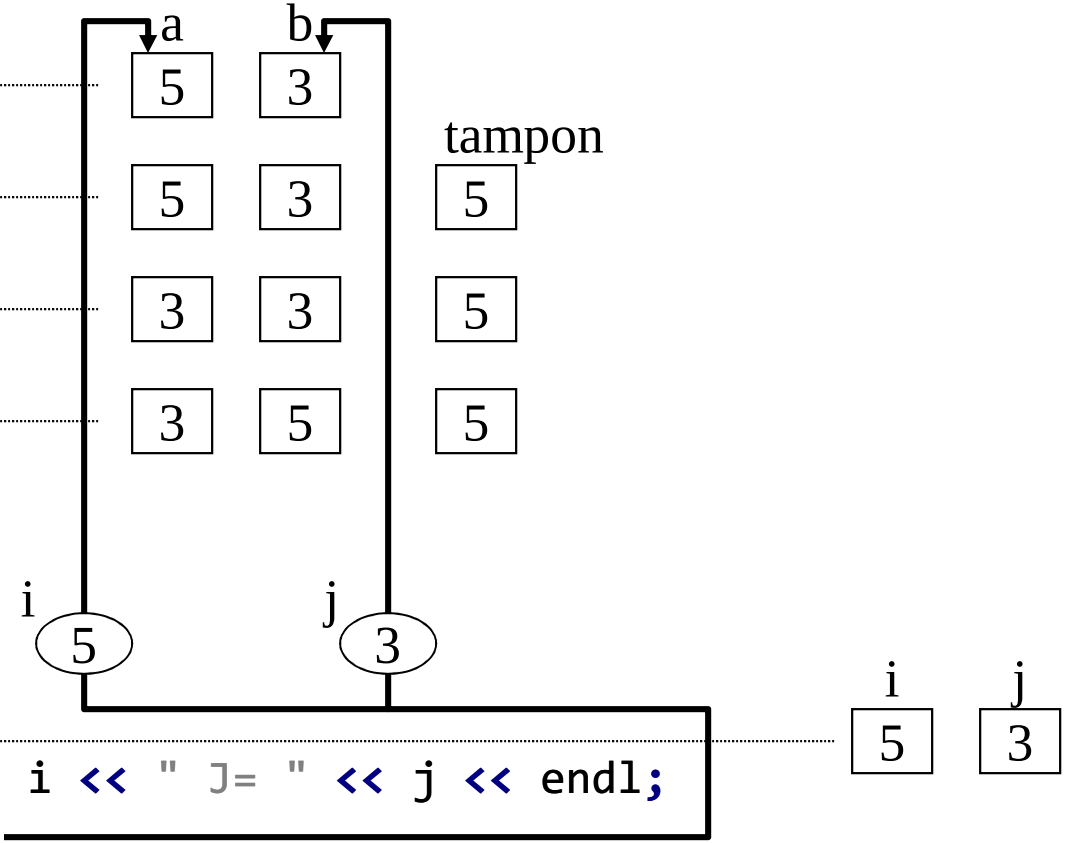
## PAR ADRESSE

- L'adresse de l'argument est transmise au paramètre correspondant.
- Le paramètre est initialisé à l'adresse (de l'argument) reçue, et par le fait même réfère à la valeur de l'argument.
- Lorsque le paramètre est modifié, l'argument est modifié.

# Transmission par valeur

```
void permuter(int a, int b)     // Passage par valeur.
{
    int tampon = a;
    a = b;
    b = tampon;
}

int main()
{
    int i = 5;
    int j = 3;
    cout << "I= " << i << " J= " << j << endl;
    permuter(i, j);
    cout << "I= " << i << " J= " << j << endl;
}
```

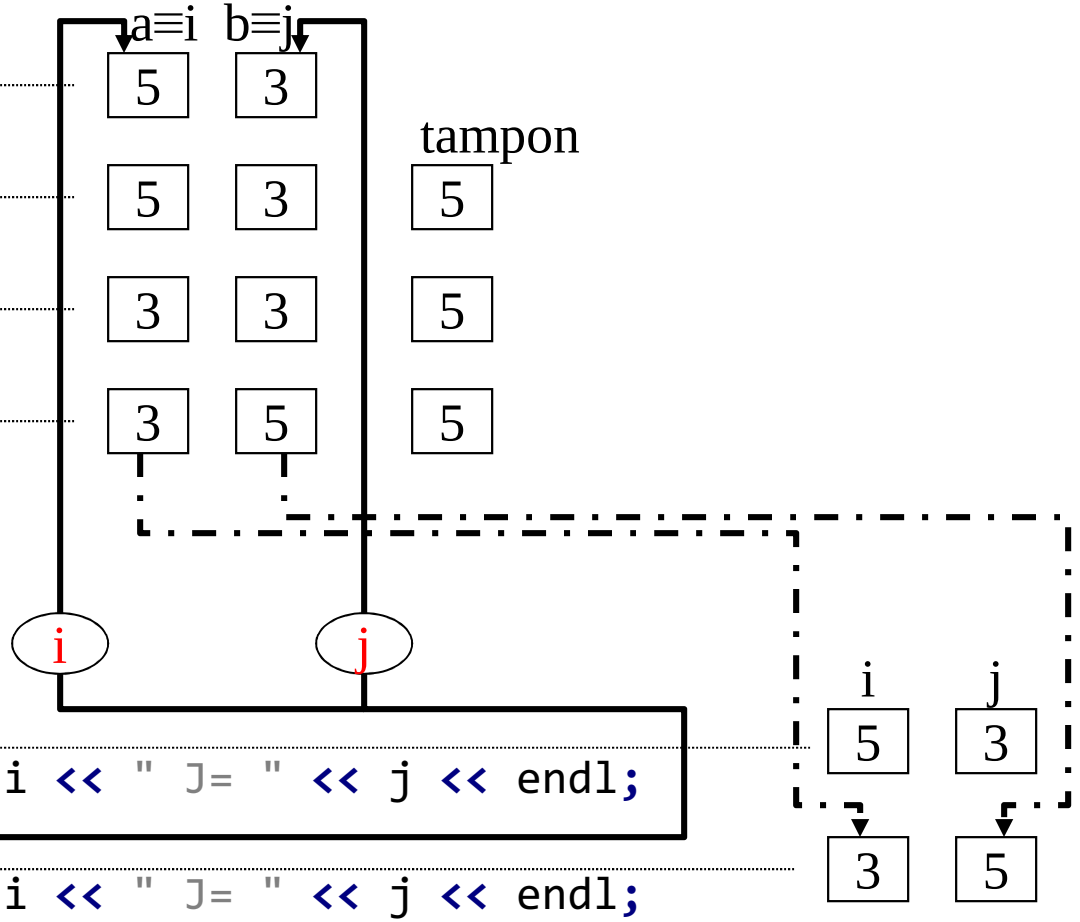



V_passage.cpp

# Transmission par adresse

```
void permuter(int& a, int& b)  // Passage par adresse.
{
    int tampon = a;
    a = b;
    b = tampon;
}

int main()
{
    int i = 5;
    int j = 3;
    cout << "I= " << i << " J= " << j << endl;
    permuter(i, j);
    cout << "I= " << i << " J= " << j << endl;
}
```




V_passage.cpp

# Transmission par valeur et par adresse

```
#include <iostream>
#include <iomanip>
using namespace std;

void modifier(int entier1, int& entier2)
{
    entier1 = entier1 + 100;
    entier2 = entier2 + 100;
    cout << "(2) Les valeurs sont ";
    cout << setw(4) << entier1 << setw(4) << entier2 << endl;
}

int main()
{
    int compteur = 7, indice = 12;

    cout << "(1) Les valeurs sont ";
    cout << setw(4) << compteur << setw(4) << indice << endl;

    modifier(compteur, indice);

    cout << "(3) Les valeurs sont ";
    cout << setw(4) << compteur << setw(4) << indice << endl;
}
```

Résultat de l'exécution:

```
(1) Les valeurs sont    7  12
(2) Les valeurs sont  107 112
(3) Les valeurs sont    7 112
```

V_passage_valeur_adresse.cpp

# Transmission de paramètres

- La transmission par valeur sert à fournir à la fonction une valeur dont elle a besoin pour exécuter correctement ses instructions. On qualifie le paramètre (IN).
- La transmission par adresse sert à retourner une valeur au point d'appel via l'argument. Dans ce cas on qualifie le paramètre (OUT). La fonction peut, au besoin, utiliser une valeur initiale. Dans ce cas on qualifie le paramètre (IN/OUT).
- La transmission par adresse peut aussi servir à éviter une copie de valeur. On qualifie alors le paramètre de (IN), et ajoutons « const » devant le type.

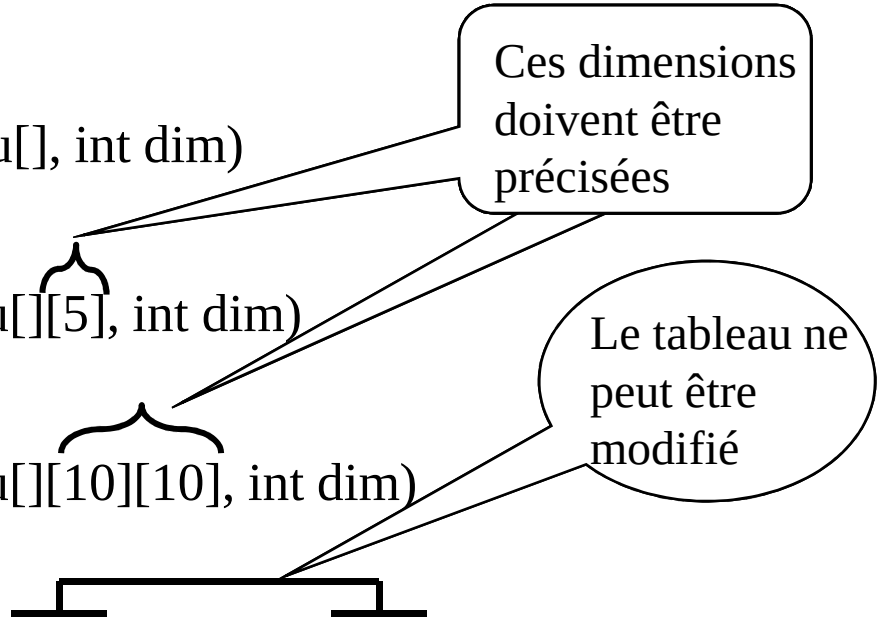
# Transmission de paramètres

- Il doit y avoir correspondance entre les arguments de l'appel et les paramètres de la fonction pour ce qui est de leur nombre et leur type.
- L'argument dans un appel doit absolument être une variable si le paramètre correspondant est transmis par adresse non « const ». Il ne peut être une constante ni une expression.
- Si le paramètre est transmis par valeur, l'argument dans l'appel peut être une variable, une constante ou une expression.

IMPORTANT

# Un tableau comme paramètre

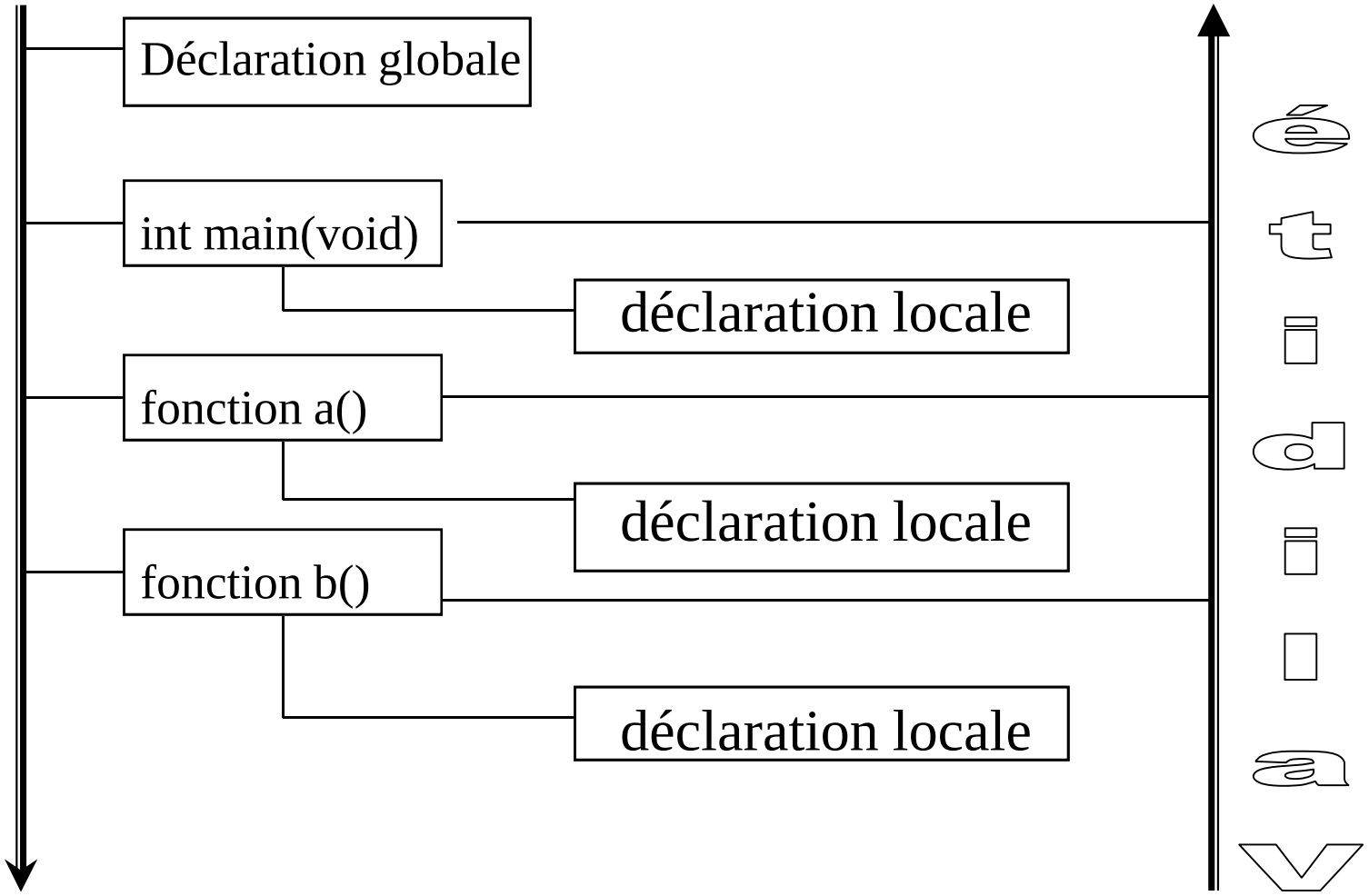
- Toujours transmis par adresse (sans &) puisque l'identificateur d'un tableau contient comme information l'adresse du premier élément du tableau.
  - void afficher_1D(int tableau[], int dim)
  - void afficher_2D(int tableau[][5], int dim)
  - void afficher_3D(int tableau[][10][10], int dim)
  - void somme(int result[10], const int a[10], const int b[10])



# Exercice

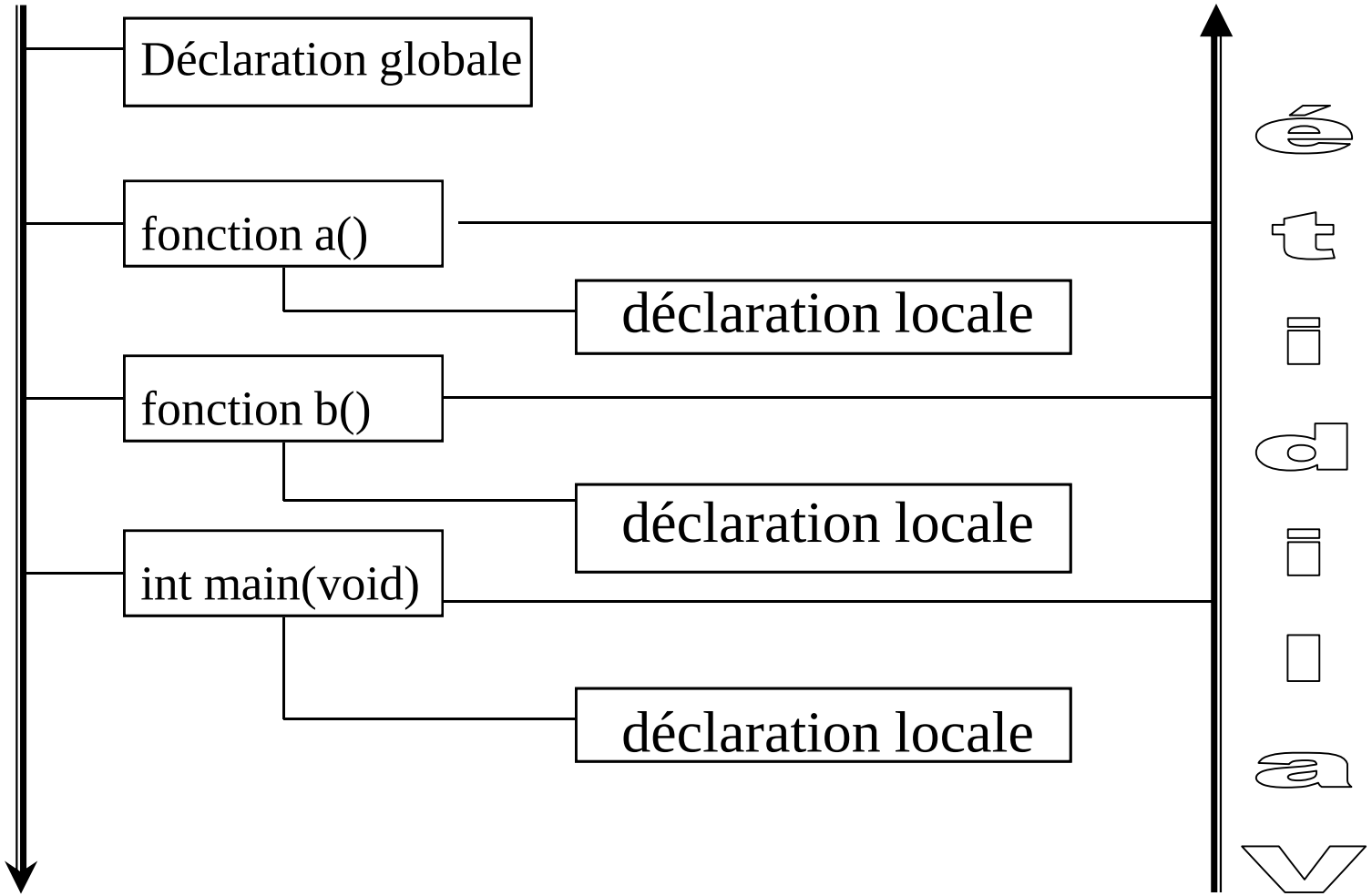
- Écrire un programme qui calcule la note obtenue lors d'une compétition. Ce programme contient:
  - une fonction qui lit la note des 8 juges;
  - une fonction qui calcule la note en les additionnant sauf la note maximale et la note minimale;
  - une fonction qui trouve la note min et la note max.

# Domaine de validité



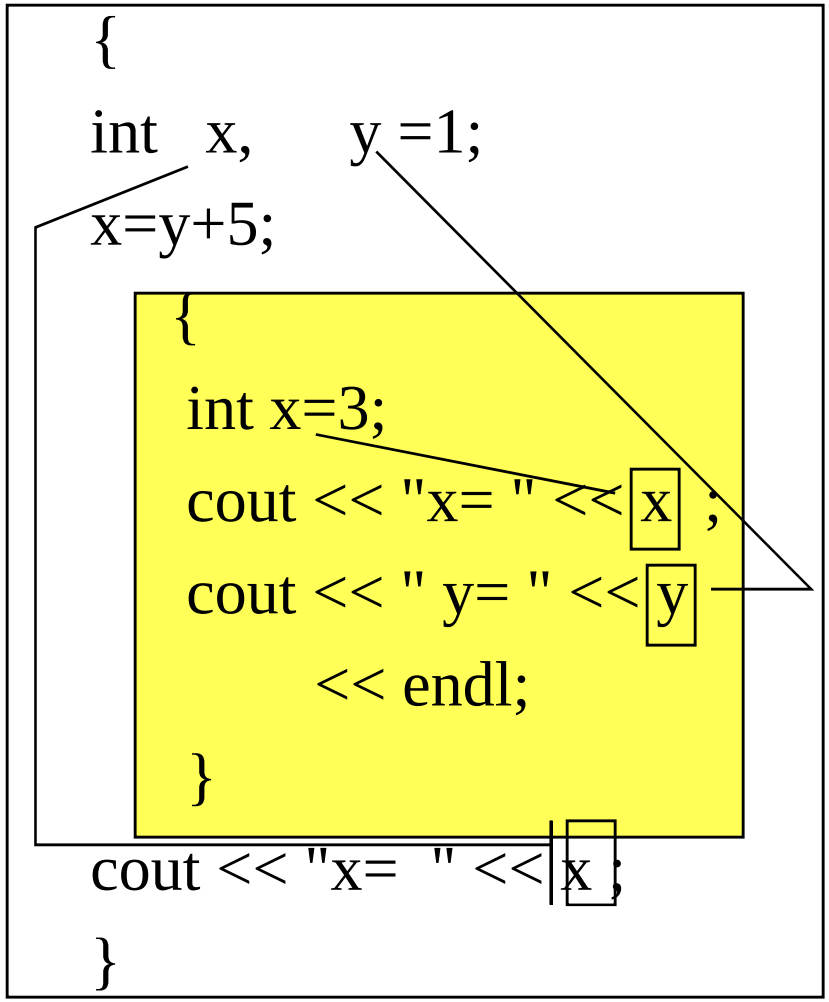

La fonction main() n'a accès à aucune fonction ==> prototypage

# Domaine de validité




La fonction main() a accès à toutes les fonctions déclarées auparavant.

# Domaine de validité




Affichage

x= 3 y= 1

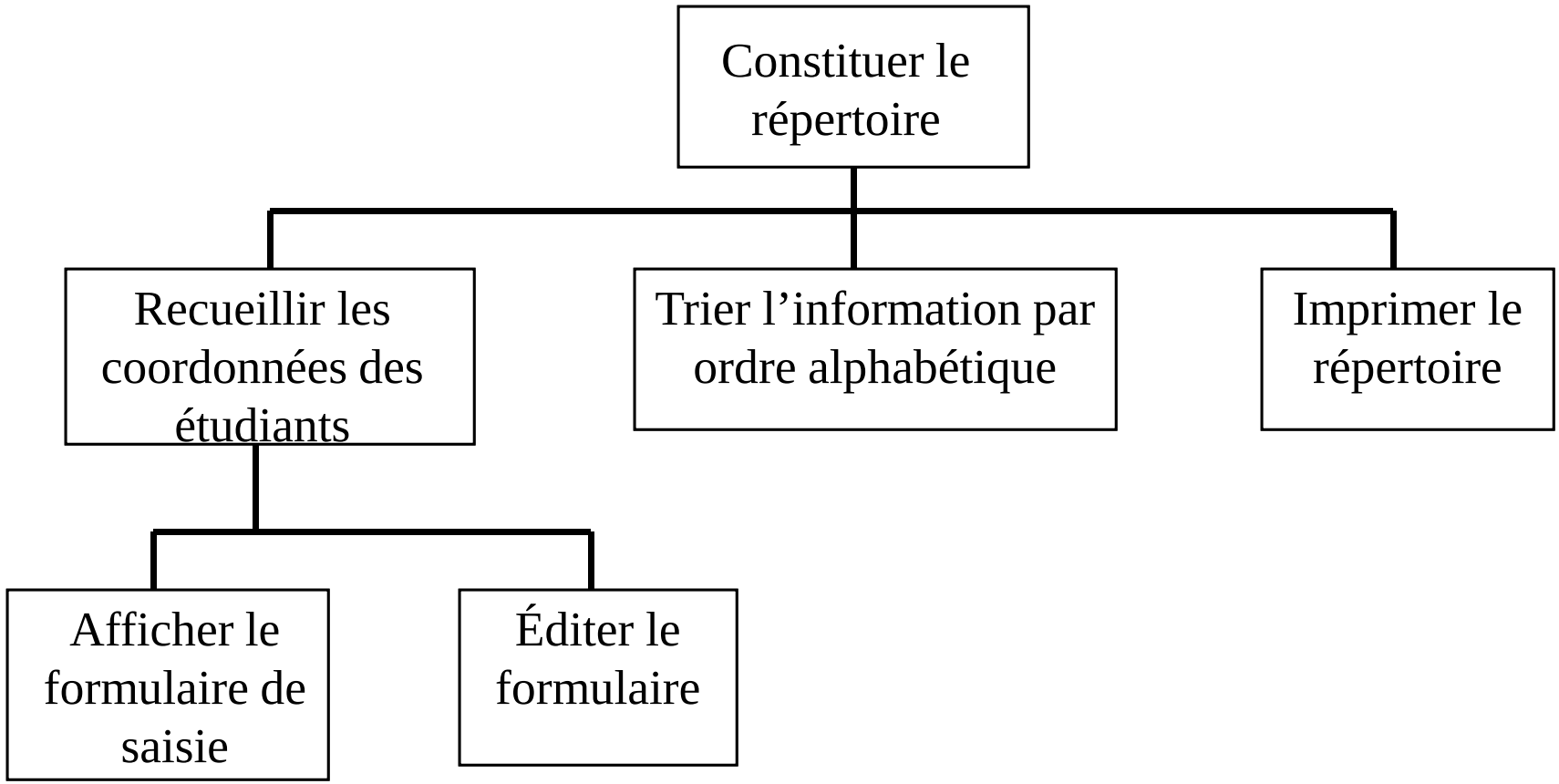
x= 6

# Fonction

- void spécifie qu'il n'y a pas de valeur de retour
- Mode de transmission de paramètres
  - ⇨Transmission par valeur
  - ⇨Transmission par adresse à l'aide de référence [C++]
  - ⇨Transmission par adresse à l'aide de pointeur [C]
- Initialisation des paramètres (Notion utilisée au chap.10)
- Surcharge de fonction (Notion utilisée au chap.10)
- Fonction récursive  (n'est pas au programme du cours)

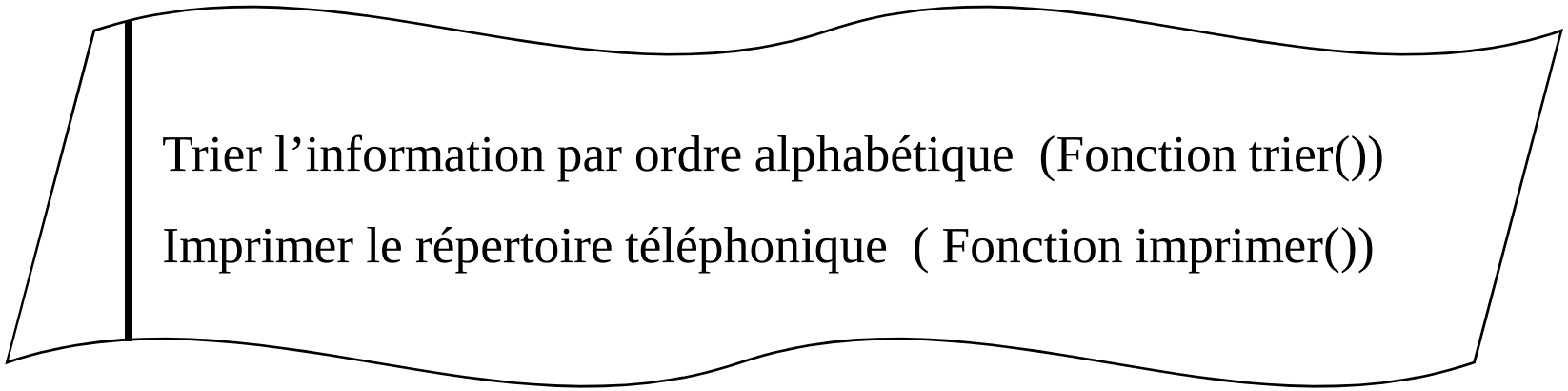
# Algorithme

- Décomposition d'une tâche en sous-tâche



# Algorithme

- Il faut rédiger un algorithme par fonction.
- Une opération correspondant à la tâche d'une fonction est précisée en ajoutant le nom de la fonction entre parenthèses.



# Emplacement des fonctions

```
#include <iostream>
#include <iomanip>
using namespace std;

// Computes the total cost, including 5% sales tax,
// on nItems items at a cost of itemPrice each.
double totalCost(int nItems, double itemPrice);

int main()
{
    double pricePerItem;
    int numberOfItems;
    cout << "Enter the number of items purchased: ";
    cin >> numberOfItems;
    cout << "Enter the price per item $";
    cin >> pricePerItem;

    double bill = totalCost(numberOfItems, pricePerItem);

    cout << fixed << showpoint << setprecision(2);
    cout << numberOfItems << " items at "
         << "$" << pricePerItem << " each." << endl
         << "Final bill, including tax, is $" << bill
         << endl;
}

double totalCost(int nItems, double itemPrice)
{
    static const double TAXRATE = 0.05; // 5% sales tax
    double subtotal = itemPrice * nItems;
    return subtotal + subtotal*TAXRATE;
}
```

Déclaration de fonction
ou prototype
(doit être avant l'utilisation)

Appel de fonction

Entête de fonction

Corps de la fonction

Définition de la fonction
(peut être après l'utilisation)

V_emplacement_apres.cpp

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

// Computes the total cost, including 5% sales tax,
// on nItems items at a cost of itemPrice each.
double totalCost(int nItems, double itemPrice)
{
    static const double TAXRATE = 0.05; // 5% sales tax
    double subtotal = itemPrice * nItems;
    return subtotal + subtotal*TAXRATE;
}

int main()
{
    double pricePerItem;
    int numberOfItems;
    cout << "Enter the number of items purchased: ";
    cin >> numberOfItems;
    cout << "Enter the price per item $";
    cin >> pricePerItem;

    double bill = totalCost(numberOfItems, pricePerItem);

    cout << fixed << showpoint << setprecision(2);
    cout << numberOfItems << " items at "
         << "$" << pricePerItem << " each." << endl
         << "Final bill, including tax, is $" << bill
         << endl;
}
```

Entête de fonction

Corps de la fonction

Définition de la fonction
(doit être avant l'utilisation si pas de prototype)

Appel de fonction

V_emplacement_avant.cpp

# Régles de portée (visibilitée)

- Les fonctions sont des unités.
- Les variables des fonctions sont distinctes des autres variables déclarées à l'extérieur de la définition de la fonction.
- Des variables ayant le même identificateur dans différentes fonctions, ou différents blocs d'une fonction, sont distinctes.

# Variables locales

```
double moyenne(double tableau[], int nombre);
int main()
{ double max, min, moy;
  double liste[20];
  int nombreElelments;
……
  moy = moyenne(liste, nombreElements);
……

}

double moyenne(double tableau[], int nombre)
{ int i;
  double moy;
……
}
```

Variables locales au main

Variables locales à la fonction moyenne

Il n'y a aucun lien entre les deux variables qui ont le nom 'moy'.

# Variables locales (suite)

```
void main()
{
   …….
   for (int i = 0; i < 10; i++)
       for (int j = 0; j < 10; j++) {
              ……..
       }
   while (!trouver) {
       int i, temp;
       ……..
   }



}
```

i, j variables locales aux blocs for

i, temp variables locales au bloc while

Il n'y a aucun lien entre les deux variables qui ont le nom 'i'.

# Variables globales

- Une variable globale est accessible par toutes les fonctions définies dans le fichier y compris le main.

# Variables Globales

```
double taux;

double moyenne( double tableau[], int nombre)
{ int i;
  double moy;
…… // utilisation de la variable taux
}
int main()
{ double max, min, moy;
  double liste[20];
  int nombreElelments;
…… // utilisation de la variable taux
  moy = moyenne(liste, nombreElements);
……

}
```

Variable globale

Éviter d'utiliser les variables globales dans le cours INF1005C
(guide de codage point 48)

Les constantes globales sont tout à fait permises.

# « Variables constantes »

```
void main()
{
    static const double pi = 3.1416;

}
```

Constante locale

Les constantes « ne peuvent pas être modifiées » après leur déclaration.
Par contre:

```
void f(int x)
{
    const int y = x + 1;
    static const int z = x + 1;
    …
}
```

'y' a une valeur différente à chaque appel

'z' a la même valeur pour tous les appels;
le premier appel lui fixe sa valeur

Les variables les plus constantes sont donc 'static const'.

# Transmission des types composés (struct et types C++)

- Les types composés peuvent être transmis/retournés par valeur
  - Copie complète de tous les champs
  - Contrairement aux tableaux

par adresse (4 octets) ↓ par valeur (3608 octets + taille des textes) ↓

```
void tableauVsStruct(Ordinateur tableau[MAX_ORDIS], ListeOrdi structure)
{
    tableau[0].cpuGHz = 2.2;          // modifie le tableau original
    structure.liste[0].cpuGHz = 2.2; // modifie la copie de la structure
}
```

( avec les mêmes définitions de Ordinateur et ListeOrdi que l'exemple au chapitre 4; MAX_ORDIS = 50 )

# Transmission des types composés (struct et types C++) (suite)

- Généralement transmis par adresse pour
  - Performance : le type peut être gros (`string`)
  - Simplicité : copier la valeur peut être problématique (…`stream`)
  - Ne nuit pas la lisibilité

```
void afficherOrdinateur(const Ordinateur& ordinateur)
{
    cout << ordinateur.compagnie << '\t'
         << ordinateur.format << '\t'
         << ordinateur.cpuGHz << '\t'
         << ordinateur.memoireRamGiB << '\t'
         << ordinateur.tailleEcranPouces << '\t'
         << ordinateur.memoireDisqueGB << endl;
}
```

- Attention de spécifier correctement le `const` si paramètre IN:
  - Soit l'expression: afficher("bonjour");
  - L'entête de fonction doit être: void afficher(**const** string& texte);

# Transmission des types composés (struct et types C++) (suite)

- Les « stream » peuvent seulement être transmis par référence
  - Jamais const (ou presque): déplacer la tête de lecture est une modification

**& pas const** paramètre OUT

```
void lireOrdinateur1(istream& fichier, Ordinateur& ordinateur)
{
    fichier >> ordinateur.format
            >> ordinateur.compagnie
            >> ordinateur.tailleEcranPouces
            >> ordinateur.cpuGHz
            >> ordinateur.memoireRamGiB
            >> ordinateur.memoireDisqueGB;
}
```

V_structTab.cpp

# Transmission des types composés (struct et types C++) (suite)

- Le retour par valeur peut être optimisé par le compilateur
- Règle générale: écrire ce qui est le plus lisible
  - Si la performance est cruciale (jamais dans ce cours) vérifier si l'optimisation est faite

Probable copie de valeur, mais généralement plus lisible qu'un paramètre OUT

```
Ordinateur lireOrdinateur2(istream& fichier)
{
    Ordinateur ordinateur;
    fichier >> ordinateur.format
            >> ordinateur.compagnie
            >> ordinateur.tailleEcranPouces
            >> ordinateur.cpuGHz
            >> ordinateur.memoireRamGiB
            >> ordinateur.memoireDisqueGB;
    return ordinateur;
}
```

V_structTab.cpp

# Transmission de tableaux

- Soit les variables:

```
int vals[] = { 1, 5, 3 };
int nVals = int(size(vals)); // PAS 'sizeof' (nombre d'octets).
```

- Comment passer ce tableau à une fonction?
- Passage d'un tableau (plus tôt dans ce chapitre):

```
void afficher(const int tableau[], int dim) {
    for (int i = 0; i < dim; i++)
        cout << tableau[i] << " ";
```

  - Doit passer 2 paramètres pour un seul tableau
  - Lien entre tableau et dim pas très clair (inconnu par C++)
  - Appel:

```
afficher(vals, nVals);
```

- Comment lier clairement le tableau et sa taille?

V_passage_tableau.cpp

# Transmission de tableaux (suite)

- Utilisation d'une struct:

```
struct TableauInt {
    const int* valeurs;  // Équivalent du paramètre const int tableau[]
    int        nValeurs; // Équivalent du paramètre int dim
};
void afficher_v2(TableauInt tableau) {
    for (int i = 0; i < tableau.nValeurs; i++)
        cout << tableau.valeurs[i] << " ";
```

  - Lien évident entre le tableau et sa taille
  - Appels:

```
afficher_v2({vals, nVals});  // Accolades pour construire la struct.
TableauInt tab = { vals, nVals };
afficher_v2(tab);            // Passage d'une struct existante.
```

- Mais
  - Doit passer la bonne taille
  - Doit définir une struct par type
- Peut-on faire mieux?

V_passage_tableau.cpp

# Transmission de tableaux en C++20

(doit installer la bibliothèque; la norme sort seulement en 2020)

- Type « span » : généralisation de la struct précédente

```
void afficher_v3(span<const int> tableau) {
    for (int i = 0; i < tableau.size(); i++) // .size() comme pour string.
        cout << tableau[i] << " ";           // Accès comme un tableau.
```

  - Appels:

```
afficher_v3(vals);           // Conversion en span implicite.
afficher_v3({vals, nVals});  // Peut construire span comme la struct.
span s = vals;
afficher_v3(s.subspan(1, 2)); // Prend une partie d'un span.
          // .subspan(index_début, nombre_éléments)
afficher(s.data(), s.size()); // Pour utiliser l'ancienne fonction.
```

- Syntaxe dite *template* : le type des éléments entre < >
- Pas encore de taille variable multiples dimensions
  - span<int[10]> tableau fonctionne pour l'équivalent de int tableau[][10]