Fonctions

Les Fonctions

Rôle d'une fonction en algorithmique

Le rôle d'une fonction en algorithmique est de regrouper des instructions ou traitements qui doivent être faits de manière répétitive au sein d'un programme. Par exemple, dans un programme traitant des tableaux, on voudrait afficher plusieurs fois des tableaux dont les variables stockent des valeurs différentes. Cependant, même si les valeurs sont différentes, l'affichage d'un tableau se résume toujours à un parcours de toutes les variables utilisées avec une boucle `pour` (de l'indice 0 jusqu'à l'indice taille_utile-1), avec un affichage de chaque valeur grâce à l'instruction `afficher()`.

Il serait utile de regrouper ces instructions d'affichage, pour n'en avoir qu'un seul exemplaire, et de pouvoir s'en servir lorsqu'on en a besoin, sans avoir à saisir les lignes dans le programme.

C'est à cela que sert une fonction : elle regroupe des instructions auxquelles on peut faire appel. Il s'agit en fait d'un petit programme qui sera utilisé par le programme : on parle aussi de sous-programme pour une fonction.

Les entrées, les sorties

Une fonction, en plus des instructions qu'elle comporte, reçoit des valeurs en entrée, ce sont des valeurs sur lesquelles elle va effectuer le traitement), et est susceptible de produire une valeur en sorite, valeur que l'on pourra récupérer par la suite.

Nous allons prendre l'analogie avec une fonction mathématique en ce qui concerne les entrées et sorties. Ecrivons une fonction de la même manière qu'avec le formalisme des mathématiques :

$F : \mathbb{R} \times \mathbb{R} \longmapsto \mathbb{R}$

$x, y \longrightarrow x^2+xy+y^2$

Les entrées sont les données que l'on fournit à la fonction pour qu'elle puisse les traiter et fournir un résultat : ici il s'agit des valeurs x et y.

La sortie est le résultat que donne la fonction, ici il s'agit de la valeur calculée x^2+xy+y^2.

types des entrées et sorties

On peut aussi remarquer que, d'après la définition (mathématique) de la fonction, on peut indiquer quel est le type des entrées et des sorties. Il s'agit, dans cet exemple, de valeurs réelles.

On peut résumer ceci en disant que la fonction proposée en exemple a comme entrée deux valeurs de type réel, effectue un calcul, et a comme sortie une valeur de type réel.

Exemples de fonctions avec leurs entrées/sorties

Prenons d'autres exemple d'autres fonctions, en insistant au fur et à mesure sur le côté informatique :

Fonction calculant une factorielle , $n!=\prod_{i=1}^{i=n} i$

Entrée : un nombre entier

Sortie : un nombre entier

Fonction calculant une courbe en cloche : $f(n,x)=(nx).e^{-nx}$

Entrée : une nombre entier, un nombre réel

Sortie : un nombre réel

Fonction faisant l'affichage d'un entier

Entrée : un nombre entier

Sortie : rien

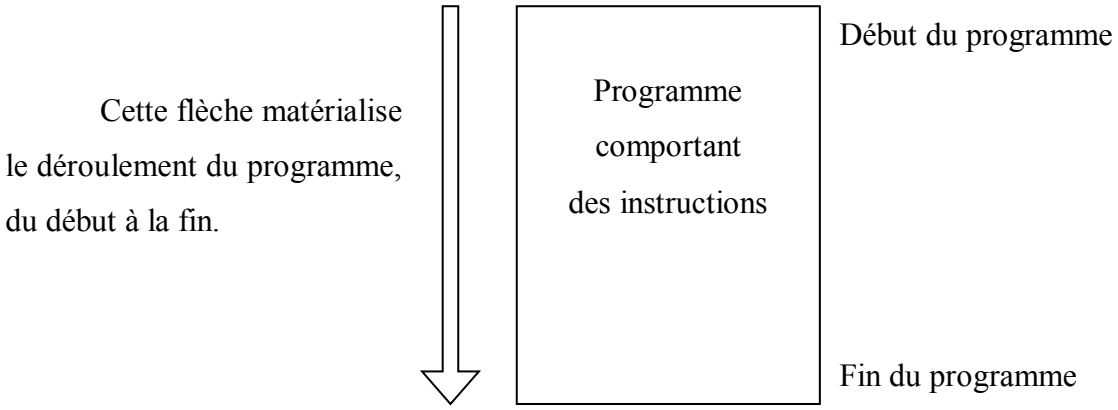
Fonction faisant la saisie d'une valeur de type caractère

Entrée : rien

Sortie : un caractère

Programme, fonctions, séquences

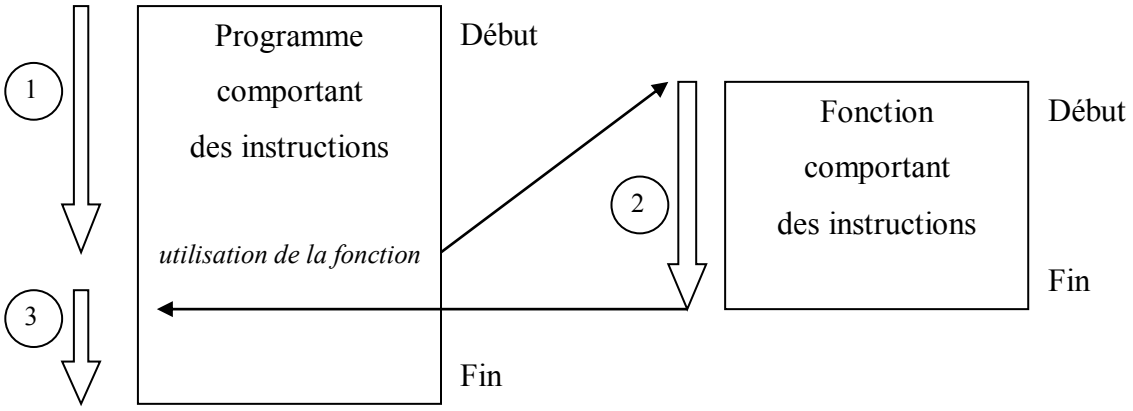
Avant de nous intéresser plus précisément à la manière dont on définit les fonctions en algorithmique, nous allons voir comment on les utilise au sein d'un programme. Pour ce faire, nous allons employer une nouvelle représentation par bloc, ou chaque bloc consistera en un ensemble d'instructions. Un programme simple (sans fonctions), sera représenté par un bloc unique qui est exécuté du début à la fin :



Le fait que la flèche décrivant le déroulement du programme soit droite ne veut pas dire que le programme est purement séquentiel, il peut très bien y avoir, parmi les instructions de ce programme, des boucles, des tests, des sélections. Cette flèche indique simplement que les instructions sont effectuées dans un certain ordre.

Lorsque l'on décide d'utiliser une fonction, que l'on peut appeler sous-programme, on dispose en fait d'un deuxième bloc contenant des instructions.

Le programme se déroule normalement, et la fonction n'intervient pas. Si l'on veut que la fonction soit utilisée, il faut le préciser dans le programme en employant une instruction d'utilisation de la fonction. On a alors le déroulement suivant :



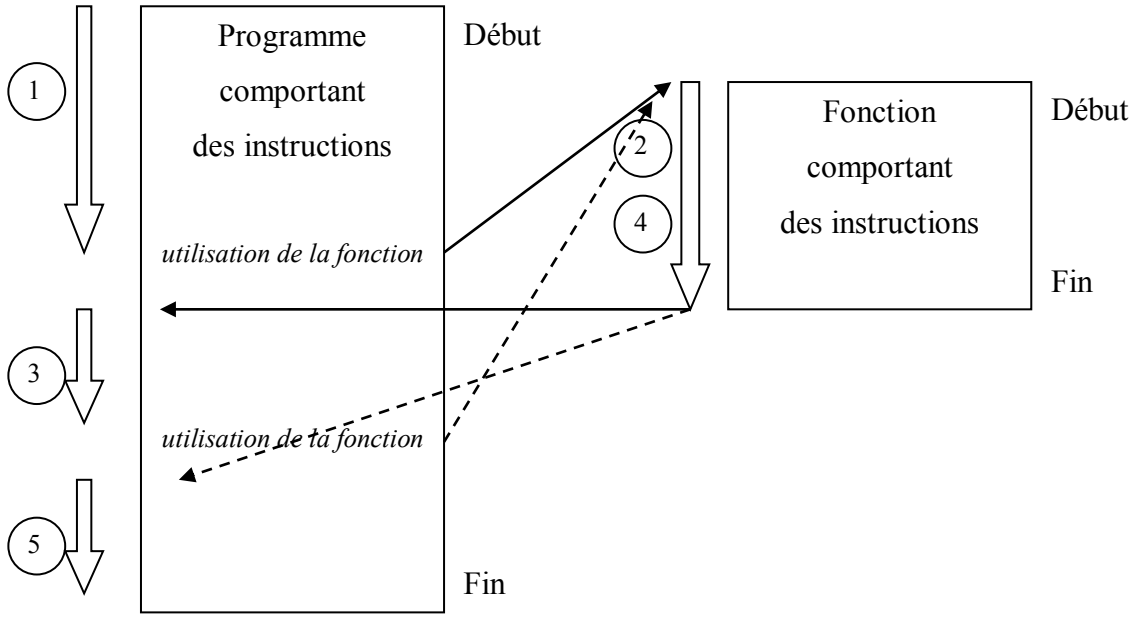
Le programme commence à être exécuté, jusqu'à l'instruction d'utilisation de la fonction (étape 1), et son déroulement s'interrompt. La fonction est exécutée du début à la fin, (étape 2) puis le programme reprend là où il s'était arrêté (étape 3) et est exécuté jusqu'à la fin.

Pourquoi donc ne pas inclure directement les instruction de la fonction dans le programme puisque le déroulement ne semble pas changer ?

Il y a deux raisons fondamentales à l'utilisation de fonctions, pour répondre à cette question :

Répartition des instructions dans plusieurs bouts de programme : il vaut mieux, pour obtenir un programme lisible et facile à maintenir, utiliser plusieurs blocs contenant peu d'instructions qu'un énorme bloc contenant toutes les instructions.

Réutilisation : une fonction peut être utilisée plusieurs fois dans le même programme :



Pour obtenir le déroulement équivalent dans un seul programme, il faudrait y recopier deux fois les instructions de la fonction : ce serait une perte de temps et de place. Le fait qu'il n'y ait besoin que d'un seul exemplaire de la fonction offre donc un gain de place et de temps (on ne saisit qu'une seule fois la fonction et on peut s'en servir plusieurs fois).

Entête, définition, appel, nom de fonction

En informatique, une fonction, que l'on veut utiliser dans un programme, se définit en plusieurs étapes.

La fonction doit être définie, c'est à dire que l'on doit indiquer quelles sont les instructions qui la composent : il s'agit de la définition de la fonction, où l'on associe les instructions à l'identification de la fonction..

On doit donc trouver une définition de la fonction, qui comporte :

Une identification ou **entête de la fonction**, suivie des instructions de la fonction, ou **corps de la fonction**.

La fonction doit être définie et comporter : une entête, pour l'identifier et un corps contenant ses instructions.

Comment écrire une définition de fonction

Entête

L'entête d'une fonction contient les informations nécessaires à identifier la fonction : son nom, ses entrées et leurs types, sa sortie

Syntaxe de l'entête :

```
fonction nom(entrée : liste entrées avec leurs types → sortie : type de la sortie)
```

la liste des entrées avec leur type suit exactement la même syntaxe que les définitions de variables que nous avons traitées dans le cours concernant la syntaxe de la définition de variables en langage algorithmique, et ceci n'est pas un simple hasard, comme nous nous en rendrons compte par la suite.

La seule différence réside dans le fait que si plusieurs entrées ont le même type, on ne peut pas les écrire comme on le ferait pour une définition multiple de plusieurs variables du même type. Il faut rappeler le type de l'entrée pour chacune des entrées.

La sortie ne nécessite pas d'être nommée, car c'est à la fonction ou au programme appelant la fonction de récupérer cette valeur. Le rôle de la fonction se limite uniquement, dans le cas de la sortie, à pouvoir fournir une valeur numérique.

exemples d'entête pour les fonctions déjà évoquées dans la première partie du polycopié :

```
fonction calc(entree : reel x, reel y → sortie : reel) // la fonction calculant x2+xy+y2, a deux entrées X et Y et une sortie, entrées et sortie sont de type reel
```

Fonction calculant une factorielle :

```
fonction facto(entree : reel n → sortie : entier)
```

Fonction calculant une courbe en cloche : $f(n,x)=(nx).e^{-nx}$

```
fonction cloche(entree : reel x, entier n → sortie : reel)
```

Fonction faisant l'affichage d'un entier

```
fonction aff_ent(entree : entier t) // pas de sortie
```

Fonction faisant la saisie d'une valeur de type caractère

```
fonction saisie_char(→ sortie : caractere) // pas d'entree
```

Fonction sans entree, sans sortie

```
fonction mystere()
```

Corps

Le corps de la fonction est constitué des instructions de la fonction, et est placé directement à la suite de l'entête de la fonction, entre accolades {}.

Dans le corps de la fonction, outre les instructions, on peut également trouver des définitions de variables qui peuvent être utiles pour faire des calculs intermédiaires lorsque l'on utilise la fonction.

Les variables locales

Ces variables, définies à l'intérieur de la fonction, sont des variables dites locales, car elles ne sont connues que par la fonction. Cela est logique, car c'est la fonction qui les définit (en quelque sorte) pour son usage propre. Une autre fonction, ou un programme principal, ne connaissent pas l'existence de ces variables, donc à plus forte raison, ne connaissent ni leur nom, ni leur type, ni leur valeur, ni leur adresse. Elles ne peuvent les utiliser.

Les entrées de la fonction, précisées dans l'entête, sont également considérées comme des variables locales de la fonction : c'est pour cela que la syntaxe d'écriture des entrées est si proche de la syntaxe d'une définition de variable.

Enfin, les définitions des variables locales à la fonction suivent exactement les mêmes règles que celles que nous avons déjà rencontrées pour les définitions de variables d'un programme, et, encore une fois, cela ne tient pas au hasard, car…

Le programme principal est une fonction

Aussi surprenant que cela puisse paraître, c'est le cas. En langage C, mais aussi pour les autres langages, toute instruction doit faire partie de la définition d'une fonction.

ⓘ *Le paragraphe qui suit est essentiellement technique et n'est pas indispensable à la compréhension du cours, vous pouvez ne pas le lire si vous n'êtes pas encore très à l'aise avec les notions abordées dans ce cours et y revenir par la suite .*

On peut en effet considérer qu'un programme est toujours en train de fonctionner sur l'ordinateur : le système d'exploitation. En ce sens, un programme que vous souhaitez exécuter va être compris, par le programme système, par un appel à une fonction qui doit réaliser les instructions de votre programme. Cependant, si votre programme comporte lui-même plusieurs fonctions, le système ne va pas pouvoir choisir par laquelle commencer. Pour

indiquer au système la fonction par laquelle il doit débuter, on a choisi de lui associer un nom particulier qui sera reconnu comme étant le début d'exécution du programme. En langage algorithmique, ce nom est `principale()` (de `fonction principale()`) et en langage C, ce nom est `main()`.

Syntaxe d'un programme

Au lieu d'écrire un programme sous la forme :

```
programme nom_du_prog
définitions des variables
instructions
```

on peut maintenant écrire :

```
fonction principale()
{
définition des variables
instructions
}
```

ces deux écritures sont strictement équivalentes, et d'ailleurs, certains d'entre vous auront déjà remarqué qu'en langage C, c'est la deuxième forme qui est utilisée : main() est une fonction du langage C !

l'instruction de retour

Pour traiter intégralement les fonctions, nous avons besoin d'une nouvelle instruction permettant que la fonction communique sa sortie à la fonction qui l'utilise. La sortie d'une fonction (il y en a au maximum une) est en fait une valeur numérique que celle ci fournit, et la seule information nécessaire à la bonne interprétation de cette valeur numérique est sont type : c'est pourquoi on ne précise que le type de cette sortie, il est inutile de lui associer un nom. Il nous faudrait donc une instruction dont l'effet serait : "répondre à la fonction appelante la valeur qu'elle demande". Cette instruction existe, heureusement, et en langage algorithmique, son nom est `retourner`.

Syntaxe : `retourner expression;`

Effet : transmet la valeur de l'expression et met fin à l'exécution de la fonction.

Toute instruction placée après l'instruction `retourner` est purement et simplement ignorée. Donc cette instruction s'emploie forcément comme dernière instruction d'une fonction.

La seule contrainte à respecter est que le type de l'expression soit le même que celui de la sortie, qui est indiqué dans l'entête de la fonction.

Cas des fonctions sans sortie :

Lorsqu'une fonction ne possède pas de sortie, il n'est pas indispensable d'utiliser l'instruction `retourner`, à part pour indiquer que la fonction se termine, ce qui peut aussi être repéré par l'accolade fermante de la fonction.

Dans ce cas précis, on utilisera l'instruction `retourner;` seule sans lui associer d'expression, puisqu' aucune valeur ne sera demandée à cette fonction.

Donc, que la fonction ait une sortie ou non**, la dernière instruction d'une fonction doit systématiquement** être `retourner`.

exemples

quelques exemples de corps de fonction :

```
fonction facto(entree : n : entier → sortie : entier)
{
      entier compt, resultat;    // variable locales

      resultat ← 1;

      si (n > 0) alors// n est aussi une variable locale
      {
            pour compt de 1 à n faire  // boucle pour calcul
            {
                  resultat ← resultat * compt;
            }
      }

      retourner resultat;
}
```

```
fonction aff_ent(entree : entier toto)
{
      afficher(toto);
}
```

```
fonction saisie_char(→ sortie : caractere)
{
      caractere val_saisie;

      afficher("saisissez un caractere :");
      saisir(val_saisie);
```

```
      retourner val_saisie;
}
```

exemples avec des fonctions retournant des expressions

On peut très bien imaginer une fonction un peu stupide (programmée par vous savez qui, par exemple…), dont le rôle est de répondre la valeur 1 lorsqu'on l'appelle. Cela est tout à fait possible en informatique.

```
fonction repond_1(→ sortie : entier)
{
      retourner 1;
}
```

on peut donc remarquer que ce qui suit l'instruction retourner est bien une expression et non obligatoirement un nom de variable.

Deuxième exemple : une fonction qui réalise l'addition de deux entiers. Evidemment, cet exemple n'a qu'un rôle illustratif, car l'opérateur + est bien entendu beaucoup plus efficace dans ce cas précis.

```
fonction addition(entree : entier t_1, entier t_2 → sortie : entier)
{
      // pas besoin de variables intermediaires, le calcul se fera
      // dans l'expression qui sera renvoyée par la fonction

      retourner t_1+t_2;
}
```

Dans cet exemple, l'ordinateur, lorsqu'il rencontre l'instruction `retourner t_1+t_2;` commence par calculer la valeur `t_1+t_2`, puis retourne cette valeur.

Troisième exemple : allons dans l'excès inverse: la fonction doit faire un calcul complexe, par exemple le calcul d'une fonction gaussienne $G(x,t)=\frac{1}{\sqrt{2\pi t}}e^{\frac{-x^2}{2t}}$

```
fonction gauss(entree : reel x, reel sig → sortie : reel)
{
      retourner(1./racine(2.*3.1415*sig))*expo(-1.*x*x/(2.0*sig));
}
```

Il y avait peut être, dans ce cas, intérêt à utiliser des variables locales pour mieux comprendre le calcul. On remarque aussi que cette fonction fait appel à deux autres fonctions : racine() et expo().

quelques exemples de programme avec fonctions illustrant le principe des variables locales.

Note : ces fonctions ne font rien d'intéressant par elles-mêmes, ce sont les variables qu'elles utilisent qui sont importantes.

```
fonction A(entree : entier a1 → sortie : reel)
{
      reel t;

      t ← a1*1.0707;

      retourner n*t; // on récupère n de la fonction B et on
                     // multiplie par t qu'on a calculé

// ERREUR : la variable n est une variable locale à la fonction
// B et est donc totalement inconnue de la fonction A
}

fonction B(entree : reel x → sortie : reel)
{
      entier n;

      n ← x*a1; // calcul de x * a1 et troncature quand on affecte à
                // n qui est entier

// ERREUR : la variable a1 est une variable locale à la fonction A
// et ne peut donc être utilisée dans la fonction B

      retourner (reel)n; // on renvoie n, transtypé en réel
}
```

Dans ce cas, le compilateur indiquera que les variable référencées n'existent pas. La manière la plus simple pour éviter des confusions est de parler de ces variables en leur adjoignant le nom de la fonction dans laquelle elles sont définies. Dans cet exemple, la variable `n` définie dans la fonction `B` peut être considérée comme : "la variable `n` de la fonction `B`", c'est en tout cas de cette manière que l'ordinateur la considère. Si vous prenez cette simple habitude, vous aurez moins de difficultés à comprendre les mécanismes de communication que nous allons aborder par la suite.

Il existe également une autre méthode pour se souvenir qu'une variable est locale et donc inconnue aux autres fonctions. La définition de la variable `t` dans la fonction `A` se fait après l'accolade ouvrante : on peut donc dire que cette variable est créée au début de la fonction `A`. A la fin de la fonction `A`, pour une meilleure gestion de la mémoire, cette fonction 'fait le ménage' : elle est priée de laisser la mémoire dans le même état que celui dans lequel elle l'a obtenue à son début. A la fin de cette fonction, donc, la variable `t` est détruite

(rassurez-vous, elle ne souffre pas). C'est ce que l'on appelle **la durée de vie d'une variable : une variable n'existe que le temps nécessaire à son utilisation**.

Nous aurons de toute façon l'occasion de revenir sur ces points un peu plus avant dans le cours.

Placement des fonctions dans un programme

Dans un programme, les définitions des fonctions se placent avant le programme principal, de manière à ce que le programme connaisse les entêtes de fonction, qui sont nécessaires à leur bon usage. En effet, on doit définit une fonction avant de l'appeler. Par contre, il n'y a aucune obligation d'appeler une fonction définie si elle n'apparaît finalement pas utile.

Un programme avec fonctions aura donc l'allure globale suivante :

```
définition de la fonction n°1

définition de la fonction n°2
…
définition de la fonction n° N

programme principal
     appels éventuels aux fonctions précédemment définies.
```

Appel des fonctions

L'appel d'une fonction correspond à une demande de son utilisation, ceci est fait dans une autre fonction, dont par exemple le programme principal. Afin d'appeler une fonction, on doit préciser : son nom, ainsi que les valeurs que l'on fournit pour les entrées. On ne précise pas la valeur de la sortie, car c'est la fonction appelée qui est en charge de la fournir !

La fonction (ou programme principal) qui appelle (ou utilise) une fonction est dite : fonction **appelante**; la fonction qui est utilisée est dite fonction **appelée**.

Soit une fonction nommé fonc, et possédant une liste d'entrées : type1 entrée1, type2 entrée2,…,type*n* entrée *n*. Son entête est donc :

```
fonction fonc(entree : type1 entree1, type2 entree2,…, typen entreen
→ sortie : type_sortie)
```

Pour appeler cette fonction, la syntaxe est la suivante :

`fonc(expr1, expr2,…,exprn);` où `fonc` est le nom de la fonction, et `expri` , où *i* est compris entre 1 et *n*, est une expression dont le type est celui de l'entrée *i*.

Gestion des entrées : les arguments

Lors de l'appel de la fonction, une expression donnant une valeur à une entrée de la fonction est appelée **argument de la fonction**.

Exemple : soit une fonction de calcul et le programme l'utilisant (l'appelant).

```
fonction calcul(entree : reel x, reel y → sortie : reel)
{
      reel resultat;

      resultat ← x*x*x + 3.0*x*x*y + 3.0*x*y*y + y*y*y;
      // calcul de x³ + 3.x²y + 3.xy² + y³

      retourner resultat;
}

programme calcul_poly
reel x_1, x_2;

afficher("entrez deux valeurs reelles :");
saisir(x_1);
saisir(x_2);

// appels possibles de la fonction de calcul

calcul(1.0, -7.32);

calcul(x_1,x_2);

calcul(x_1/2.0, x_1);

calcul(x_1+x_2, x_1-x_2);
```

Tous ces appels sont corrects, car les arguments sont bien des expression dont le type est : `reel`. Retenez bien qu'un argument n'est pas forcément une variable, mais peut être une expression.

Dans tous ces cas, l'ordinateur réalise les deux étapes suivantes :

a) Calcul de la valeur de l'expression pour chacun des arguments

b) Transmission de cette valeur à l'entrée de la fonction correspondante pour que la fonction s'exécute avec les valeurs transmises.

Gestion de la valeur de sortie de la fonction : affectation

Lorsque l'on appelle de cette manière une fonction, on obtient la valeur de la sortie retournée par la fonction appelée ainsi :

L'écriture `nom_de_fonction(liste_des_arguments);` est en réalité une expression dont le type est celui de la sortie de la fonction, qui est précisé dans l'entête de cette dernière. Il s'agit donc d'une valeur numérique que l'on doit affecter à une variable si on veut la conserver. Il s'agit ici du même phénomène que celui que l'on rencontre avec la commande `reservation()` (on devrait dire, maintenant, la fonction `reservation()`, car c'est bien une fonction !) : le résultat de `reservation()` doit être stockée dans un pointeur.

Lorsque vous voulez conserver le résultat retourné par une fonction appelée, il faut affecter ce résultat (obtenu par l'appel) dans une variable du même type que celui de la sortie de la fonction appelée.

Exemple : soit une fonction calculant la moyenne de deux entiers, on désire appeler cette fonction et afficher le résultat obtenu.

```
fonction moy(entree : entier a, entier b → sortie : reel)
{
      reel moyenne;

      moyenne ← (reel)(a+b)/2.0;

      retourner moyenne;
}

programme affiche_moy

entier ent1, ent2, resultat;

afficher("saisissez deux entiers :");
saisir(ent1);
saisir(ent2);

resultat ← moy(ent1,ent2);

afficher("la moyenne vaut :",resultat);
```

On aurait pu remplacer ces deux dernières lignes par la ligne suivante :

```
afficher("la moyenne vaut :",moy(ent1,ent2));
```

car `moy(ent1,ent2);` est une expression de type `reel` qui vaut la sortie de la fonction appelée.

Petit rappel sur les variables locales, avant de passer au point suivant qui est fondamental :

Les variables `a`, `b` et `moyenne` sont des variables connues uniquement de la fonction `moy` : le programme ne les connaît pas.

Les variables `ent1`, `ent2` et `resultat` sont des variables du programme, la fonction `moy` ne les connaît pas.

Le passage des paramètres

Nous allons étudier, dans cette partie, le mécanisme de passage des paramètres, qui est un point fondamental concernant les fonctions. Nous allons voir en détail la manière dont se fait la communication entre les arguments fournis par la fonction appelante et les paramètres (ou entrées) reçues par la fonction appelée.

Pour cela, il est commode d'utiliser une schématisation par blocs à laquelle on adjoint les variables locales à chacune des fonctions.

Exemples : soient la fonction et le programme principal suivants :

```
fonction toto(entree : reel tata, reel tutu → sortie : reel)
{
      reel titi;

      titi ← (tata/tutu) - (tutu/tata);

      retourner titi;
}

fonction principale()
{
      reel x,y,z,t;

      x← 3.54;
      y← -12584.007;
      z← toto(y,x);
      t← toto(z*x,y-8.18);

      afficher("z = ",z,"\n");
      afficher("t = ",t,"\n");

      afficher("un autre calcul :",toto(z,t));
}
```

affichera :

```
z = -3554.803956
t = -0.001300
un autre calcul : 2734887.745877
```

Avant de passer au schéma par bloc, intéressons-nous au mécanisme de communication qui est utilisé par l'ordinateur pour transmettre les valeurs des arguments aux paramètres.

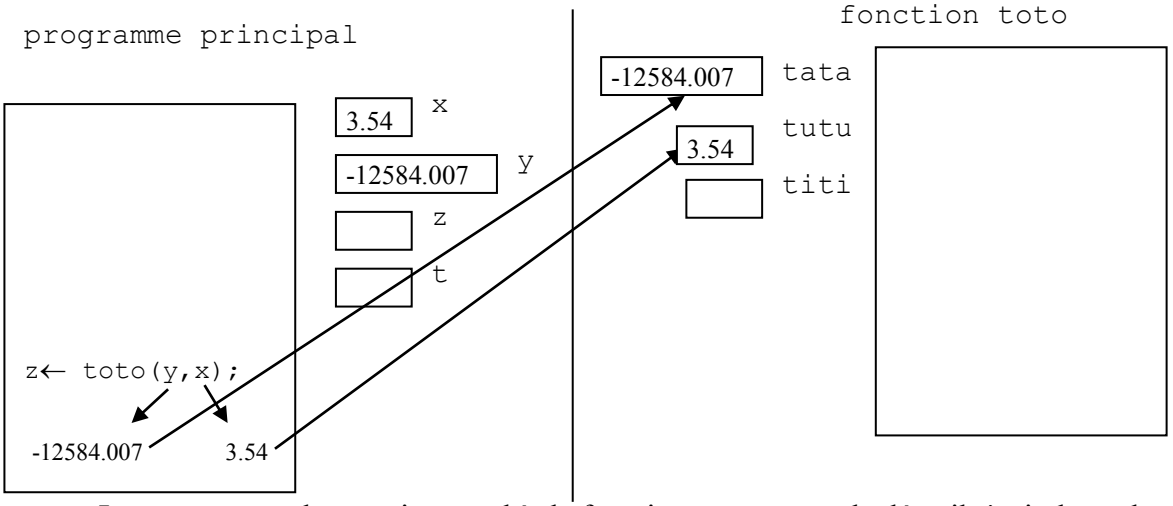
Le mécanisme de recopie

Les actions effectuées par l'ordinateur lors du passage des paramètres (transmission de la valeur des arguments) sont les suivantes :

1) les valeurs des arguments sont **calculées** (ou évaluées)
2) ces valeurs sont **recopiées** dans les paramètres correspondants de la fonction : l'ordre de recopie est celui dans lequel les entrées ou paramètres de la fonction sont écrits dans l'entête de la fonction : l'argument 1 dans le paramètre 1, et ainsi de suite…
3) la fonction **est exécutée** et fait ses calculs et instructions
4) l'instruction `retourner` est exécutée : l'expression située après cette instruction est calculée : ceci donne la valeur que l'on récupérer dans le programme ou la fonction appelant la fonction qui vient de se terminer.

Schéma-bloc pour le programme exemple

Le schéma par bloc symbolise chaque fonction par un bloc qui ne connaît que ses variables locales. On fera donc apparaître ces variables locales à côté du bloc, et surtout, on séparera les blocs par un trait épais symbolisant le fait que les fonctions ne connaissent pas les variables des autres fonctions.



Les arguments du premier appel à la fonction toto sont calculés : il s'agit des valeurs : -12584.007 et 3.54 (on calcule, dans l'ordre, les valeurs de `y` puis `x`, dans l'ordre dans lequel ils sont donnés en argument). Ces valeurs sont rangées dans les paramètres de la fonction, dans le même ordre. Le premier paramètre se nomme `tata`, il reçoit la première valeur : -12584.007 et le second se nomme `tutu` : il reçoit la valeur 3.54.

La fonction calcule la valeur de titi, sa variable locale, par la formule donnée dans la fonction : `titi` vaut -3554.803956.

Dans le programme principal, l'écriture toto(y,x); est une expression qui à la valeur de ce que retourne la fonction appelée : c'est donc la valeur -3554.803956. Elle est affectée à la variable z.

Deuxième appel :

Le premier argument est `z*x`, qui est calculé : -12584,00600424

Le deuxième argument est `y-8.18`, qui est calculé et vaut : -12592,187

La fonction effectue le calcul et retourne la valeur de `titi` : -0.001300, affectée à la variable `t`.

Troisième appel : il est fait dans la fonction `afficher()`, la valeur retournée par la fonction sera donc affichée sans être affectée dans une variable.

Intégrité des variables locales

Les variables locales à un programme ou à une fonction ne peuvent pas être modifiées par une autre fonction lorsque l'on applique ce mécanisme pour des paramètres de type simple (`entier`, `caractere` ou `reel`). En effet, il ne faut pas confondre les arguments d'une fonction appelée et les variables du programme qui appelle cette fonction. Un argument n'est pas nécessairement une variable, et même si c'est le cas, c'est la valeur de l'argument qui est transmis à la fonction et non la variable elle-même. On peut donc en conclure qu'une variable utilisée comme argument d'un appel de fonction ne sera pas modifiée par l'appel, car c'est simplement sa valeur qui est recopiée.

Les fonctions et les tableaux

Les tableaux peuvent très bien servir en tant que paramètres de fonction ou arguments de fonction.

Syntaxe utilisée pour les entrées de type 'tableau'

un tableau est en fait une adresse, et c'est donc une adresse qui sera transmise par le biais d'une entrée d'un tel type. On ne peut pas transmettre, avec un seul paramètre, d'autres informations que l'adresse, c'est à dire la taille utile ou même la taille maximum du tableau. Si l'on veut transmettre l'une de ces informations, il faudra utiliser une entrée supplémentaire. Ainsi, un paramètre de type tableau sera défini comme un tableau contenant un certain type de valeurs, mais sans fournir ni la taille maximum, ni la taille utile.

En pratique : une entrée de type tableau sera fournie de la manière suivante :

```
type_des_valeurs_stockées nom_entree[]
```

Les crochets ne contiennent aucune valeur, ils sont juste présents pour indiquer que le type de l'entrée est un tableau contenant des valeurs d'un certain type.

Exemple : une entrée de type tableau de `reel`, nommée `tab_r`, sera définie de la manière suivante :

```
reel tab_r[]
```

entrée de type tableau d'entiers :

```
entier tab_ent[]
```

idem pour un tableau de caractere :

```
caractere tab_c[]
```

Lorsque l'on voudra traiter les valeurs stockées dans ce tableau, il faudra par contre avoir une information concernant la taille utile de ce tableau : il faudra donc associer systématiquement cette entrée à une entrée de type tableau. N'oubliez pas que, même si la taille utile est définie dans le programme principal ou dans une autre fonction que celle qui traite le tableau, cette taille utile sera stockée dans une variable à laquelle la fonction ne pourra pas accéder ! il faudra donc la lui transmettre.

Exemple : fonction affichant un tableau d'entiers :

```
fonction aff_tab(entier tablo[], entier util)
{
      entier cpt;

      pour cpt de 0 à util-1 faire
      {
            afficher(tablo[cpt]," ");
      }

      afficher("\n");
}
```

Ici, l'entrée nommé `tablo` ne permet de caractériser que le type : tableau contenant des entiers, et non sa taille maximum ou utile; on doit systématiquement lui associer une deuxième entrée de type `entier`.

appel d'une fonction avec un argument de type tableau

lorsqu'un tableau est utilisé comme un argument, il est inutile d'utiliser la notation avec les crochets. En effet, ces crochets sont utilisés pour définir le tableau, ou pour accéder à

un élément particulier stocké dans le tableau. Le tableau lui-même (c'est à dire l'adresse à laquelle sont stockées les valeurs), est repéré par son nom, sans les crochets.

Rappel : soit la définition suivante :

```
caractere tab_car[20];
```

cela indique que tab_car est un tableau contenant au plus 20 caractères. Donc `tab_car` est de type : tableau de `caractere`, ou encore pointeur vers des caractères. `tab_car` est une adresse.

Donc, lorsque l'on veut fournir à une fonction un argument qui est un tableau, on doit lui fournir une adresse. Dans notre exemple, l'argument serait : `tab_car` ou encore `&tabcar[0]` (rappelez-vous, il s'agit exactement de la même chose).

Voici, à titre d'exemple, le programme principal qui appelle la fonction `aff_tab` définie dans le paragraphe précédent :

```
programme tab_avec_fonction

entier t_ent[20] ← {1,9,5,-4,-12,0,1234};
entier tai_ut

tai_ut ← 7;

aff_tab(t_ent,tai_ut); // réalise l'affichage
```

la notation `t_ent[]`, pour un argument, n'aurait pas de sens (alors qu'elle en a un pour une entrée ou paramètre). Rappelons que l'ordinateur calcule la valeur de l'argument pour la transmettre à la fonction. **`t_ent`**, ici, est une valeur numérique qui correspond à l'adresse à laquelle sont stockées les valeurs du tableau. **`t_ent[un_indice]`** serait de type entier, ce qui ne correspond pas au type de l'entrée attendue par la fonction, et **`t_ent[]`** n'est pas une notation que l'ordinateur est capable de calculer.

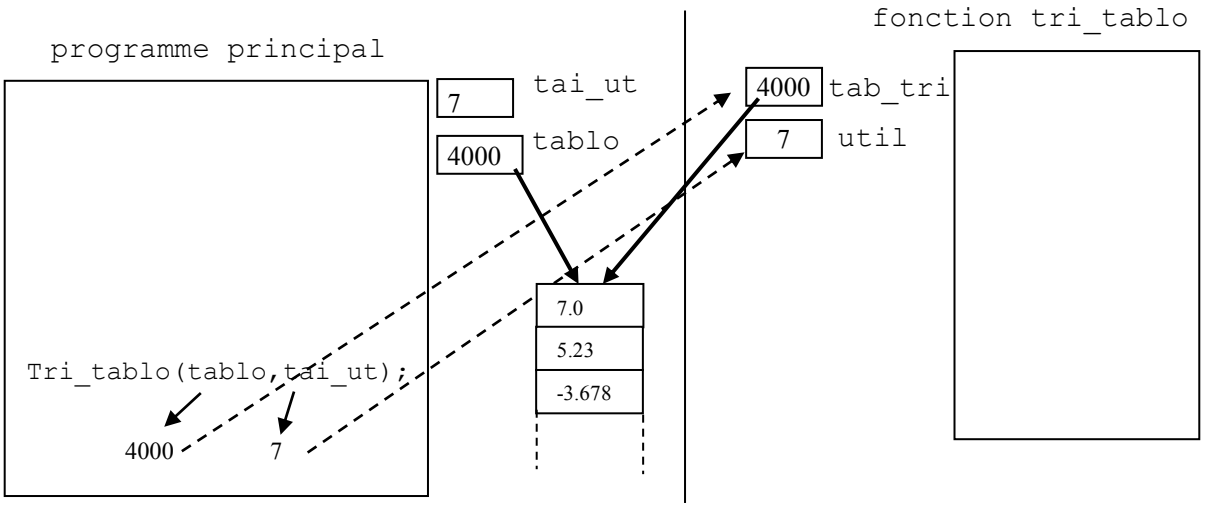
Passage de paramètres lorsque l'argument est un tableau

La méthode de passage de paramètre décrite précédemment reste la même, cependant ses effets sont différents puisque la valeur qui est transmise à la fonction est le tableau lui-même. Rappelons une fois de plus qu'un tableau stocke, tout comme un pointeur auquel il est presque équivalent, une adresse : c'est l'adresse à laquelle est stockée son élément d'indice 0.

Ainsi, le paramètre correspondant de la fonction appelée stockera la même adresse que celle qui est donnée en argument. C'est le principe même du phénomène de recopie qui intervient pour n'importe quelle valeur transmise à une fonction.

Exemple : soit un tableau `tablo` de `reel` défini dans le programme principal, on suppose que les valeurs en sont stockées à l'adresse 4000. On suppose également que l'on dispose d'une fonction qui effectue un tri de tableau de `reel` : `tri_tablo`.

```
fonction tri_tablo(entree : reel tabtri[], entier util)
{
      entier cpt1, cpt2;
      reel echange;
      pour cpt1 de 0 à util-1
      {
            pour cpt2 de 0 à (util-cpt1-2)
            {
                  si(tabtri[cpt2]> tabtri[cpt2+1]) alors
                  {
                        echange ← tabtri[cpt2];
                        tabtri[cpt2] ← tabtri[cpt2+1];
                        tabtri[cpt2+1] ← echange;
                  }
            }
      }

      retourner; // car il n'y a pas de sortie
}

programme tri_avec_fonction

reel tablo[50] ← {7.0,5.23,-3.678,424.3,5.0E+25,-1.0,0.0000008};
entier tai_ut ← 7;

tri_tablo(tablo, tai_ut);

// affichage du tableau tablo
```



L'argument qui est calculé et transmis à la fonction est le tableau lui-même, c'est à dire l'adresse à laquelle sont rangées les valeurs qui y sont stockées. Dans l'exemple fourni, cette valeur est 4000. Le paramètre `tab_tri` de la fonction de tri reçoit cette valeur 4000 et a donc accès aux valeurs stockées dans le tableau du programme principal. Lorsqu'une adresse est transmise à une fonction, le contenu (c'est à dire ce qui se situe en mémoire à l'adresse transmise) est accessible par la fonction appelante et la fonction appelée : ce phénomène est appelé : partage du contenu.

Ainsi, lorsqu'un tableau est transmis, toute modification ou traitement fait par une fonction sur les valeurs qui y sont stockées sont conservés car il n'existe qu'un seul exemplaire en mémoire de ces valeurs, et c'est l'adresse de cette zone de mémoire qui a été communiquée à la fonction.

Les fonctions et les pointeurs

Le passage de tableau en paramètre est en fait une très bonne illustration des effets de la transmission d'une adresse à une fonction : le même phénomène entre en jeu lorsque ce sont des pointeurs qui sont utilisés pour ce passage de paramètres, puisqu'un tableau est un pointeur.

Nous avons ainsi remarqué que le passage d'une adresse, dans le cas d'un tableau, permet d'obtenir un accès à une variable du programme principal à partir d'une fonction recevant l'adresse de cette variable. Nous allons donc utiliser explicitement cette transmission d'adresse de variable pour avoir un accès à son contenu, et ce grâce à la notation * déjà traitée dans le cours concernant les pointeurs.

Paramètre de type : "adresse de".

La syntaxe utilisée pour un paramètre de fonction lors de la définition de celle-ci, lorsque le paramètre est un pointeur est la suivante :

```
type_pointé *nom_du_paramètre;
```

Cela revient donc à utiliser la syntaxe de définition d'une variable de type pointeur. Pour un paramètre, cette écriture s'interprète un peu différemment, même si cette interprétation est tout à fait cohérente avec tous les aspects abordés avec les pointeurs.

Interprétation

Le paramètre `nom_du_paramètre` est l'adresse d'une valeur de `type_pointé`.

Exemple :

`entier *p1 : p1` est l'adresse d'un `entier`

`reel *p_r : p_r` est l'adresse d'un `reel`

Pourquoi faire cette distinction ici ? Tout simplement parce qu'un argument fourni à une fonction lors de son appel est une expression, et non une variable : cela signifie, entre autres, que lors de l'appel à une fonction dont un paramètre est un pointeur, l'argument associé ne devra pas obligatoirement être un pointeur, mais tout simplement une adresse.

Il pourra donc s'agir : de l'adresse d'une variable existante ou d'une adresse stockée dans un pointeur. Nous allons traiter deux exemples d'appel d'une fonction dont les paramètres sont des pointeurs. Il s'agit d'une fonction réalisant l'échange de deux valeurs.

Le très classique exemple de l'échange

```
fonction swap(entree : reel *p_1, reel *p_2)
{
      reel exg;

      exg ← *p_1;
      *p_1 ← *p_2;
      *p_2 ← exg;
}
```

Première version du programme : appel de la fonction swap en lui fournissant en argument deux adresses de variables définies dans le programme principal. On notera que l'on n'a pas besoin, dans ce cas, de définir des pointeurs dans le programme principal.

```
programme echange
{
      reel x,y;

      x ← 1.25;
      y ← -2.65;

      afficher("x vaut :", x," et y vaut :",y,"\n");

      swap(&x,&y);

      afficher("x vaut :", x," et y vaut :",y,"\n");
}
```

Le compilateur va juste vérifier que l'argument donné à l'appel de la fonction `swap()` est bien l'adresse d'une valeur de type `reel`.

Deuxième version du programme : on définit des pointeurs que l'on donnera directement en argument de la fonction

```
programme echange
{
      reel *p_x, *p_y;

      // attention, il faut allouer de la mémoire !

      p_x ← reservation(1 reel);
      p_y ← reservation(1 reel);

      *p_x ← 1.25;
      *p_y ← -2.65;

      afficher("x vaut :",*p_x," et y vaut :",*p_y,"\n");

      swap(p_x,p_y);

      afficher("x vaut :", *p_x," et y vaut :", *p_y,"\n");
}
```

Troisième et dernière version : qu'en pensez-vous ?

```
programme echange
{
      reel t_x[1], t_y[1];

      t_x[0] ← 1.25;
      t_y[0] ← -2.65;

      afficher("x vaut :",t_x[0]," et y vaut :",t_y[0],"\n");

      swap(t_x,t_y);

      afficher("x vaut :", *t_x," et y vaut :", *t_y,"\n");
}
```