6. Fonctions

Une fonction est une partie du programme séparée du reste. L'utilisation des fonctions a deux grands avantages :

- Elles coupent un programme long dans des parties plus faciles à programmer, ou à identifier et comprendre pour un programme donné.
- Elles permettent d'éviter la duplication inutile des bouts de programme que l'on utilise plusieurs fois.

En fait vous savez déjà utiliser des fonctions (par exemple `sqrt`) et les programmer : `int main()` est une fonction tout à fait normale du point de vue syntaxique, spéciale seulement parce qu'elle est obligatoire et l'exécution du programme y commence. La syntaxe générale d'une fonction est :

```
type nom(arguments) {
  bloc de commandes (y compris return);
}
```

Exemple :

```
int deux_fois_somme_jusqua(int n) {
  int i, res = 0;
  for (i = 1; i <= n; i++) res += i;
  return 2*res;
}
```

Le bloc de commandes (dont les accolades sont toujours obligatoires) doit obligatoirement contenir une commande `return` *expression*`;` où le résultat d'*expression* est du type de la fonction comme indiqué avant son nom. Vu que `return` fait aussi sortir directement de la fonction (comme un `break` dans une boucle), cette commande sera généralement la dernière du bloc.

- Les arguments de la fonction consistent en une liste de variables avec leurs types, séparées par des virgules. Exemples :
  - `double f() // Fonction sans arguments`
  - `double f(double ax, int n1) // Fonction a 2 arguments`
  - `int f(double a, double b, double c) // Fonction a 3 arguments`
    Remarque bien qu'il faut répéter le type (`double a,b,c` ne marche pas).

  Par contre, il n'y a toujours qu'une seule valeur à retourner avec `return`.
- Il existe aussi le type spécial `void` pour une fonction qui ne retourne rien : `void f(...)`. Dans ce cas seulement la fonction ne doit pas contenir une commande `return`.
- Une fonction que vous avez programmée vous-mêmes s'utilise exactement de la même façon que les fonctions des bibliothèques, par exemple :
  `int i, j = 5; i = deux_fois_somme_jusqua(j);`
  ou `cout << deux_fois_somme_jusqua(3) << endl;`
- Les arguments sont passés par valeur dans le C. Cela veut dire que, dans l'exemple ci-dessus, c'est juste la valeur 5 et non pas la variable `j` qui est fournie à la fonction. Il est donc impossible pour la fonction de changer la valeur de la variable `j`.
- Normalement chaque appel d'une fonction se passe de la même façon, dans `deux_fois_somme_jusqua` la variable `res` est initialisée à zéro à chaque appel de la fonction. Par contre, des variables déclarées avec le mot clé `static` (p.ex. : `static int i = 0;`) ne sont initialisées qu'au premier appel, et gardent leur dernière valeur d'un appel au suivant.

Variables locales et globales

Toutes les variables déclarées à l'intérieur d'une fonction (`main` ou autre) sont des **variables locales**. Cela veut dire que ces variables n'existent que dans cette fonction et sont inconnues par les autres fonctions. Si l'on déclare des variables locales du même nom dans deux fonctions différentes, ces variables seront en effet des variables différentes qui n'ont rien à voir entre elles.
Par contre, toutes les variables déclarées à l'extérieur de toute fonction sont des **variables globales**, connues et utilisables par chaque fonction suivant sa définition. À utiliser avec modération.

```
#include<iostream>
using namespace std;
int a = 5;                  // a variable globale, utilisable dans f et main
int f(int x) {              // x et i variables locales dans f
  int i = 1;                // x est initialisee a la valeur de i*y,
  return a*x + i;           //   c'est a dire 20
}
int main() {
  int i = 2; int y;         // y et i variables locales dans main
  a = 2*a; y = a;           // i dans main et i dans f sont independantes
  cout << f(i*y) << endl;   // Affiche 201
  return 0;
}
```

Ce sont seulement des déclarations que l'on peut mettre en dehors des fonctions, toutes les autres commandes normales se trouvent forcément à l'intérieur d'une fonction (`main` ou autre).

Ordre de compilation

Même si l'exécution d'un programme commence par la fonction `main`, la compilation commence simplement au début du fichier. Cela veut dire qu'il faut mettre les variables globales et les fonctions avant le premier endroit où elles sont utilisées dans l'ordre du fichier, sinon on aura une erreur de compilation. Elles sont seulement connues par les fonctions suivant leur définition.

Si l'on veut quand même définir une fonction après sa première utilisation dans l'ordre du fichier, il faut déclarer le prototype de la fonction avant. Le prototype d'une fonction est simplement la première ligne de sa définition avec l'accolade remplacée par un point-virgule. La déclaration d'un prototype peut se faire de façon globale (en dehors de toute fonction, connue par toutes les fonctions qui suivent) ou locale (dans une fonction, connue seulement par cette fonction).

Exemple :

```
void f(double x);  // prototype de la fonction f, declaration globale
int main() {
  ...
  f(x);            // f peut maintenant etre utilisee ici,
  ...              //   meme si sa definition est en bas
}
void f(double x) {  // definition de la fonction f
  ...
}
```

Exemple d'un programme complet utilisant des fonctions pour tracer l'allure (tournée de 90°) d'une fonction mathématique avec des # :

```
#include<iostream>
#include<math.h>
using namespace std;

double f(double x) {            // fonction mathematique a tracer
  return 20.*exp(-x*x/20.);
}

void ligne(double x) {          // fonction qui fait afficher une ligne de #
  int i;
  cout << "|";
  for (i = 1; i <= x; i++)
    cout << "#";
  cout << endl;
}

int main() {
  int k;
  for (k = -8; k <= 8; k++)
    ligne(f(k));
  return 0;
}
```

Comparaison en fonction de dx du calcul d'une dérivée par les deux méthodes $\frac{df}{dx} = (f(x+dx) - f(x))/dx$ et $\frac{df}{dx} = (f(x+dx) - f(x-dx))/(2dx)$:

```
#include<bibli_fonctions.h>
double f(double x) { // fonction dont on va calculer la derivee
  return exp(x); }   // accolade sur la meme ligne par manque de place !
double deriveef1(double x, double dx) {
  return (f(x+dx) - f(x))/dx; }
double deriveef2(double x, double dx) {
  return (f(x+dx) - f(x-dx))/(2*dx); }
int main() {
  double dx, x = 0.; // on calcule la derivee en x=0
  fstream fich;
  fich.open("derivee.res", ios::out);
  for(dx = 1.; dx > 0.005; dx -= 0.01) // et non pas dx>=0.01 car dx reel
    fich << dx << " " << deriveef1(x,dx) << " " << deriveef2(x,dx) << endl;
  fich.close();
  ostringstream pyth;
  pyth << "A = loadtxt('derivee.res')\n"
    << "plot(A[:,0], A[:,1], label='methode 1')\n"
    << "plot(A[:,0], A[:,2], label='methode 2')\n"
    << "xlabel('dx')\n" // pour rajouter des etiquettes sur les axes
    << "ylabel('df/dx(" << x << ")')\n"
    << "legend()\n"     // avec label et legend on peut creer une legende
   ;
  make_plot_py(pyth);
  return 0;
}
```