Les bases de l'Informatique
Les fonctions

Thierry Vaira

BTS SN

v1.0 - 7 juillet 2017

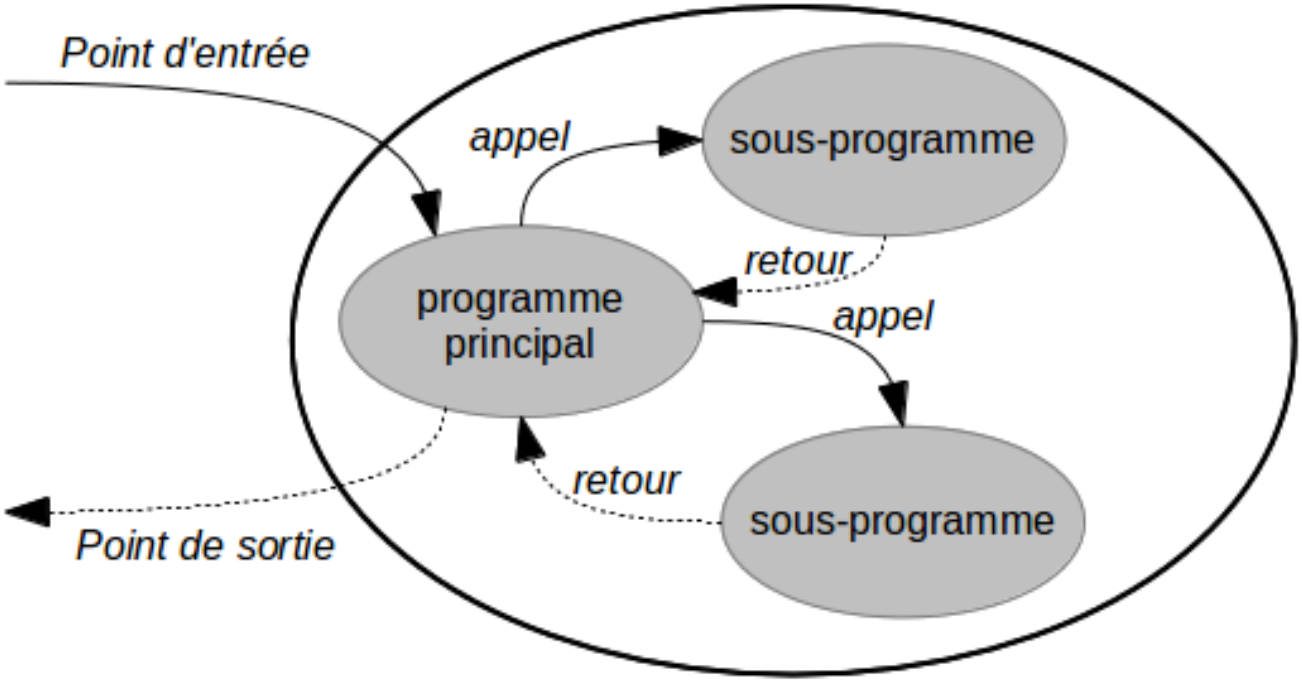
L'approche fonctionnelle

- Décomposer un problème en sous problèmes :
  - Ceci conduit souvent à diminuer la complexité d'un problème et permet de le résoudre plus facilement.
- Éviter de répéter plusieurs fois les mêmes lignes de code :
  - Ceci facilite la résolution de bogues mais aussi le processus de maintenance.
- Généraliser certaines parties de programmes :
  - La décomposition en module permet de constituer des sous-programmes réutilisables dans d'autres contextes.

Structure d'un programme

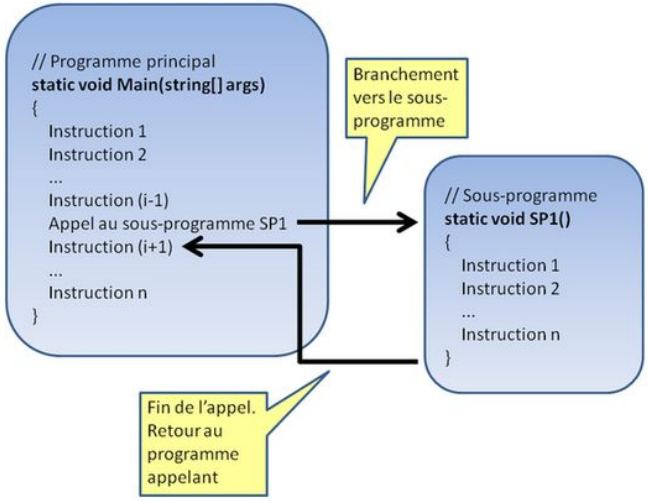
➠ Dans le cas d'une approche fonctionnelle, un programme n'est plus une simple séquence d'instructions mais est constitué :

- D'un ensemble de **sous-programmes** et
- D'un et un **seul programme principal** : unique et obligatoire.



Déroulement d'un programme

- L'exécution du programme commence par l'exécution du programme principal
- L'appel à un sous programme permet de déclencher son exécution, en interrompant le déroulement séquentiel des instructions du programme principal
- Le déroulement des instructions du programme reprend, dès que le sous programme est terminé, à l'instruction qui suit l'appel



Les sous-programmes

➟ On distingue deux types de sous-programmes :

- Les **fonctions**
  - Sous-programme qui retourne **une et une seule valeur** : permet de ne récupérer qu'un résultat.
  - Par convention, ce type de sous-programme ne devrait pas interagir avec l'environnement (écran, utilisateur).
- Les **procédures**
  - Sous-programme qui permet de récupérer de **0 à *n* résultats**
  - Par convention, ce type de sous-programme peut interagir avec l'environnement (écran, utilisateur).
- Cette distinction ne se retrouve pas dans tous les langages de programmation !
  - Par exemple, le C/C++ n'admet que le concept des fonctions qui serviront à la fois pour les fonctions et les procédures.

Définir une fonction

Algorithme :

```
Fonction poserQuestion(q : Chaîne de
     caractères) : Caractère
Donnée(s) : q la question
Résultat : Lit la réponse et retourne
     'V' pour vrai, sinon 'F' pour faux
Variable locale r : Caractère

Début
    Ecrire q
    Répéter
        Ecrire "(V)rai ou (F)aux ?"
        Lire r
    TantQue (r<>'V' ET r<>'F')
    Retourner r
Fin
```

En C++ :

```
char poserQuestion(string q)
{
    char r;

    cout << q;
    do
    {
        cout << "(V)rai ou (F)aux ? ";
        cin >> r;
    }
    while (r!='V' && r!='F');
    return r;
}
```

Appeler une fonction

Algorithme :

```
...
score <- 0
question <- "1. ADN signifie Anti-
     Démangeaison-Nasale ?"
reponse <- poserQuestion(question)
Si reponse = 'F'
    Alors score <- score - 1
    Sinon score <- score + 1
FinSi

question <- "2. La programmation c'est
     facile ?"
...
```

En C++ :

```
...
score = 0;
question = "1. ADN signifie Anti-
     Démangeaison-Nasale ?";
reponse = poserQuestion(question);
if (reponse == 'F')
{
   score = score - 1;
}
else
{
   score = score + 1;
}

question = "2. La programmation c'est
     facile ?";
...
```

Programmation modulaire

- Le découpage d'un programme en sous-programmes est appelée **programmation modulaire**.
- La programmation modulaire se justifie par de multiples raisons :
  - un programme écrit d'un seul tenant devient très difficile à comprendre dès lors qu'il dépasse une page de texte
  - la programmation modulaire permet d'éviter des séquences d'instructions répétitives
  - la programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écris et mis au point une seule fois
  - des fonctions convenablement choisies permettent souvent de dissimuler les détails d'un calcul contenu dans certaines parties du programme qu'il n'est pas indispensable de connaître.
- D'une manière générale, la programmation modulaire clarifie l'ensemble d'un programme et facilite les modifications ultérieures.

Déclaration de fonction

- Le langage C n'offre pour sa part qu'une seule sorte de module : la **fonction**. Si une fonction ne rend aucun résultat et elle peut alors être considérée comme une **procédure**.
- Une fonction est déclarée de la manière suivante :

```
[classe de memorisation] [type] identificateur (parametres);
```

  - La classe de mémorisation et type sont optionnels.
  - Lorsque la classe de mémorisation est absente, la fonction est considérée externe et est donc accessible par tous les modules avec lesquels une édition de liens est faite.
  - Si le type est absent, le résultat de la fonction est par défaut un entier `int`.
  - Lorsqu'il est présent, il définit le type du résultat que produit la fonction.

Définition de fonction

- Une fonction est définie de la manière suivante :

```
[classe de memorisation] [type] identificateur (parametres)
{
    ...
    return resultat;
}
```

  - La liste de paramètres n'est en fait qu'une liste de couples **type/identificateur** séparés par des virgules.
  - Derrière cela, se trouve le bloc `{}` qui définit les traitements effectués dans la fonction.
  - Pour retourner son résultat, la fonction doit utiliser l'instruction `return`.
- Dans tous les cas, une fonction peut être utilisée :
  - comme opérande dans une expression, à partir du moment où il y a concordance de types
  - comme instruction.

Instruction return

- L'instruction `return` provoque le retour chez l'appelant de la fonction courante. Les deux formes d'utilisation sont : `return` ou `return expression`.
- Dans ce premier cas, il s'agit d'une fonction **ne retournant aucun résultat (`void`)**. Elle a été programmée en tant que **procédure**.

```
void foo() {
   instruction;
   return;
}
```

- Dans ce second cas, il s'agit d'une **fonction** qui **retourne la valeur de l'`expression a%2`**. Si nécessaire la valeur retournée est convertie, comme pour une affectation, dans le type du résultat.

```
int estImpair(int a) {
   return (a%2); // retourne 0 (pair) ou 1 (impair)
}
```

Exemple : la fonction multiplier

```c
#include <stdio.h>

/* La fonction de nom multiplier calcule la multiplication de deux
    entiers (a et b) et retourne le résultat sous forme d'entier */
int multiplier(int a, int b) {
   return a * b;
}

int main() {
   int x = 2; int y = 3; int res1, res2;
   // Appel de fonction en lui passant les valeurs de x et y
   res1 = multiplier(x, y); // on récupère la valeur retournée dans res1
   printf("La multiplication de %d par %d donne comme résultat : %d\n",
       x, y , res1);
   res2 = multiplier(x, y) + 2; // on peut utiliser un fonction dans une
        instruction
   printf("Et si on ajoute 2, on obtient : %d\n", res2);
   return 0;
}
```

Exemple : la fonction multiplier

Exemple

```
La multiplication de 2 par 3 donne comme résultat : 6
Et si on ajoute 2, on obtient : 8
```

Remarques : En aucun cas l'imbrication de fonctions n'est possible. Enfin, l'appel de fonctions en C peut se faire de manière récursive (une fonction peut s'appeler elle-même).

- Pour conclure, il faut distinguer :
  - la **déclaration** d'une fonction qui est une instruction fournissant au compilateur un certain nombre d'informations concernant une fonction. Il existe une forme recommandée dite **prototype** :

    `int plus(int, int);` ← fichier en-tête (.h)
  - sa **définition** qui revient à écrire le **corps** de la fonction (qui définit donc les traitements effectués dans le bloc {} de la fonction)

    `int plus(int a, int b) { return a + b; }` ← fichier source (.c ou .cpp)
  - l'**appel** qui est son utilisation. Elle doit correspondre à la déclaration faite au compilateur qui vérifie.

    `int res = plus(2, 2);` ← fichier source (.c ou .cpp)
- *Remarque :* La définition d'une fonction tient lieu de déclaration. Mais elle doit être "connue" avant son utilisation (un appel). Sinon, le compilateur génère un message d'avertissement (warning) qui indiquera qu'il a lui-même fait une déclaration implicite de cette fonction : `"attention : implicit declaration of function 'multiplier'"`

Pointeur vers une fonction

- Le **nom d'une fonction est une constante de type pointeur** :

```
int f(int x, int y)
{
   return x+y;
}

int (*pf)(int, int); // pointeur vers fonction admettant 2 entiers en
    paramètres et retournant un entier

pf = f;           // pointeur vers la fonction f
printf("%d\n", (*pf)(3, 5)); // affiche 8
```

Passage de paramètres

- Lors d'un appel de fonction, les paramètres passés à la fonction doivent correspondre aux paramètres déclarés par leur nombre et par la compatibilité de leurs types.
- En C, il n'existe qu'un seul mode de passage des paramètres : c'est le **passage par valeur**. On dit aussi par **passage par recopie (de valeurs)**. Il faut donc considérer un paramètre comme une variable locale ayant été initialisée avant l'appel de la fonction.

```
int multiplier(int a, int b);

int x = 2; int y = 3; int res;
// Appel de fonction en lui passant les valeurs de x et y
// donc la valeur de x est recopié dans a et la valeur de y est recopié
    dans b
res = multiplier(x, y);
```

Remarque : les noms peuvent être identiques cela ne change rien au mécanisme de recopie !

Problème : tentative de permutation de deux variables

```
#include <stdio.h>
void permuter(int a, int b) {
   int c;
   c = a;
   a = b;
   b = c;
   printf("Dans la fonction après permutation : a = %d et b = %d\n", a,
       b);
}

int main() {
   int a = 2; int b = 3;

   printf("Dans le main : a = %d et b = %d\n", a, b);
   permuter(a, b);
   printf("Après l'appel de la fonction : a = %d et b = %d\n", a, b);

   return 0;
}
```

Mise en évidence du problème de modification de copies

Effectivement, cela ne marche pas car la fonction `permuter` à travailler sur des copies de `a` et `b` :

La permutation ne fonctionne pas :

```
Dans le main : a = 2 et b = 3

Dans la fonction après permutation : a = 3 et b = 2

Après l'appel de la fonction : a = 2 et b = 3
```

Piste : L'exception à cette règle est le cas des **tableaux**. En effet, lorsque le nom d'un tableau constitue l'argument d'une fonction, c'est l'adresse du premier élément qui est transmise. Ses élément ne sont donc pas recopiés

Permutation d'éléments d'un tableau (1/3)

Ceci reste en effet cohérent avec le fait qu'il n'existe pas de variable désignant un tableau comme un tout. Quand on déclare `int t[10]`, `t` ne désigne pas l'ensemble du tableau. `t` est une **constante de type pointeur** vers un `int` dont la valeur est `&t[0]` (adresse du premier élément du tableau).

```
#include <stdio.h>

void permuter(int t[])
{
   int c;

   c = t[0];
   t[0] = t[1];
   t[1] = c;
   printf("Dans la fonction après permutation : t[0] = %d et t[1] = %d\n
        ", t[0], t[1]);
}
```

Permutation d'éléments d'un tableau (2/3)

Remarque : c'est une très bonne chose en fait car dans le cas d'un "gros tableau", on évite ainsi de recopier toutes les cases. Le **passage d'une adresse** (par exemple 32 bits) sera beaucoup plus efficace et rapide.

```
int main()
{
   int t[2] = { 2, 3 };

   printf("Dans le main : t[0] = %d et t[1] = %d\n", t[0], t[1]);
   permuter(t);
   printf("Après l'appel de la fonction : t[0] = %d et t[1] = %d\n", t
        [0], t[1]);

   return 0;
}
```

Permutation d'éléments d'un tableau (3/3)

Effectivement, cela marche car la fonction `permuter` à travailler avec l'adresse du tableau :

Exemple

```
Dans le main : t[0] = 2 et t[1] = 3

Dans la fonction après permutation : t[0] = 3 et t[1] = 2

Après l'appel de la fonction : t[0] = 3 et t[1] = 2
```

En conséquence, pour réaliser l'effet de **passage de paramètre par adresse**, il suffit alors de **passer en paramètre un pointeur vers la variable**.

Passage par adresse (1/2)

- Pour qu'une fonction puisse modifier le contenu d'une variable passée en paramètre, on doit utiliser en C un **pointeur sur cette variable**.
- Exemple de déclaration : `void permuter(int *pa, int *pb); // pa et pb sont des pointeurs sur des int.`
- Le passage se fait toujours par valeur ou par recopie sauf que maintenant, on passe l'**adresse d'une variable**.

```
void permuter(int *pa, int *pb);
int main() {
   int a = 2; int b = 3;
   printf("Dans le main : a = %d et b = %d\n", a, b);
   permuter(&a, &b); // l'adresse (&) de a est recopiée dans le pointeur
        pa, idem pour l'adresse de b dans pb
   printf("Après l'appel de la fonction : a = %d et b = %d\n", a, b);
   return 0;
}
```

Passage par adresse (2/2)

```
void permuter(int *pa, int *pb) {
   int c;
   c = *pa;
   *pa = *pb;
   *pb = c;
   printf("Dans la fonction après permutation : a = %d et b = %d\n", *pa
        , *pb);
}
```

La permutation fonctionne maintenant :

```
Dans le main : a = 2 et b = 3
Dans la fonction après permutation : a = 3 et b = 2
Après l'appel de la fonction : a = 3 et b = 2
```

Protection contre le passage par adresse (1/4)

Remarque n°1 : lorsqu'on passe une adresse par pointeur, il y a un risque que la fonction modifie cette adresse. Dans ce cas, on peut se protéger en déclarant l'**adresse passée comme constante**.

```
#include <stdio.h>
void foo(int * const pa) {
   int un_autre_a;
   pa = &un_autre_a; // tentative de modification d'adresse contenue
        dans le pointeur
}
int main() { int a = 2; foo(&a); return 0; }
```

On obtient un contrôle d'accès à la compilation :

```
In function 'foo':
erreur: assignment of read-only location 'pa'
```

Protection contre le passage par adresse (2/4)

Remarque n°2 : On a vu que lorsque l'on doit passer en paramètre des "grosses" tailles de variables, il serait plus efficace et plus rapide à l'exécution de passer une adresse. Mais il y aurait un risque que la fonction modifie cette variable. Dans le cas d'un accès limité en lecture, on peut se protéger en déclarant le **pointeur comme constant**.

```
#include <stdio.h>
void foo(const long double *pa) {
   printf("Je suis un pointeur de taille plus légère %d octets\n",
        sizeof(pa));
   printf("Je contiens l'adresse de a : pa = %p\n", pa);
   printf("et j'ai un accès en lecture : a = %.1Lf\n", *pa);
   /* et non, tu ne peux mas modifier le contenu de la variable pointée
        ! */
   printf("mais pas en écriture !\n");
   *pa = 3.5; // ligne 10 à enlever !
}
```

Protection contre le passage par adresse (3/4)

On obtient un contrôle d'accès à la compilation :

```
In function 'foo':
ligne 10: erreur: assignment of read-only location '*pa'
```

Protection contre le passage par adresse (4/4)

```
int main() { long double a = 2.0;
  printf("Je suis une grosse variable de taille %d octets\n",sizeof(a));
  printf("Je suis a et j'ai pour valeur : a = %.1Lf\n", a);
  printf("et pour adresse : &a = %p\n\n", &a);
  foo(&a);
  return 0;
}
```

Une utilisation de pointeur constant :

```
Je suis une grosse variable de taille 12 octets
Je suis a et j'ai pour valeur : a = 2.0
et pour adresse : &a = 0xbf8d8240
Je suis un pointeur de taille plus légère 4 octets
Je contiens l'adresse de a : pa = 0xbf8d8240
et j'ai un accès en lecture : a = 2.0
mais pas en écriture !
```

Passage par référence (1/2)

- En C++, on a aussi la possibilité d'utiliser le **passage par référence**.
- Intérêt : lorsqu'on passe des variables (ou des objets) en paramètre de fonctions ou méthodes et que le coût d'une recopie par valeur est trop important ("gros" objet), on choisira un **passage par référence**.

```
void permuter(int &a, int &b) {
 int c;
 c = a; a = b; b = c;
 printf("Dans la fonction après permutation : a = %d et b = %d\n", a,b);
}

int main() {
 int a = 2; int b = 3;
 printf("Dans le main : a = %d et b = %d\n", a, b);
 permuter(a, b);
 printf("Après l'appel de la fonction : a = %d et b = %d\n", a, b);
 return 0;
}
```

Passage par référence (2/2)

La permutation fonctionne également en utilisant un passage par référence :

```
Dans le main : a = 2 et b = 3
Dans la fonction après permutation : a = 3 et b = 2
Après l'appel de la fonction : a = 3 et b = 2
```

Protection contre le passage par référence

Si le paramètre ne doit pas être modifié, on utilisera alors un passage par référence sur une variable constante :

```
void foo(const long double &a) {
   printf("J'ai un accès en lecture : a = %.1Lf\n", a);
   printf("mais pas en écriture !\n");
   //a = 3.5; // interdit car const ! (voir message)
}
int main() { long double a = 2.0;
   printf("Je suis a et j'ai pour valeur : a = %.1Lf\n", a);
   foo(a);
   return 0;
}
```

Le compilateur nous préviendra de toute tentative de modification :

```
In function 'void foo(const long double&)':
erreur: assignment of read-only reference 'a'
```

Bilan

- En résumé, voici les différentes déclarations en fonction du contrôle d'accès désiré sur un paramètre reçu :
  - passage par valeur → accès en lecture seule à la variable passée en paramètre : `void foo(int a);`
  - passage par adresse → accès en lecture seule à la variable passée en paramètre : `void foo(const int *a);`
  - passage par adresse → accès en lecture et en écriture à la variable passée en paramètre : `void foo(int *a);`
  - passage par adresse → accès en lecture et en écriture à la variable passée en paramètre (sans modification de son adresse) : `void foo(int * const a);`
  - passage par adresse → accès en lecture seule à la variable passée en paramètre (sans modification de son adresse) : `void foo(const int * const a);`
  - passage par référence → accès en lecture et en écriture à la variable passée en paramètre : `void foo(int &a);`
  - passage par référence → accès en lecture seule à la variable passée en paramètre : `void foo(const int &a);`

Surcharge

- Il est généralement conseillé d'attribuer des noms distincts à des fonctions différentes.
- Cependant, lorsque des fonctions effectuent la même tache sur des objets de type différent, il peut être pratique de leur attribuer des noms identiques.
- Ce n'est pas possible de réaliser cela en C mais seulement en C++.
- L'utilisation d'un même nom pour des fonctions (ou méthodes) s'appliquant à des types différents est nommée **surcharge**.
- *Remarque :* cette technique est déjà utilisée dans le langage C++ pour les opérateurs de base. L'addition, par exemple, ne possède qu'une seul nom (+) alors qu'il est possible de l'appliquer à des valeurs entières, virgule flottante, etc ...

La surcharge de fonctions en action (1/2)

```
#include <iostream>

using namespace std;

// Un seul nom au lieu de print_int, print_float, ...
void print(int);
void print(float);

int main (int argc, char **argv)
{
   int n = 2;
   float x = 2.5;

   print(n);
   print(x);

   return 0;
}
```

La surcharge de fonctions en action (1/2)

```
void print(int a)
{
   cout << "je suis print et j'affiche un int : " << a << endl;
}

void print(float a)
{
   cout << "je suis print et j'affiche un float : " << a << endl;
}
```

On obtient :

```
je suis print et j'affiche un int : 2
je suis print et j'affiche un float : 2.5
```

Signature et Prototype

- *Remarque importante :* la surcharge ne fonctionne que pour des **signatures différentes** et le type de retour d'une fonction ne fait pas partie de la signature.
- On distingue :
  - La **signature d'une fonction est le nombre et le type de chacun de ses arguments**.
  - Le **prototype d'une fonction est le nombre et le type de ses arguments (signature) et aussi de sa valeur de retour**.
- Aller plus loin : il est aussi possible de **surcharger les opérateurs de base** avec des signatures différentes pour ses propres classes.