Chapitre 3 : Les fonctions en C++

I. Introduction aux fonctions : les fonctions standards

A. Notion de Fonction

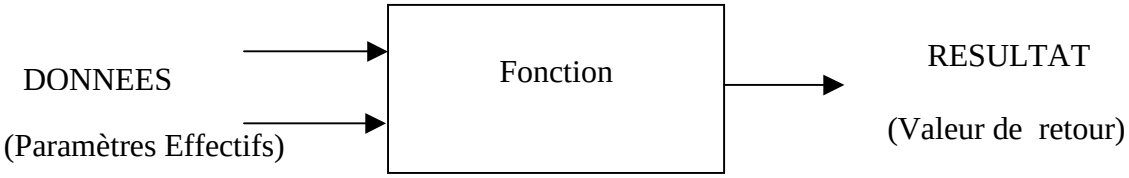
Imaginons que dans un programme, vous ayez besoin de calculer une racine carrée. Rappelons que l'opérateur racine carrée n'existe pas en C++, pas plus que l'opérateur puissance. Comment faire?
Deux possibilités s'offrent à vous :

- soit vous écrivez vous-même entièrement la partie du programme qui permet d'extraire une racine carrée, à partir des opérateurs de base du langage (les opérateurs primitifs). Mais le programme d'extraction d'une racine est délicat à écrire voir impossible quand on n'en maîtrise pas l'algorithme.
- soit vous utilisez une **fonction déjà existante** qui permet d'extraire une racine carrée. Une telle fonction existe dans le langage C++ : c'est une fonction **standard** appelée sqrt.

Une fonction est un **morceau de programme autonome, utilisé par un autre programme pour réaliser une opération précise**.

- Une fonction est standard quand elle est livrée avec l'EDI du langage. Le programmeur peut l'utiliser comme s'il s'agissait d'un opérateur du langage.
- Une fonction peut aussi être écrite par le programmeur pour ses propres besoins. Nous verrons comment écrire et utiliser une fonction dans la partie suivante.

Une fonction au sens strict peut être représentée par une boite noire (un mécanisme invisible) qui donne un et un seul résultat à partir d'une ou plusieurs données (voire aucune).



- Le résultat d'une fonction est appelée **VALEUR DE RETOUR**.
- Les données à partir desquelles une fonction calcule son résultat sont appelées **PARAMETRES EFFECTIFS** ou **ARGUMENTS**

En C++, la notion de fonction est plus large : en C++, une fonction peut ne rien retourner. En C++, on utilise le mot fonction pour désigner tous les sous-programmes, y compris les procédures. (voir le cours d'algo)

B. Utilisation des fonctions standards

En C++, les fonctions standards sont regroupées dans des **bibliothèques** (library en anglais). Pour utiliser une fonction standard, il faut inclure le **fichier d'en-tête** où elle est déclarée.
Par exemple, pour utiliser la fonction getch( ), qui se contente de saisir sans retour à l'écran d'un caractère tapé au clavier, il faut inclure le fichier d'en-tête <conio.h>.

Ensuite, pour exécuter une fonction à l'intérieur d'un programme, il faut effectuer un **APPEL** de cette fonction. L'appel d'une fonction consiste tout simplement à écrire son nom, suivi entre parenthèses des paramètres effectifs (les données).

Exemple
Voilà un programme qui permet tout simplement d'afficher la racine carrée d'un nombre entier saisi par l'utilisateur. Pour cela, on utilise la fonction sqrt déclarée dans le fichier d'en-tête math.h.

```
#include <iostream.h>
#include <math.h>

main( )
{
   int n;
   cout << "Taper un nombre ";
   cin >> n;
   cout << "La racine carrée de ce nombre est "<< sqrt(n) ;
}
```

inclusion du FICHIER d'EN-TETE

APPEL de la fonction sqrt

PARAMETRE effectif ou ARGUMENT

A l'exécution, l'appel d'une fonction est remplacé par la valeur retournée (résultat) : sqrt(n) correspond à la valeur de la racine carrée de n. Donc on peut utiliser l'appel d'une fonction comme toute autre valeur : on peut l'afficher, l'affecter à une variable ou l'utiliser dans une expression (calcul, condition,…).

Cas particulier :
Dans le cas d'une fonction qui ne retourne rien (fonction void ou procédure en algorithmique), l'appel correspond à une instruction à part entière et non à une valeur. L'appel d'une procédure ne peut en aucun cas se trouver à l'intérieure d'une expression.

C. Les spécifications d'une fonction

Pour pouvoir utiliser une fonction, il est inutile de connaître comment elle fonctionne. Il suffit de savoir à quoi elle sert et comment s'en servir. (savez vous exactement comment fonctionne un téléviseur? Non, et pourtant vous savez vous en servir !)

Les spécifications d'une fonction servent justement à indiquer quand et comment s'en servir :

Les spécifications d'une fonction (ou d'une procédure) rassemblent les informations nécessaires à son utilisation c'est-à-dire :
son nom, son rôle, ses paramètres (leur ordre, leur type), le type de la valeur retournée et sa localisation (en C++, le fichier d'en-tête dans lequel elle est déclarée).

Exemple : spécifications de la fonction sqrt( ) du C++

nom rôle paramètre

La fonction **sqrt** **extrait la racine carrée** **du nombre de type double** passé en paramètre. La valeur retournée est de **type double**. Elle est déclarée dans le fichier d'en-tête **math.h**

type de la valeur retourné localisation

On peut trouver les spécifications des fonctions standards dans l'aide – en anglais !- de l'environnement de développement. **Quand un programmeur écrit une fonction, il doit toujours en fournir les spécifications pour que les autres programmeurs puissent l'utiliser.**

D. Les fonctions standards les plus utilisées

- Les **fonctions mathématiques** du fichier d'en-tête **math.h**

Fonction	*Description*	*Exemple*
ceil (x)	retourne la valeur de x arrondie à l'entier supérieur	ceil(3.1416) retourne 4.0
fabs(x)	retourne la valeur absolue de x	fabs(-2) retourne 2.0
floor(x)	retourne la valeur de x arrondie à l'entier inférieur	floor(3.1416) retourne 3.0
pow(x, p)	retourne x à la puissance p	pow(2, 3) retourne 8 (2^3)
sqrt(x)	retourne la racine carrée de x	sqrt(2) retourne 1.41421

Toutes ces fonctions retournent une valeur de type double. Les paramètres sont aussi de type double. Mais si on utilise des paramètres de type int ou float, ils seront automatiquement convertis en double.

- **Autres fonctions utiles**

- **clrscr**( ) conio.h
  efface l'écran

- **random**(n) stdlib.h
retourne un entier aléatoire compris entre 0 et n (exclu). A chaque fois que cette fonction est appelée, elle retourne un nombre différent. Mais à chaque utilisation du programme, c'est la même série de nombre aléatoire qui est générée. Pour éviter cela, on réinitialise la liste de nombre aléatoires avec la fonction (procédure) suivante :
- **randomize**( ) stdlib.h
randomize( ) ne doit être appelé qu'une seule fois en début de programme (même si on utilise plusieurs fois random par la suite.

- **Exemple** : Simulation d'un tirage à pile ou face

```
# include <iostream.h>
# include <conio.h>
# include <stdlib.h>
main()
{
  int res;
  randomize( );        //réinitialisation du générateur de nombres aléatoires
  res = random(2);     //permet de choisir au hasard entre 0 et 1
  if (res == 0)        //on définit arbitrairement que 0 est PILE et on affiche le résultat
        cout << "PILE";
  else
        cout << "FACE";
  getch( );
}
```

II. La création de fonctions

En vertu des principes de la programmation modulaire, qui préconise de découper le code des applications en petites unités, les programmeurs sont conduits à créer leurs propres sous-programmes. Nous allons voir comment faire en C++.

Spécificités du C et C++:

- **tous les sous-programmes sont appelés FONCTIONS** même ceux qui ne renvoie rien (et qu'on appelle procédure en algorithmique et dans les autres langages).

- **le programme principal est lui même une fonction**, appelée obligatoirement main( ) car c'est la toute première fonction appelée à l'exécution du programme.

Définition et appel d'une fonction (au sens algorithmique)

➢ Définition

Syntaxe:

```
type_retourné nom_fonction (type_paramètre nom_paramètre, …)        //en-tête
{
        /*corps de la fonction*/
        return valeur_retournée;
}
```

Exemple: **définition** de la fonction somcarre ( )

```
double somcarre (float a, float b)
{
        double sc;
        sc = a * a + b * b;
        return  sc;
}
```

Attention : il faut indiquer le type devant chaque paramètre, même si c'est le même
(sinon, par défaut, le paramètre est de type int)

ou plus simplement

```
double somcarre (float a, float b)
{
    return a*a + b*b;
}
```

➢ L'appel d'une fonction

L'appel d'une fonction est utilisé dans une instruction comme une valeur. On peut même utiliser l'appel d'une fonction comme valeur de paramètre pour une autre fonction.

ex: appel de la fonction somcarre pour calculer la variance v de deux nombres n1 et n2.

```
double v;
float n1, n2;
…//saisie de n1 et n2
v = (somcarre(n1, n2) – (n1+n2)) / n;
```

Remarquez que les types des paramètres effectifs correspondent au type des paramètres formels correspondants.

Définition et appel d'une procédure (au sens algorithmique) : les fonctions void

Le mot procédure n'existe pas en C++. On parle de **fonction void**. (void veut dire vide en anglais). Il faut obligatoirement indiquer void comme type retourné. Une fonction void ne contient pas d'instruction return.

```
void Affiche_acceuil (char s, char em)
{
        if ( s == 'm' )
              cout << "Bonjour Monsieur";
        else
        {
              if (em== 'c' )
                  cout <<"Bonjour Mademoiselle";
              else
                  cout << "Bonjour Madame";
        }
}
```

Si vous oubliez le void, le compilateur affectera int comme type de retour (car int est le type par défaut), ce qui entraînera une erreur.

L'appel d'une fonction void correspond à une instruction à part entière et se fait donc sur une ligne à part. On ne peut pas utiliser l'appel à l'intérieur d'une instruction puisqu'il ne correspond à aucune valeur.

Exemple:

```
# include …
main( )
{
   char hf, ec;
   cout<< "Tapez m si vous êtes un homme ou f si vous êtes une femme";
   cin >> hf;
   cout << "Tapez c si vous êtes célibataire ou m si vous êtes marié";
   cin >> ec;
   Affiche_acceuil(hf, em);  //l'appel lui-même est une instruction
   getch( );
}
```

Agencement des fonctions entre elles

Le **compilateur n'accepte d'appeler une fonction que s'il connaît toutes ses caractéristiques** (nom, type de la valeur de retour et type des paramètres).
Si l'appel d'une fonction se trouve après la définition de cette fonction, celle-ci est connue du compilateur, donc cette méthode va fonctionner.
Mais cette solution n'est guère utilisée en pratique. En réalité, une fonction est seulement déclarée avant d'être appelée et sa définition est reportée après celle de la fonction appelante.

Pour les fonctions définies par le programmeurs, deux méthodes sont possibles:

- soit on définit complètement la fonction appelée avant la fonction appelante
- soit on commence par déclarer le prototype de la fonction appelée, on définit la fonction appelante puis on définit complètement la fonction appelée (à la suite de la fonction appelante ou dans un autre fichier).

La deuxième méthode est la plus courante et c'est celle que nous utiliserons.

➢ Déclaration d'une fonction : le prototype

La déclaration d'une fonction passe par l'écriture de son prototype. Le prototype d'une fonction a la même syntaxe que son en-tête, et est terminé par un ;
La déclaration des fonctions se fait au début du code source, avant le main( ).

Exemple complet

```
// 1) inclusion des fichiers d'en-tête pour utiliser les fonctions prédéfinies
# include <iostream.h>
# include <math.h>

// 2)  déclaration = prototype de la fonction somcarre ( )
float somcarre (float a, float b);

// 3) définition de la fonction main ( ) : programme principal
main ( )
{
double v;
float n1, n2;
cout << "entrez deux nombres";
cin >> n1, n2;
v = (somcarre(n1, n2) – (n1+n2)) / n;   // instruction contenant l'appel de la fonction somcarre ( )
cout << "l'écart type de ces nombres est" << sqrt(v);
}

// 4) définition de la fonction somcarre ( )
float somcarre (float a, float b)     //en-tête
{
        float sc;
        sc = a * a + b * b;
        return sc;
}
```

Remarque :
Le prototype des fonctions prédéfinies est contenue dans le fichier d'en-tête correspondant à la directive #include … Le compilateur connaît toutes les fonctions définies dans les fichiers d'en-tête inclus dans le code source.

Le passage des paramètres

➢ le passage par valeur : passage par défaut

Par défaut les paramètres d'une fonction en C++ sont passés par valeur. Cela signifie qu'à l'appel, la valeur des paramètres effectifs est copiée dans les paramètres formels correspondants. Au retour d'appel, la valeur des paramètres effectifs est inchangée, même si les paramètres formels ont été modifiés par la fonction.

Ce type de passage de paramètre par valeur est donc bien adapté pour les paramètres correspondant à des données. En revanche, il est inadapté pour gérer des paramètres résultats ou données-résultats.

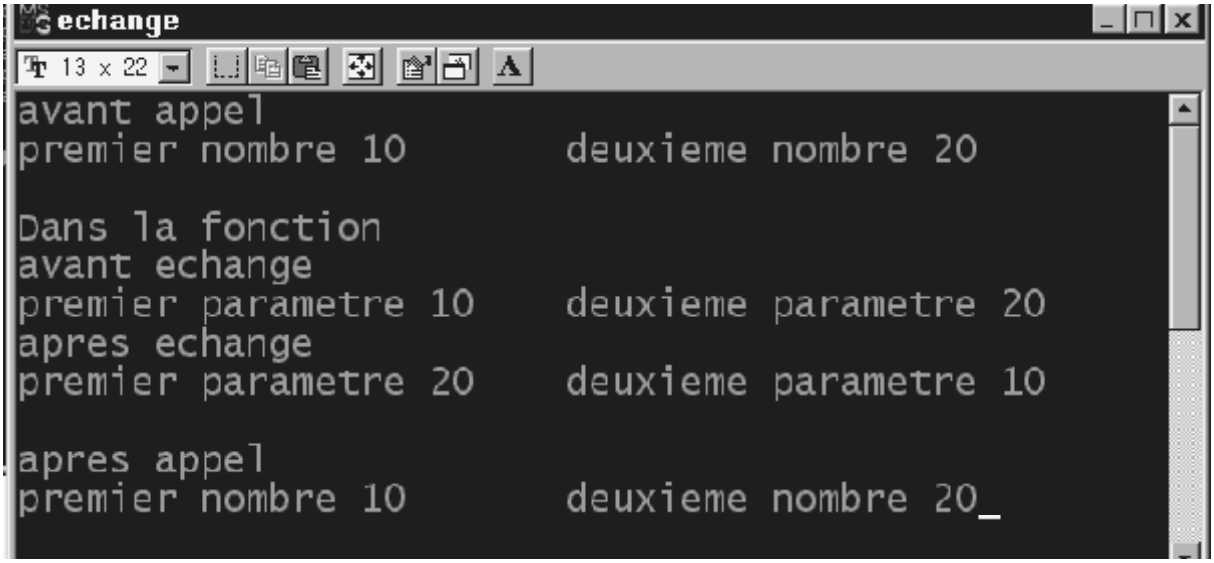
Voyons un exemple qui montre que le passage par valeur ne permet pas de modifier les paramètres effectifs:

```
//cette fonction void échange les valeurs des paramètres (croit-on…)
void echange (int x, int y)
{
   int temp;
   cout << "\n\nDans la fonction";
   cout << "avant echange \npremier parametre " << x << "\tdeuxieme parametre " << y;
   temp = x;
   x = y;
   y = temp;
   cout << "\naprès echange \npremier parametre " << x << "\tdeuxieme parametre " << y;
}

//test de la fonction
main()
{
   int n = 10;
   int p = 20;
   cout << "avant appel \npremier nombre " << n << "\ndeuxieme nombre " << p;
   echange(n, p);
   cout << "\n\naprès appel \npremier nombre " << n << "\ndeuxieme nombre " << p;
}

/*on s'attend à ce que les valeurs de n et de p soient inversées après l'appel de la fonction, mais il n'en est rien, car le passage par valeur ne permet pas d'accéder aux paramètres effectifs*/
```

Sortie d'écran correspondante :



➢ Le passage par référence : &

Pour créer un **paramètre résultat ou donnée-résultat**, il faut utiliser un autre mode de passage de paramètre : le **passage par référence** (appelé passage par variable dans d'autres langages).
Pour déclarer un paramètre formel qui doit être passé par référence, on le précède du signe **&**.
Ainsi, **toute modification du paramètre formel se répercute directement sur le paramètre effectif correspondant**. Tout se passe comme si la fonction agissait directement sur le paramètre effectif.

Dans notre exemple, les deux paramètres formels x et y ont le statut donnée-résultat. Pour que l'échange se répercute sur les paramètres effectifs n et p, il faut donc passer x et y par référence.

```
//cette fonction void échange les valeurs des paramètres (vraiment !)
void echange (int & x, int & y)
{
   int temp;
   cout << "avant echange \npremier parametre " << x << "\ndeuxieme parametre " << y;
   temp = x;
   x = y;
   y = temp;
   cout << "après echange \npremier parametre " << x << "\ndeuxieme parametre " << y;
}
```

/* la fonction principale ne change pas. **L'appel se fait de la même manière, que les paramètres soient transmis par valeur ou par référence**. */

```
main( )
{
   int n = 10;
   int p = 20;
   cout << "avant appel \npremier nombre " << n << "\ndeuxieme nombre " << p;
   echange(n, p);
   cout << "après appel \npremier nombre " << n << "\ndeuxieme nombre " << p;
 }
```

Le cas particulier des tableaux

Le langage C++ permet de transmettre un tout un tableau en paramètre d'une fonction, mais le mécanisme de transmission par défaut est particulier (**les tableaux ne sont jamais passés par valeur**)
Un seul mode de transmission des paramètres existe pour les tableaux : il s'agit du **passage par adresse**. Avec ce mode de transmission, tout comme le passage par référence, tout se passe comme si la fonction appelée travaillait directement avec le tableau mentionné lors de l'appel.
Bien que le passage par adresse et le passage par référence ont des points communs, ils ne fonctionnent pas de la même manière. Un tableau ne peut pas être passé par référence : il ne faut **jamais** le faire précéder de **&**, même si c'est un résultat !

Exemple :

```
void raz(int tab[5]);    //prototype de la fonction void raz

main( )
{
  int i;
  int t1[5] = {2, 6, 3 , 1, 4}
  cout << " tableau avant appel de raz\n";
  for( i=0; i<5; i++)
        cout << t1[i] << "  ";
  raz(t1);   //appel de la fonction avec le tableau en paramètre
  cout << "\ntableau après appel de raz";
  for( i=0; i<5; i++)
        cout << t1[i];
}

void raz(int tab[5])     //définition de la fonction void raz
{
  for(int i = 0; i<5; i++)
        tab[i] =0;
 }
```

sortie d'écran :

```
tableau avant appel de raz
2  6  3  1  4
tableau après appel de raz
0  0  0  0  0
```

- **Cas des paramètres tableau de taille variable**

Il est possible de passer en paramètre un tableau dont la taille est variable (crochets vides). La taille effective (ou le nombre d'éléments utilisés) du tableau passé en paramètre doit alors être passée aussi en paramètre. Cela permet d'utiliser une fonction traitant des tableaux, avec des tableaux de taille différente.

Voilà comment on pourrait adapter la fonction raz définie précédemment :

```
void raz(int tab[ ], int taille) //définition de la fonction void raz
{
  for(int i = 0; i<taille; i++)
        tab[i] =0;
 }
```

l'appel de cette fonction void comporterait alors deux paramètres: le nom du tableau, puis sa taille :

raz (t1, 5);

et on pourrait utiliser cette fonction avec d'autres tableaux de int de taille différente

```
raz(t2, 10);      //avec t2 un tableau d'entiers d'au moins 10 éléments
```

Les variables globales (à utiliser le moins possible !)

Il est possible de définir des variables globales : ce sont des variables communes à toutes les fonctions d'un même fichier source (y compris le programme principal). Toutes les fonctions peuvent utiliser et modifier une variable globale.
Une variable globale doit être déclarée avant le main, juste avant ou après les prototypes des fonctions.

```
int g;   //variable globale
void ajoute_un( );   //prototype

main( )
{
    g =  10;      //initialisation de g
    ajoute_un( );
    cout << g;  //affiche la nouvelle valeur de g qui est 11
}

void ajoute_un( )
{
   g = g + 1;
}
```

✋ Attention ! Une variable globale peut être cachée par une variable locale!
Les variables globales ont une portée qui s'étend à l'ensemble du programme et de ses fonctions. Mais si une variable locale à une fonction porte le même nom qu'une variable globale, cette variable locale cache la variable globale qui ne peut alors plus être utilisée. Cela entraîne souvent des erreurs difficilement décelables. Il ne faut donc jamais déclarer une variable de même nom qu'une variable globale.

Même exemple avec redéclaration de g dans la fonction

```
int g;   //variable globale
void ajoute_un( );

main( )
{
   g =  10;      //initialisation de g
   ajoute_un( );
   cout << g;      //affiche la valeur de la variable globale g qui n'a pas été modifiée par la fonction ajoute_un
}

void ajoute_un( )
{
   int g;          //la variable locale g cache la variable globale de même nom
   g = g + 1;    //la variable globale n'est plus accessible. L'incrémentation se fait en local
}
```