3

Les fonctions

Nous avons déjà utilisé depuis le premier chapitre les fonctions `lire` et `ecrire` pour saisir et afficher des valeurs. Une fonction fournit un service dans un algorithme isolé. Lorsque votre algorithme doit effectuer plusieurs fois la même tâche, il est judicieux d'isoler cette tâche dans une fonction et de l'appeler aux moments opportuns : votre algorithme n'en sera que plus facile à écrire et à modifier.

L'intérêt de l'utilisation des fonctions est double. Il existe dans tous les langages informatiques des bibliothèques de fonctions associées à des domaines de traitements particuliers (traitement des fichiers, des images, des animations, etc.). Pour maîtriser un langage, un programmeur doit connaître et utiliser les bibliothèques de fonctions.

La maîtrise des fonctions est une étape nécessaire à la compréhension ultérieure de la notion d'objet et de méthode. Il est raisonnable d'être progressif dans l'étude d'un bloc de programme exécuté à l'extérieur d'un algorithme : la fonction en est l'exemple le plus simple.

Apprenons à utiliser et à créer des fonctions.

Les fonctions simples

Définition

> **Définition**
>
> **Fonction**
>
> Une fonction est un algorithme indépendant. L'appel (avec ou sans paramètres) de la fonction déclenche l'exécution de son bloc d'instructions. Une fonction se termine en retournant ou non une valeur.

> **Définition**
> **Procédure**
> Une procédure est une fonction qui retourne vide : aucune valeur n'est retournée.

La structure d'une fonction est la suivante :

```
fonction nomDeLaFonction(liste des paramètres): typeRetourne
Debut
       bloc d'instructions;
Fin
```

Trois étapes sont toujours nécessaires à l'exécution d'une fonction :

1. Le programme appelant interrompt son exécution.
2. La fonction appelée effectue son bloc d'instructions. Dès qu'une instruction `retourne` est exécutée, la fonction s'arrête.
3. Le programme appelant reprend alors son exécution.

> **Définition**
> **L'arrêt de la fonction**
> Une fonction s'arrête lorsque son exécution atteint la fin du bloc d'instructions, ou lorsque l'instruction `retourne` est exécutée (avec ou sans valeur).

Le programmeur doit penser à concevoir et écrire des fonctions pour améliorer son programme. Il y gagnera sur plusieurs points :

- Le code des algorithmes est plus simple, plus clair et plus court. Dans un algorithme, appeler une fonction se fait en une seule ligne et la fonction peut être appelée à plusieurs reprises.
- Une seule modification dans la fonction sera automatiquement répercutée sur tous les algorithmes qui utilisent cette fonction.
- L'utilisation de fonctions génériques dans des algorithmes différents permet de réutiliser son travail et de gagner du temps.

Fonction sans valeur retournée

Apprenons à écrire et utiliser une fonction simple qui doit afficher `"bonjour"`. Cette fonction ne retourne pas de valeur : ceci est signalé en précisant qu'elle retourne `vide`.

```
fonction afficheBonjour(): vide
Debut
       ecrire("bonjour");
       retourne;
Fin
```

Une fonction se termine toujours par l'instruction `retourne`. Cette fonction effectuera les instructions situées entre `Debut` et `Fin`.

Écrivons un algorithme qui appelle la fonction `afficheBonjour()`.

```
Algorithme utilise-fonction
Debut
       afficheBonjour();
Fin
```

Voici la suite des instructions exécutées au cours du temps : voir figure 3-1.

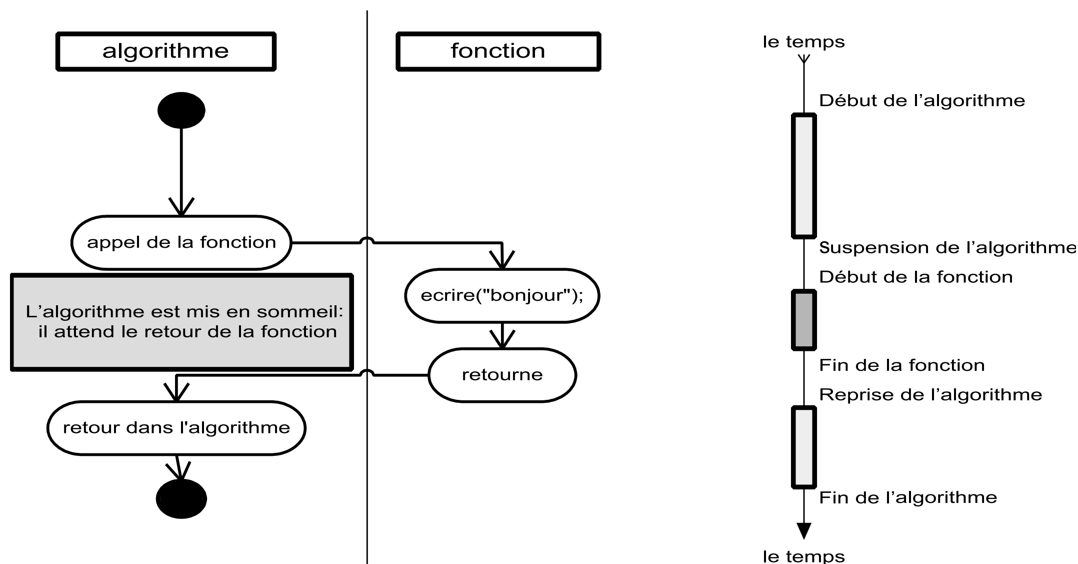


Figure 3-1

Passage de l'algorithme à la fonction.

Imaginons un autre algorithme qui appelle 10 fois la fonction `afficheBonjour()`.

```
Algorithme utilise-fonction-10
variables: indice: entier;
Debut
       indice ← 0;
       tant_que (indice < 10) faire
       {
          afficheBonjour();
          indice ← indice + 1;
       }
Fin
```

Pour faciliter la lecture des algorithmes, il convient de respecter des règles (inspirées du langage Java) pour nommer les fonctions.

- Le nom d'une fonction commence par une minuscule.
- Le nom d'une fonction ne comporte pas d'espace.

- Si le nom de la fonction est composé de plusieurs mots, faire commencer chacun d'eux par une majuscule (par exemple : `sommeDeDeuxEntiers`, `valeurMax`) et ne pas faire figurer de traits d'union.

Fonction avec une valeur retournée

> **Définition**
>
> **La valeur de retour**
>
> Une fonction peut retourner une valeur au programme appelant. Cette valeur est unique. Le retour de la valeur signifie l'arrêt de la fonction.

Introduisons une autre fonction qui permet de lire une note entre 0 et 20. L'algorithme associé a été étudié au cours du chapitre 2, il suffit de le transformer en fonction :

```
fonction lireNote(): entier
variables: note: entier;
Debut
  ecrire("Entrez une note :");
  lire(note);                                          // l'utilisateur entre la note

  tant_que ((note < 0) OU (note > 20)) faire
  {
     ecrire("Vous avez fait une erreur, essayez encore :");
                                                       // message d'erreur affiché
     lire(note);                                       // on recommence la saisie
  }
  retourne(note);
Fin
```

La fonction `lireNote()` retourne une valeur entière à la fin de son exécution. L'instruction `retourne` indique la fin immédiate de la fonction et le retour dans le programme appelant.

L'environnement des données

Les paramètres

Le programme appelant doit donner à certaines fonctions des valeurs pour effectuer ses calculs. La fonction associe à ses valeurs des variables afin de les manipuler : ce sont les paramètres de la fonction.

> **Définition**
>
> **Les paramètres**
>
> Un paramètre est une variable locale à une fonction. Il possède dès le début de la fonction la valeur passée par le programme appelant.

Le passage des paramètres

Prenons l'exemple d'une fonction `maxDe2Valeurs` qui retourne le maximum de deux valeurs passées en paramètre. Cette fonction doit retourner une valeur entière : celle-ci est calculée en tenant compte des deux valeurs passées en paramètres.

```
fonction maxDe2Valeurs(p1: entier, p2: entier): entier
variables: resultat: entier;
Debut
       si (p1 < p2 ) alors
       {
          resultat ← p2;
       } sinon {
          resultat ← p1;
       }
          retourne(resultat);
Fin
```

Cette fonction effectuera les opérations situées entre `Debut` et `Fin`. Soit un algorithme qui appelle la fonction `maxDe2Valeurs()`.

```
Algorithme utilise-fonction-max
variables: valeur1, max: entier;
Debut
      lire(valeur1);
      max ← maxDe2Valeurs(valeur1, 25);
      ecrire(max);
Fin
```

L'utilisation de la fonction s'effectue toujours en trois temps :

1. Avant de modifier la valeur de la variable `max`, l'algorithme s'arrête pour évaluer l'expression `maxDe2Valeurs(valeur1, 25)`. Supposons que l'utilisateur ait saisi la valeur `12` pour `valeur1`, l'expression à évaluer est alors `maxDe2Valeurs(12, 25)`.
2. La fonction travaille dans *un environnement de données* complètement dissocié de celui du programme appelant. Les seules valeurs connues de la fonction sont :
   – les deux *paramètres* `p1` et `p2` ;
   – la *variable* locale `resultat`.
3. Au retour de la fonction, l'expression `maxDe2Valeurs(12,25)` est remplacée par la valeur `25`. L'algorithme continue son déroulement comme si l'expression avait été `max ← 25`.

variables: resultat: entier;			
Debut	p1 = 12	p2 = 25	résultat = ?
si (p1 < p2) **alors**	p1 = 12	p2 = 25	résultat = ?
resultat ← p2;	p1 = 12	p2 = 25	résultat = 25
sinon	instruction non exécutée		
resultat ← p1;	instruction non exécutée		
retourne(resultat);	Arrêt. La valeur 25 est retournée		
Fin	Variables et paramètres disparaissent à la Fin		

On remarquera qu'il faut éviter autant que possible d'écrire l'instruction `retourne` au milieu de la fonction. La lisibilité est alors moins facile. Par exemple, la fonction suivante est identique à la précédente :

```
fonction maxDe2Valeurs(p1: entier, p2: entier): entier
Debut
       si (p1 < p2 ) alors
       {
              retourne(p2);
       } sinon {
              retourne(p1);
       }
Fin
```

Néanmoins, dans le cas d'une fonction comportant plusieurs instructions `retourne`, faites attention : la fonction se termine immédiatement lors de la première instruction `retourne` exécutée.

Les données d'une fonction

Dans l'utilisation et l'écriture d'une fonction, la plus grande difficulté est de comprendre l'ensemble des données auxquelles la fonction a accès.

> **Définition**
>
> **L'environnement de données**
>
> Un environnement de données, appelé aussi espace d'adressage, correspond à l'ensemble des variables associées exclusivement à un algorithme ou à une fonction.

Une variable définie dans un algorithme (respectivement dans une fonction), existe uniquement le temps limité de l'exécution de l'algorithme (respectivement de la fonction). Peu importe le nom des variables définies dans la fonction pour pouvoir l'utiliser. Ainsi, un programmeur ne donne jamais le nom des variables internes à une fonction dont il est l'auteur.

Pour revenir à l'exemple précédent, on peut écrire la définition de la même fonction de différentes manières :

```
fonction maxDe2Valeurs(p1: entier, p2: entier): entier
```

ou

```
fonction maxDe2Valeurs(valeur1: entier, valeur2: entier): entier
```

ou

```
fonction maxDe2Valeurs(entier, entier): entier
```

Dans les trois cas, l'utilisation de la fonction est identique :

```
leMax ← maxDe2Valeurs(455, 48);
```

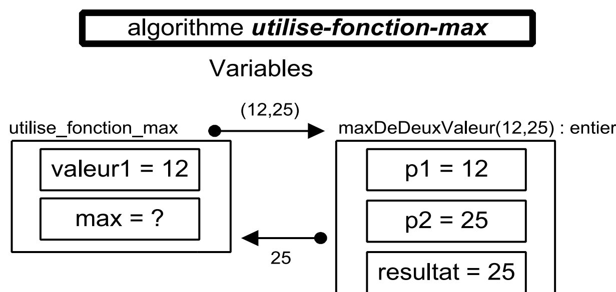
Il est tout à fait possible que 2 variables, l'une déclarée dans le programme appelant et l'autre déclarée dans la fonction, portent le même nom. Elles peuvent être du même type ou non, peu importe, puisqu'elles sont utilisées de manière différente dans des environnements de données différents.

À travers le schéma mémoire (figure 3-2), nous visualisons que l'algorithme et la fonction sont dans des « boîtes indépendantes » représentant les espaces d'adressage indépendants. Il n'existe aucun moyen pour l'algorithme d'avoir accès aux variables de la fonction, ni à la fonction d'avoir accès aux variables de l'algorithme. Le seul échange se fait :

- de l'algorithme vers la fonction, en passant des valeurs grâce aux paramètres ;
- de la fonction vers l'algorithme en retournant une seule et unique valeur.

Figure 3-2

Deux environnements de données distincts.



On dit souvent que « la fonction `maxDe2Valeurs()` retourne la variable entière `resultat` » : c'est un abus de langage. En effet, il faudrait dire que « la fonction `maxDe2Valeurs()` retourne la valeur de la variable entière `resultat` » : c'est la valeur 25 qui est retournée au programme appelant dans notre exemple. L'algorithme `utilise_fonction_max` ignore l'environnement des données de la fonction (donc la variable `resultat`).

Les paramètres et les variables

La plupart du temps, l'exécution d'une fonction est paramétrable grâce à des valeurs qui lui sont passées. Les paramètres sont des variables de la fonction : il est donc faux de vouloir les redéfinir dans la zone de déclaration des variables.

Une fonction peut accéder à deux types de données :

- Les **paramètres**, dont les valeurs sont connues dès le début de la fonction. Les valeurs sont passées en paramètres. Il est inutile de nommer les paramètres avec le même nom que les variables utilisées lors de l'appel de la fonction.
- Les **variables** (appelée variables locales) définies dans le bloc de déclaration des variables.

  Au cours de l'exécution d'une fonction, les variables définies dans le programme appelant sont inconnues : aussi bien leur nom que leur valeur.

La valeur retournée est unique. Il est impossible pour une fonction de retourner plusieurs valeurs, mais également de modifier directement une variable du programme appelant (sauf la fonction `lire` introduite au premier chapitre).

Techniques

Définir une fonction

> **Définition**
>
> **La signature d'une fonction**
>
> La signature d'une fonction décrit les éléments permettant de l'appeler correctement :
>
> - le nom de la fonction ;
> - le type (et l'ordre) des paramètres ;
> - le type de la valeur retournée.

Un programmeur qui souhaite utiliser une fonction n'a pas besoin de connaître le corps de la fonction (situé entre `Debut` et `Fin`), ni même le nom ou les types des variables internes à la fonction, mais seulement les caractéristiques nécessaires à son utilisation : sa signature. Il s'agit en fait de la carte d'identité de la fonction.

Deux fonctions ayant des signatures différentes sont différentes.

Bien sûr, le programmeur doit connaître l'action de la fonction qu'il utilise en plus de savoir l'appeler.

Quelques signatures de fonctions :

Signature de la fonction	Résultat de la fonction
`afficheBonjour(): vide`	Affiche « bonjour ».
`maxDe2Valeurs(entier, entier): entier`	Retourne une valeur entière, le maximum des deux paramètres.
`hasard(entier): entier`	Retourne une valeur entière aléatoire comprise entre 0 et la valeur passée en paramètre comprise.
`partieEntiere(reel): entier`	Retourne la valeur de l'entier égale à la partie entière du réel passé en paramètre.
`racineCarree(entier): reel`	Retourne le réel égal à la racine carrée de la valeur positive passée en paramètre.
`valeurAbsolue(entier): entier`	Retourne la valeur absolue de la valeur entière passée en paramètre.
`valeurAbsolue(reel): reel`	Retourne la valeur absolue de la valeur réelle passée en paramètre.

Les cinq dernières fonctions sont employées dans la résolution de certains problèmes.

> **Définition**
>
> **Le polymorphisme paramétrique**
>
> Deux fonctions peuvent avoir le même nom et des paramètres différents en nombre ou en type. Le polymorphisme paramétrique garantit automatiquement l'exécution de la bonne fonction associée au bon nombre de paramètres et à leurs types. En effet, les programmes identifient une fonction par sa signature (et pas uniquement par son nom).

Les erreurs fréquentes à éviter

Erreurs à éviter dans l'utilisation d'une fonction :

- Oublier les parenthèses.
- Ne pas respecter le type de retour.
- Ne pas respecter le type des paramètres.
- Ne pas réécrire à chaque fois une fonction qui existe déjà.
- Croire que la fonction peut modifier la variable du programme appelant.

```
Algorithme utilise-fonction-5-erreurs
variables:valeur1, max: entier;
Debut
        afficheBonjour;        // mettre les parenthèses : afficheBonjour() ;
        hasard(5);             // et la valeur retournée ? valeur1←hasard(s) ;
        hasard(2.5);           // le type du paramètre ?
Fin
```

Erreurs à éviter dans l'écriture d'une fonction :

- Donner le même nom à un paramètre et à une variable.
- Placer plusieurs `retourne` consécutifs dans la fonction.
- Vouloir retourner plusieurs valeurs.
- Oublier de retourner la valeur ou retourner une valeur du mauvais type.
- Vouloir continuer un traitement après l'instruction `retourne`.
- Penser qu'en modifiant la valeur d'un paramètre, celui-ci sera modifié dans le programme appelant.

```
fonction fonctionErreur(p1:entier): entier
variables: resultat: réel;
             p1: entier;              // erreur : p1 est déjà un paramètre !
Debut
        retourne(resultat + p1);      // erreur : mauvais type de retour
        p1 ← 2;                       // la variable passée à la
                                      // fonction n'aura pas été modifiée
        retourne(2, resultat);        // erreur : on ne peut pas
                                      // retourner plusieurs valeurs
        resultat ← p1;                // erreur : cette opération
                                      // ne sera jamais exécutée
Fin
```

Les paramètres instance

En utilisant les chaînes et les dates dans des fonctions, nous allons étudier une manière de modifier directement une variable du programme appelant dans la fonction.

Imaginons une fonction qui convertit une chaîne de caractères en minuscules.

Fonction qui retourne une instance

La première approche consiste à définir la donnée (la chaîne à convertir) et le résultat (la chaîne en minuscule).

La signature de la fonction serait alors :

```
fonction convertirEnMinuscule(phrase:Chaine): Chaine
```

La solution serait de ne pas toucher à la chaîne passée en paramètre, et de travailler sur une copie qui serait retournée : l'utilisateur garderait alors la chaîne avec des majuscules et obtiendrait la chaîne composée uniquement de minuscules. La fonction solution serait ainsi :

```
fonction convertirEnMinuscule(c:Chaine): Chaine
variables: car: caractere;
           indice: entier;
           resultat: Chaine;
Debut
  resultat ← new Chaine(c);

  indice ← 0;                                    // initialisation de l'indice à 0
  tant_que (indice < resultat.longueur()) faire
  {
     car ← resultat.iemeCar(indice);
     si ((car ≥ 'A') ET (car ≤ 'Z')) alors
     {   car ← car + ('a' – 'A');
         resultat.modifierIeme(indice, car);
     }
     indice ← indice + 1;                        // incrémentation de l'indice
  }
  retourne resultat;
Fin
```

Et son utilisation :

```
Algorithme utilise_fonction_minuscule
variables: ch1, ch2: Chaine ;
Debut
  ch1 ← Chaine("DuPoNd");
  ch2 ← convertirEnMinuscule(ch1);
  ch2.ecrire();
Fin
```

Représentons le schéma mémoire de l'appel de la fonction et les deux environnements de données (voir figure 3-3).

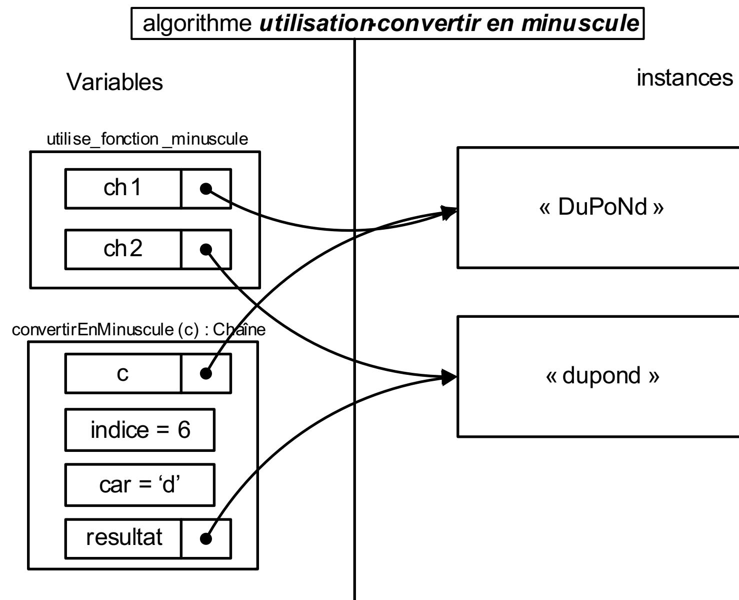


Figure 3-3
État de la mémoire.

Ce schéma mémoire nous montre deux aspects importants des fonctions :

- Le programme appelant et la fonction sont dans des environnements de données différents, mais ils peuvent contenir des variables qui désignent la même instance (la même case).
- Le paramètre de la fonction est considéré comme une variable locale pour celle-ci.

Fonction qui modifie une instance paramètre

Une autre solution serait de convertir directement l'instance passée en paramètre. La signature de la fonction devient alors :

```
fonction convertirEnMinuscule(Chaine): vide
```

Cette fonction travaille directement sur l'instance indiquée par le programme appelant.

```
fonction convertirEnMinuscule(ch: Chaine): vide
variables: car: caractere;
           indice: entier;
Debut
  indice ← 0;                              // initialisation de l'indice à 0
  tant_que (indice < ch.longueur()) faire
```

```
  {
     car ← ch.iemeCar();
     si ((car ≥ 'A') ET (car ≤ 'Z')) alors
     {   car ← car + ('a' – 'A');
         ch.modifierIeme(indice, car);
     }
     indice ← indice + 1;                    // incrémentation de l'indice
  }
  retourne;
Fin
```

Et son utilisation :

```
Algorithme utilise-fonction-minuscule
variables: ch1: Chaine;
Debut
  ch1 ← Chaine("DuPoNd");
  convertirEnMinuscule(ch1);
  ch1.ecrire();
Fin
```

Représentons le schéma mémoire de l'appel à la fonction en cours d'exécution (après trois tours de boucles) : voir figure 3-4.

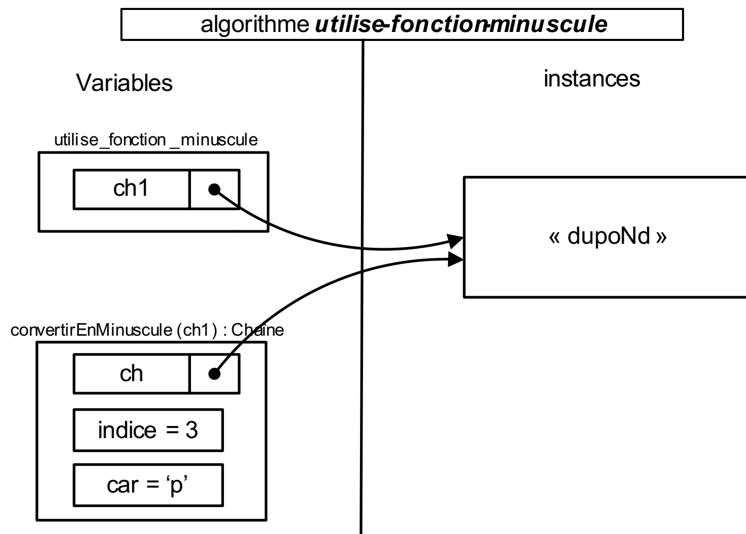


Figure 3-4
État de la mémoire à l'itération indice égale 3.

Cette technique est très utilisée : elle permet indirectement de partager des environnements de données.

La récursivité

Définition

La notion de récursivité est assez naturelle mais pas toujours très simple à mettre en œuvre.

> **Définition**
>
> **Fonction récursive**
>
> Une fonction est dite récursive si elle s'appelle elle-même.

La récursivité est une technique de programmation très puissante : elle permet quelquefois de trouver rapidement des solutions élégantes à des problèmes compliqués. Certains domaines sont plus propices aux solutions récursives simples (et des solutions itératives très compliquées) comme les mathématiques, la géométrie, les hiérarchies, etc. La difficulté est de penser à cette technique de programmation pour imaginer un algorithme.

Deux conditions sont nécessaires pour être en mesure d'utiliser la récursivité :

- Il faut pouvoir exprimer un algorithme sous forme d'une fonction de telle manière que sa valeur à un certain rang ne dépende que de sa valeur aux rangs inférieurs.
- On doit aussi connaître la solution pour les rangs initiaux.

La technique de programmation est toujours la même : elle est assez déconcertante au début.

> **Technique pour écrire une fonction récursive**
>
> Il suffit d'utiliser la fonction que vous n'avez pas encore écrite en supposant qu'elle donne déjà un résultat.

Un algorithme récursif se compose de deux parties :

1. Au moins une condition d'arrêt des appels récursifs, où les valeurs à déterminer sont immédiatement connues.
2. Un appel récursif. La fonction s'appelle elle-même, dans un autre environnement.

Pour une fonction récursive qui retourne vide :

```
fonction fonctionRecursive(liste des parametres): vide
Debut
  si (condition d'arrêt) alors                    // condition d'arrêt et de retour
  {
     retourne;                                    // à mettre au début du corps de la méthode
  }
  sinon
  {
     fonctionRecursive(liste des nouveaux parametres);
                                                  // appel récursif
  }
Fin
```

Pour une fonction récursive qui retourne une valeur :

```
fonction fonctionRecursive(liste des paramètres): typeRetourne
Debut
  si (condition d'arrêt) alors   // condition d'arrêt et de retour
  {
     retourne(…);                // à mettre au début du corps de la méthode
  }
  sinon
  {
                                 // appel récursif
     retourne(fonctionRecursive(liste des nouveaux parametres));
  }
Fin
```

Cette structure est à connaître par cœur, tout comme l'exemple suivant de la fonction factorielle.

La fonction factorielle

Définition

Un premier exemple de fonction récursive, très classique et par cela incontournable, va éclairer la notion de récursivité. Rappelons que la fonction factorielle est définie par :

`factorielle(n) = n ! = 1 × 2 × … × (n–1) × n.` Donc `factorielle(1) = 1`, `factorielle(2) = 2`, `factorielle(3) = 1 × 2 × 3 = 6` et `factorielle(4) = 1 × 2 × 3 × 4 = 24`

On peut réécrire la fonction `factorielle(n)` d'une manière récurrente strictement équivalente à la précédente :

- `factorielle(1) = 1;`
- `factorielle(n) = n × factorielle(n–1)`, pour `n > 0`.

À partir de cette définition récurrente, il va être assez simple de définir la fonction factorielle de manière récursive.

La fonction

Pour écrire `factorielle(n)`, il suffit de se dire que la fonction `factorielle(n–1)` donne déjà le bon résultat : c'est assez déroutant puisqu'on est justement en train d'écrire la fonction `factorielle`. De là, il vient naturellement que `factorielle(n) = n × factorielle(n–1)` ; il suffit d'écrire la condition d'arrêt et le tour est joué.

```
fonction factorielle(nb: entier): entier
variables: f: entier;
Debut
  si (nb = 1) alors
  {                              // condition de sortie
     f ← 1;
     retourne(f);                // sortie
  }
  sinon {
     f ← nb × factorielle(nb–1); // la fonction s'appelle elle-même
     retourne(f);
  }
Fin
```

Et l'algorithme d'utilisation :

```
Algorithme ManipulationDeFactorielle
Debut
   ecrire(factorielle(3));
Fin
```

L'exécution

Comment fonctionne cet algorithme ? Calculons `factorielle(3)` (voir figure 3-5).

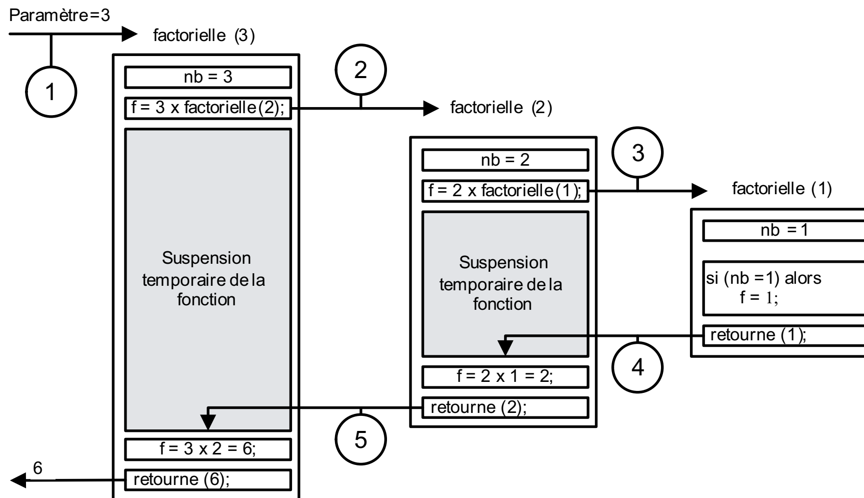


Figure 3-5
Calcul de factorielle de 3.

Chaque fonction s'exécute dans son environnement de données associé : lors de l'exécution du calcul de `factorielle(1)`, il y a trois variables `nb` définies avec trois valeurs différentes dans chaque environnement de données.

Précisons le déroulement des calculs :

Le calcul de `factorielle(3)` est lancé (étape n° 1).

Pour évaluer la valeur `2 × factorielle(2)`, le calcul de `factorielle(3)` se suspend pour connaître la valeur de `factorielle(2)` (étape n° 2).

Le calcul de `factorielle(2)` se suspend à son tour pour évaluer `factorielle(1)` (étape n° 3).

Grâce à la condition d'arrêt, `factorielle(1)` retourne 1 : cette valeur remplace `factorielle(1)` dans le calcul suspendu de `factorielle(2)` (étape n° 4).

Le calcul de `factorielle(2)` peut reprendre là où il était suspendu et s'effectuer, `factorielle(2)` retourne 2 (étape n° 5) et le calcul de `factorielle(3)` peut reprendre et s'effectuer pour produire 6.

Finalement, on a calculé :

```
factorielle(3) = 3 × factorielle(2)
               = 3 × (2 × factorielle(1))
               = 3 × (2 ×    (1)       )
               = 6
```

Notons bien qu'un tel mode de traitement est différent d'un traitement itératif dans lequel chaque calcul s'effectue sur des valeurs toujours connues. Mais si le traitement est différent, il est clair que les calculs reviennent au même.

Il y a dans l'écriture de l'algorithme, l'appel à `factorielle(nb–1)` dont on ne connaît pas a priori la valeur au moment où ce terme apparaît, sauf si `nb` vaut `1`. Mais il est essentiel de supposer que l'on sait calculer `factorielle(nb–1)` pour écrire récursivement l'algorithme de calcul de `factorielle(nb)`.

La même méthode écrite de manière plus concise donne :

```
fonction factorielle(nb: entier): entier
Debut
  si (nb = 1) alors
      retourne 1;
  retourne(nb × factorielle(nb–1)); // la fonction s'appelle elle-même
Fin
```

Rechercher une valeur dans un tableau

Ce problème a déjà été abordé de manière itérative à la fin du deuxième chapitre. Introduisons la fonction recherche : elle devra contenir comme paramètres le tableau étudié, sa taille et la valeur cherchée.

```
fonction chercher(t: tableau[] d'entiers, taille: entier,
                  valeur: entier): entier
```

Pour chercher une valeur dans un tableau de manière récursive, il faut supposer que cette valeur a été trouvée (ou pas) au rang inférieur, donc dans tout le tableau sauf la 1e case.

Il reste donc à tester le 1er élément : s'il est égal à la valeur cherchée, on retourne sa place, sinon on retourne la solution trouvée au rang inférieur.

La condition d'arrêt a lieu simplement quand il ne reste pas de case à tester : la valeur n'a alors pas été trouvée.

Il faut donc introduire aussi le rang `debut` de l'élément à tester.

```
fonction chercher(t: tableau[] d'entiers, taille: entier,
                    valeur: entier, debut: entier): entier
Debut
        si (debut ≥ taille) alors      // condition d'arrêt
```

```
            retourne (−1);
        si (t[debut] = valeur) alors   // on a trouvé
            retourne debut;
        retourne (chercher(t, taille, valeur, debut + 1));
                                        // appel  récursif entre debut + 1 et taille
Fin
```

La suite de Fibonacci

La suite de Fibonacci est une suite récurrente dont chaque terme dépend des deux précédents. Elle est définie par :

$U_0 = 0$ et $U_1 = 1$

$U_n = U_{n-1} + U_{n-2}$, pour tout entier n tel que $2 \leq n$.

Par exemple, $U_2 = U_1 + U_0 = 1 + 0 = 1$, $U_3 = U_2 + U_1 = 1 + 1 = 2$, $U_4 = U_3 + U_2 = 3$.

On désire calculer tout terme de la suite de Fibonacci de rang n, pour tout entier n donné.

Deux méthodes s'offrent à nous :

- La méthode itérative, avec une boucle `tant_que` qui va permettre de calculer tous les termes du premier jusqu'au $n^{ème}$.
- La méthode récursive. Elle sera comparable à celle utilisée pour le calcul récursif de factorielle et suit très exactement la définition de la suite de Fibonacci.

Pour écrire `U(n)`, on suppose connues et justes les valeurs retournées par `U(n−1)` et `U(n−2)` (alors même qu'on essaye d'écrire la fonction `U(n)`).

L'algorithme s'écrit dans le même esprit que celui du calcul de factorielle, en s'appuyant simplement sur la définition mathématique de la suite.

```
fonction Fibonacci(n: entier): entier
// Explication : calcul récursif à partir de la formule Un = Un−1 + Un−2
Debut
  si (n = 0) alors              // première condition d'arrêt et de retour
      retourne(0);              // cas où n=1
  sinon
  {
     si (n = 1) alors           // seconde condition d'arrêt et de retour
                                // cas où n=2
         retourne(1);
     sinon                      // appel récursif en utilisant la formule.
         retourne(Fibonacci(n−1) + Fibonacci(n−2));
  }
Fin
```

Les erreurs à ne pas commettre

L'utilisation de la récursivité semblera évidente pour certains, et demandera beaucoup plus de temps à d'autres. Il existe néanmoins certaines règles et certaines techniques qu'il faut garder à l'esprit :

- Ne pas mettre de boucle `tant_que` dans une fonction récursive (c'est faux dans 99 % des cas).
- Ne pas oublier la condition d'arrêt.
- Ne pas hésiter à utiliser le résultat de la fonction que vous êtes en train d'écrire.
- Ne pas mettre une instruction `retourne` au milieu de votre fonction récursive : les instructions suivantes ne seraient pas exécutées.

La récursivité terminale

La récursivité terminale est une notion qui peut améliorer nettement les performances de vos algorithmes. En effet, l'exécution d'une fonction utilisant une récursivité terminale est transformée en général en fonction itérative (plus rapide et moins gourmande en mémoire) par le compilateur.

> **Définition**
>
> **La récursivité terminale**
>
> Une fonction est récursive terminale si elle retourne sans autre calcul la valeur obtenue par son appel récursif.

La dernière ligne d'une telle fonction sera :

```
retourne(fonction(paramètres));
```

La fonction factorielle précédente utilise-t-elle la récursivité terminale ? La fonction factorielle se terminait par :

```
retourne(nb × factorielle(nb–1));
```

Ce n'est pas une récursivité terminale, l'évaluation de l'appel récursif `factorielle(nb–1)` est suivie par la multiplication par `nb` avant le `retourne`. La version récursive terminale serait :

```
fonction factorielle(nb: entier, resultat: entier): entier
Debut
  si (nb = 1) alors
    retourne resultat;
  retourne(factorielle(nb–1, nb×resultat));
  // la fonction s'appelle elle-même
Fin
```

Cette fonction est appelée en mettant initialement le résultat à 1 par :

```
fonction factorielle(nb: entier): entier
Debut
    retourne(factorielle(nb,1));          // appel de la fonction récursive terminale
Fin
```

Le paramètre `resultat` est calculé uniquement au fur et à mesure de l'appel récursif : `factorielle(3,1)` retourne la valeur `factorielle(2,3)` qui retourne la valeur `factorielle(1,6)` qui retourne `6`.

Cette fonction factorielle terminale est souvent transformée par le compilateur en fonction itérative.

```
fonction factorielle(nb: entier): entier

variables: resultat: entier;
Debut
        resultat ← 1;
        tant_que (nb ≠ 1) alors {
              resultat ← nb × resultat;
              nb = nb - 1;
        }
        retourne(resultat);
Fin
```

La version itérative ne crée qu'une seule variable `resultat`, ce qui explique le gain de place mémoire. L'ordinateur nécessite un peu de temps pour appeler une fonction, notamment pour changer l'espace d'adressage : la vitesse d'exécution est ainsi accrue dans la dernière version.

Concrètement en Java, voici les temps de calculs en millisecondes obtenus sur un calcul des factorielles en récursivité classique et terminale.

Récursivité	10 !	70 !	120 !	184 !
classique	5	6	7	12
terminale	1	2	5	5

Je vous conseille de tester vous-même le programme, et de remarquer qu'après 185, la tendance s'inverse en raison du temps de calcul nécessaire pour multiplier et additionner des entiers contenant plus de cent chiffres chacun.

Le gain de vitesse est encore plus impressionnant dans le calcul des éléments de Fibonacci : je vous laisse le soin de le découvrir au cours d'un exercice de bilan qui suit.

Exercices de bilan

Exercice 3.1 Reprendre les exercices du chapitre 2 et introduire les fonctions utiles.

Exercice 3.2 Écrire une fonction qui retourne le plus grand de deux entiers passés en paramètres. Même exercice avec trois entiers.

Exercice 3.3 Le jeu de cartes. Écrire une fonction qui permet de mélanger un jeu de 32 cartes.

Exercice 3.4 Écrire une fonction qui affiche un tableau à l'envers par une méthode itérative et récursive. Le tableau et la position du premier élément à afficher sont passés en paramètre.

Exercice 3.5 Écrire la fonction de Fibonacci en récursif terminal.

Exercice 3.6 Écrire une fonction qui retourne la somme de deux entiers `somme(entier, entier) retourne entier` et un algorithme qui l'utilise. Expliquer à travers cet exemple la notion de variables locales.

Exercice 3.7 Écrire un programme qui demande à l'utilisateur de deviner un nombre entre 1 et 1000. À chaque proposition, le programme indique si le nombre à trouver est inférieur ou supérieur à celui saisi.