Table des matières

1 Structures de données

Un programme est le codage, sous forme d'instructions, des opérations d'un algorithme. Un algorithme est un enchaînement judicieux d'opérations permettant d'obtenir la solution d'un problème ou d'une classe de problèmes. Nous avions distingué, dans la deuxième partie de ce cours, trois grandes phases pour la programmation : Une phase d'analyse, une phase de conception de l'algorithme et une phase d'implémentation du programme. (On aurait d'ailleurs pu ajouter des phases supplémentaires, comme une phase de tests, de vérification et de mise au point du programme obtenu, et une phase de maintenance et évolutions du programme).

En ce qui concerne les phases d'analyse et de conception, l'attention est portée sur les données ou objets du monde du problème analysé (ou problème que l'on cherche à résoudre), sur leurs structures et sur leurs relations mutuelles. De cette analyse ressortira des rapports entre ces données d'entrées, et les objets ou données que l'on souhaite mettre en évidence comme *sorties*. On catégorise ainsi l'univers représenté (celui du problème analysé) en mettant en évidence des classes d'objets, ou types d'objets particuliers. Les relations entre ces objets sont décrites comme relations entre types d'objets, ou types abstraits de données (TAD).

Ces relations étant essentiellement structurelles, elles seront traduites dans un langage particulier de programmation, au moment de la troisième phase, la phase d'implémentation, sous la forme de ce qu'on appellera des structures de données. Les structures de données seront les représentations informatiques particulières que l'on peut utiliser pour représenter les types abstraits de données (TAD).

Dans un langage impératif comme le langage C, qualifié parfois de langage de programmation « impératif à affectation », le concept central est celui de variable. Les outils classiques dont on dispose pour structurer ces variables sont leurs types, avec bien évidemment des structurations de données particulières construites à partir de types de bases, de tableaux, d'enregistrements ou de pointeurs. Mais avant d'introduire aux enregistrements et aux pointeurs, nous allons donner quelques suppléments sur les types.

1.1 La définition de nouveaux types

La partie déclarative d'un programme C peut comporter la définition de nouveaux types. Définir un nouveau type, c'est associer un identificateur à un ensemble de valeurs qui constituent le type qu'on définit. Ainsi, la construction

```
typedef <type> <nom_de_type> ;
```

ou plus généralement

```
typedef <type> <nom_de_type1>, …, <nom_de_typeN> ;
```

permet de définir un ou plusieurs types à partir d'un autre type. Ainsi, par exemple, on peut redéfinir un type pour des données destinées à représenter la position (x, y) de points sur un écran avec :

```
typedef short int Position ;
```

de même, on peut définir un type `Dimension` permettant de coder des largeurs et des hauteurs de fénêtres :

```
typedef int Dimension ;
```

1.2 Types énumérés

Une **énumération** permet de définir une suite de constantes symboliques d'un type spécifique, appelé **type énumération**. La définition d'un type énumération a la forme suivante :

```
typedef enum { <liste des noms de constantes>} <nom_type> ;
```

Exemple :

```
typedef enum {VERT, BLEU, ROUGE, ORANGE} Couleur ;
Couleur teinte1, teinte2 = ROUGE ;
```

Ici, on a deux variables de type `Couleur`, la deuxième étant initialisée à `ROUGE`.

2 Enregistrements

Les enregistrements sont, comme les tableaux, des ensembles structurés de variables. Mais si les différentes variables qui composent un tableau sont toutes du même type, les variables regroupées dans un enregistrement peuvent être de types différents.

On appelle cellules les variables qui sont regroupées en un tableau. Dans le cas des enregistrements, on parle de **membres**.

2.1 Exemple : Description d'un cours universitaire

Pour décrire un cours universitaire, on veut représenter les informations suivantes :

1. L'intitulé,
2. L'année,
3. La matière,
4. Le nombre d'étudiants inscrits,
5. La formation.

On peut évidemment utiliser différentes variables, mais il faudrait introduire 5 variables différentes par cours à décrire. Si l'on veut décrire ne serait-ce que 5 cours, il faudrait déjà introduire 25 variables et mémoriser quelles sont les variables concernant le même cours. Déclarer autant de variables est pour le moins fastidieux. Les manipuler le serait encore davantage.

On a donc besoin d'ensembles structurés de variables. Un tableau est dans ce cas inutilisable car les informations à mémoriser ne sont pas toutes du même type. Pour décrire un cours, on utilisera donc une structure *d'enregistrement*, qui permet de définir un ensemble de variables de types quelconques. Pour définir un enregistrement, on définit d'abord son type, pour préciser le nom et le type de ces membres :

Défintion de type *typeCours* = ***enregistrement***
intitule : chaîne de caractères
annee : entier
matiere : chaîne de caractères
nbInscrits : entier
formation : chaîne de caractères
fin enregistrement

On peut alors introduire différentes variables c1, c2, c3…. qui auront le type *typeCours* :
5 variables de type typeCours ***:*** *c1, c2, c3, c4, c5*

Chacune de ces variables est un enregistrement qui comporte 5 membres. Pour accéder à un membre d'un enregistrement, on utilise l'identificateur de la variable d'enregistrement suivi d'un point, suivi du nom (ou identificateur) du membre. Ainsi, on peut initialiser tour à tour les membres de l'enregistrement c1 :

c1.intitule ← "Programmation Impérative 1"
c1.annee ← 2001
c1.matiere ← "informatique"
c1.nbInscrits ← 250
c1.formation ← "MIAS 1"

Dans la définition d'un type d'enregistrement, on peut introduire des membres de n'importe quel type, pourvu que ce type ait été déclaré au préalable.

2.2 Définition

2.2.1 Définition de type enregistrement en C

La syntaxe de la définition d'un type d'enregistrement (appelé `struct` pour structure en C) est la suivante :

```
typedef struct {
   <id_type> <id_membre1>;
   <id_type> <id_membre2>;
   ...
   <id_type> <id_membreK>;
} <id_type_struct>;
```

`<id_type_struct>` est l'identificateur du type d'enregistrement défini ci-dessus et qui comporte K membres. Les identificateurs des membres d'une même structure doivent être tous différents. L'identificateur `<id_type_struct>` permet de nommer ce type d'enregistrement dans la déclaration ultérieure de variables de ce type.

Exemple

```
typedef struct {
          char intitule[50];
      int annee;
      char matiere[50];
      int nbInscrits;
      char formation[50];
} typeCours ;
typeCours   cours1, cours2 ;
```

2.2.2 Définition de variables de type enregistrement en C

Les variables de type enregistrement se définissent comme à l'ordinaire :

- Si le type enregistrement a été défini au préalable (avec `typedef`) par
  `<id_type_struct> <id_variable1>,...<id_variableN>;`
- Si le type enregistrement n'a pas été défini au préalable, avec

```
struct {
   <id_type> <id_membre1>;
   <id_type> <id_membre2>;
   ...
   <id_type> <id_membreK>;
} <id_variable1>,...,<id_variableN>;
```

Il est possible également d'initialiser une variable de type `struct` lors de sa définition de la façon suivante :

```
<id_type_struct> <id_variable>={valeur1, valeur2,...,valeurK}
```

pourvu que les valeurs énumérées aient bien le type du membre de rang correspondant. On a alors le premier membre initialisé à la valeur1, le deuxième à la valeur2, ... etc., et le Kième à la valeurK.

2.3 Utilisation

2.3.1 Utilisation standard

Un membre d'un enregistrement est une variable (comme une cellule de tableau) à laquelle on peut affecter une valeur ou qui peut être évaluée dans une expression. Pour accéder à un membre d'une variable de type `struct` on utilise l'opérateur point noté « . » comme suit :

```
<id_var_struct>.<id_membre>
```

Exemple : Addition et multiplication de deux nombres complexes

```
#include <stdio.h>

typedef struct {
    float reel;
    float imaginaire;
} complexe;

main {
  complexe a,b,c;
  int i;

  printf("Entrez un premier nombre complexe \n") ;
  printf("partie réelle, puis partie imaginaire :");
  scanf("%f%f",&a.reel,&a.imaginaire);
  printf("Entrez un deuxième nombre complexe \n") ;
  printf("partie réelle, puis partie imaginaire :");
  scanf("%f%f",&b.reel,&b.imaginaire);
            /* addition de a et b */
  c.reel = a.reel + b.reel;
  c.imaginaire = a.imaginaire + b.imaginaire;
  printf("Leur somme : %f + i %f \n", c.reel,c.imaginaire);
            /* multiplication de a et b */
  c.reel = a.reel*b.reel - a.imaginaire* b.imaginaire;
  c.imaginaire=a.reel*b.imaginaire + a.imaginaire* b.reel;
  printf("le produit : %f + i %f \n", c.reel,c.imaginaire);
}
```

Remarque : on aurait pu aussi déclarer le type `complexe` à l'intérieur du bloc de définition de la fonction `main`, avant la déclaration des variables a, b et c. Mais il est plus clair de définir les types en tête de fichier de programme.

2.3.2 Affectation généralisée

On peut affecter le contenu d'une variable d'enregistrement à une autre variable d'enregistrement de même type. On parle alors d'affectation généralisée.

Exemple :

```
typedef struct {
      char intitule[50];
      int annee;
      char matiere[50];
      int nbInscrits;
      char formation[50];
} typeCours;

typeCours progImp1sem ={"Programmation impérative", 2001,
                        "Informatique", 250,"Mias1"};
typeCours progImp2sem;
```

progImp2sem = progImp1sem;

Cette instruction implique que :

```
{progImp2sem.intitule == "Programmation impérative"}
{progImp2sem.annee == 2001}
{progImp2sem.matiere == "Informatique"}
{progImp2sem.nbInscrits == 250}
{progImp2sem.formation == "Mias1"}
```

Cependant, on ne peut pas comparer directement deux structures. Pour tester si deux structures d'enregistrement ont des membres identiques, il faudra tester un à un l'égalité des membres des deux structures.

2.3.3 Structures et tableaux

Les opérateurs `.` et `[]` ont la même priorité (il s'agit des opérateurs les plus prioritaires). Cette priorité est supérieure à celle de l'opérateur d'adresse &. L'évaluation d'une expression faisant intervenir `.` et `[]` se fait (en l'absence de parenthèses) de gauche à droite.

a. Tableau d'enregistrement

Une variable de type tableau dont les cellules sont de type enregistrement se définit comme suit :

- Si le type enregistrement a été défini au préalable avec `typedef`

```
<id. type struct> <id. variable tableau>[TAILLE];
```

- Si le type enregistrement n'a pas été défini au préalable

```
struct {
   <id type> <id. membre1>;
   <id type> <id. membre2>;
   ...
   <id type> <id. membreK>;
} <id variable tableau>[TAILLE];
```

Exemple :

```
#include <stdio.h>
#define LONGMOT 80
#define NBMAX 25

typedef struct {
```

```
      char intitule[LONGMOT];
      int annee;
      char matiere[LONGMOT];
      int nbInscrits;
      char formation[LONGMOT];
} typeCours ;

int main() {

      typeCours deug[NBMAX];
      int i = 0;
      char rep;

      printf("Entrez les cours de DEUG\n");
      do {
          printf("Cours %d :\n", i);
          printf("Intitulé :");
          scanf("%s",deug[i].intitule);
          printf("\nAnnee :");
          scanf("%d",&deug[i].annee);
          printf("\nMatiere :");
          scanf("%s",deug[i].matiere);
          printf("\nNombre d'inscrits :");
          scanf("%d", &deug[i].nbInscrits);
          printf("\nFormation :");
          scanf("%s", deug[i].formation);
          printf("Voulez-vous continuer ?(o/n)");
          scanf("%c",&rep);
          i=i+1;
      } while((rep=='o') && (i < NBMAX ));
}
```

b. Accès aux cellules des membres de type tableau

Pour accéder au premier caractère du membre `formation` d'une variable de type `typeCours` on écrira :
`<id_variable>.formation[0]` (il s'agit d'une variable de type char)

Si la variable de type `typeCours` qui nous intéresse est la ième variable du tableau `deug`, on écrira :
`deug[i].formation[0]`

3 Fonctions et procédures

La programmation modulaire vise à répartir les traitements constituant un programme complexe en sous-programmes effectuant des traitements plus élémentaires. On peut alors, comme dans un jeu de construction assembler les actions simples pour élaborer des actions plus complexes. Cette méthode a le mérite de la lisibilité et elle permet de factoriser le code.

Un sous-programme ne fonctionne pas de manière autonome : il est appelé par un autre programme. C'est une séquence d'instructions qui effectue une tâche donnée. Il a des arguments d'entrée -- qu'on appelle *paramètres*.

Traditionnellement, on distingue en programmation impérative deux types de sous-programmes : les fonctions et les procédures. Les fonctions sont des sous-programmes qui calculent une valeur de sortie (on parle de la *valeur de retour*) alors que les procédures n'ont pas de valeur de retour. On utilise les fonctions pour calculer une valeur en fonction de certains paramètres (par exemple somme (x,y) calculera la somme des deux paramètres donnés en arguments) ; les procédures étant elle simplement utilisées comme un bloc d'instructions nommé. Le fait d'avoir donner un nom au bloc d'instructions d'une procédure permet de réutiliser ce bloc sans avoir à le taper plusieurs fois dans le programme.

Nous avons déjà utilisé des fonctions (racine carrée, valeur absolue) et des procédures (lire, écrire).

Exemple

Dans cet exemple qui affiche la valeur absolue de l'entier entré par l'utilisateur, on trouve une fonction, *valeur absolue* (notée `abs`, en C) et des procédures, *écrire* et *lire*. On remarque que la fonction est utilisée dans une expression : la valeur retournée par la fonction (ou résultat) est alors affectée à la variable *res*. Les procédures forment des groupes d'instructions en tant que telles.

Algorithme
    Valeur d'entrée entière : e
    Variable de sortie : res
début
    écrire(« Entrez un entier signé : »)
    lire(e)
    res <- valeur absolue(e)
    écrire (« Valeur absolue : », res)
fin algorithme

```
#include <stdio.h>   /* librairie d'entrée/sortie */
#include <stdlib.h>  /* librairie standard */

int main () {
 /* Affichage de la valeur absolue d'un entier */

 signed int ent ;    /* variable d'entrée */
 int res ;           /* variable sortie */

 printf("Entrez un entier signé : ") ;
 scanf("%d", &ent);
 res  =    abs(ent) ;
```

```
 printf("\nValeur absolue : %d\n", res) ;
}
```

Le programmeur dispose en effet de bibliothèques définissant les procédures et fonctions usuelles. Il peut également définir ses propres procédures ou fonctions. Ces sous-programmes permettent de modulariser les programmes, d'éviter les répétitions en nommant les blocs d'instructions correspondant à des sous-tâches clairement identifiées. Ils facilitent l'écriture, la lisibilité et la maintenance des programmes complexes.

3.1 Fonctions

Une fonction est un sous-programme qui retourne une valeur.

Dans l'exemple ci-dessus, la fonction *valeur absolue* prend en argument (ou paramètre) une valeur entière, et retourne une valeur qui est la valeur absolue de l'argument. Cette valeur peut alors être utilisée comme n'importe quelle valeur. Une fonction suivie d'arguments est évaluée comme une expression : elle peut faire partie d'une expression, ou figurer comme argument d'une procédure ou fonction, comme *écrire*.

Exemple

On aurait pu remplacer (avec le même résultat à l'affichage)

res <- valeur absolue(e)
écrire (« Valeur absolue : »,res)

par

écrire (« Valeur absolue : », valeur absolue(e))

soit en C

```
printf("\nValeur absolue : %d", abs(e)) ;
```

3.1.1 Exemple de définition et d'utilisation de fonction

On peut par exemple définir une fonction qui calcule la factorielle d'un entier :

Algorithme liste de factorielles

1 variable d'entrée entière globale : e

fontion valide(entier p) : booléen
début
si p≠0 et p<50
alors valide retourne vrai
sinon valide retourne faux
finsi
fin fonction

fonction fact(entier p) :entier
début
2 variables auxiliaires entières : i et res
res<-1
pour i variant de 2 à p faire
*res<-i*res*

```
                fin pour
                fact retourne res
        fin fonction
fonction principale
début
        début
                écrire(« Entrez un entier (zéro pour quitter) : »)
                lire(e) ;
                si valide(e)
                        alors écrire(« Factorielle de », e, « = »,fact(e))
                fin si
        tant que e≠0
        fin tant que
fin fonction principale
fin algorithme
```

On constate que cet algorithme comporte 3 parties :

- La définition d'une variable ;
- La définition de deux fonctions ;
- Le corps de la fonction principale (ou corps de l'algorithme).

La fonction principale fait ici appel aux deux fonctions précédentes. La fonction *valide* est une fonction booléenne qui retourne vrai si la valeur passée en paramètre est valide et faux dans le cas contraire. La fonction *fact* est une fonction entière qui calcule la factorielle de son argument (la valeur passée en paramètre).

3.1.2 Définition d'une fonction

Une définition de fonction comporte dans l'ordre : 1) un en-tête, 2) éventuellement une partie définition pour introduire les constantes, types et variables utilisés dans le corps même de la fonction et 3) une série d'instructions, le corps de la fonction.

a. En-tête de fonction

*fonction <nom de fonction>**(**<liste des paramètres>**):**<type du résultat>*

Exemple

fonction valide(entier p) : booléen

Un en-tête de fonction obéit à la syntaxe ci-dessus. Il comporte :

- Le nom de la fonction : un identificateur qui sert à appeler la fonction. Dans l'exemple ci-dessus, le nom de la fonction est *valide*.
- Entre parenthèses et séparés par des virgules, la liste des paramètres, ou arguments, de la fonction. La déclaration de la fonction fixe leur nombre, leurs noms et leurs types. Le type de ces paramètres peut être simple ou structuré, prédéfini ou défini par l'utilisateur. Dans

ce dernier cas, la définition de type doit être faite au préalable. Cette liste des paramètres suit la syntaxe suivante :

<type 1> <par 1>, <type 2> <par 2>,..., <type N> <par N>

Dans l'exemple ci-dessus, la fonction *valide* n'a qu'un seul paramètre *p* de type entier.

- Le type du résultat, c'est-à-dire de la valeur retournée par la fonction. Le type du résultat est indépendant du type des arguments. Dans l'exemple de *valide*, le résultat de la fonction est de type booléen. On parle dans ce cas de fonction booléenne. Dans le cas de *fact*, la valeur de retour est entière. Dans la plupart des langages de programmation impérative, une fonction ne peut retourner qu'un seul résultat. Nous respecterons cette contrainte ici.

b. Partie définition

On peut avoir besoin pour calculer le résultat d'une fonction d'utiliser des variables. C'est le cas des variables *i* et *res* de la fonction *fact* ci-dessus. On déclare alors ces variables, après l'en-tête de la fonction. On peut de la même manière déclarer des constantes ou des types, et même d'autres sous-programmes.

Portée des variables : variables locales vs *variables globales*

L'ensemble des variables qui sont déclarées à l'intérieur d'une fonction sont des variables locales à cette fonction, c'est-à-dire qu'elles ne sont définies et ne peuvent être utilisées que dans le corps de cette fonction. On ne peut pas les utiliser dans les instructions qui composent par exemple le corps de l'algorithme de la fonction principale : elles n'y sont pas définies, parce qu'elles ne sont visibles que par les instructions du bloc de définition de la fonction où elles ont été déclarées.

Exemple

Dans l'algorithme *liste de factorielles*, les variables *i* et *res* sont des variables locales de la fonction *fact*. On ne pourrait pas utiliser la variable *i* dans le corps de l'algorithme principal.

En revanche, les variables déclarées en tête de l'algorithme sont des variables globales. Elles sont accessibles par les instructions du bloc de l'algorithme et par les instructions des fonctions définies dans l'algorithme.

Exemple

Dans l'algorithme *liste de factorielles*, la variable *e* est déclarée au niveau de l'algorithme et non à l'intérieur d'une définition de fonction comme *i* et *res*. C'est une variable globale. Elle est utilisée dans la fonction principale mais pourrait aussi être utilisée dans le corps de définition de la fonction *fact*.

Il est possible de faire appel à plusieurs variables de même identificateur dans des blocs de définition différents. Ces variables auront alors des adresses différentes. Cela n'est cependant pas recommandé. Si par inadvertance, on oublie de définir une variable `i` à l'intérieur d'une fonction et qu'une autre variable `i` a été définie au niveau de l'algorithme, c'est la variable globale qui sera utilisée dans la fonction et sa modification pourrait avoir des conséquences imprévues.

Imbrication de fonctions

Les fonctions peuvent être vues comme des variables particulières et on peut généraliser cette notion de visibilité (ou portée) de variable aux fonctions : on peut en effet déclarer des fonctions à l'intérieur d'autres fonctions. Une variable est visible dans le bloc où elle est déclarée et dans tous les blocs enchâssés dans ce bloc. Elle n'est pas visible par les blocs de niveaux supérieurs. Une fonction aura de même des niveaux de visibilité relatifs au niveau où elle aura été définie.

Exemple

Dans le schéma d'algorithme suivant où les fonctions A et B sont déclarées directement dans l'algorithme principal 1, et où la fonction C est enchâssée dans la fonction B, on a indiqué les variables qui sont accessibles et les fonctions qui peuvent être appelées dans chaque bloc d'instructions.

Algorithme 1
Définition de la variable globale m

fonction A
Définition des variables locales a et x
fonction B
Définition des variables locales b et x
début
corps B variables accessibles : m (globale) et a
variables locales : b et x *(x de B)*
fonction accessible : B
fin fonction B

début
corps A variable globale : m
variables locales : a et x *(x de A)*
fonctions accessibles : B et A
fin fonction A

fonction C
Définition de la variable c
début
corps C variable globale : m
variables locales : c
fonctions accessibles : A
fin fonction C

fonction principale
début
corps 1 variable globale : m
fonctions accessibles : A et C
fin fonction principale
fin algorithme

Les imbrications de fonctions ne sont pas autorisées dans tous les langages de programmation, mais elles améliorent la modularité des programmes.

c. Corps de fonction

Le corps de la fonction contient l'ensemble des instructions nécessaires au calcul du résultat à partir des valeurs données en entrée, c'est-à-dire des arguments de la fonction. Ces instructions sont écrites exactement de la même manière dans un algorithme principal et dans un sous-programme.

Le corps d'une fonction, comme celui d'un algorithme, est introduit par le mot clé *début* et terminé par le mot clé *fin*.

Instruction de retour

Le corps d'une fonction doit comporter une instruction de retour qui indique quelle est la valeur retournée (ou résultat) de la fonction. Cette valeur doit avoir le type du résultat indiqué dans l'en-tête de la fonction (*booléen* et *entier* dans le cas de *valide* et *fact*, respectivement). Cette instruction de retour a la syntaxe suivante :

<nom de la fonction> retourne <expression>

L'instruction de retour figure généralement à la fin du code de la fonction. Toute exécution de fonction doit comporter une instruction de retour. Il faut veiller à ce point dans le cas de boucles ou de branchements conditionnels. Dans l'exemple de la fonction *valide*, chacune des deux branches du *si* comporte une instruction de retour : *valide retourne vrai* et *valide retourne faux*.

Effet de bord

L'effet et l'objectif premier d'une fonction est le calcul d'une valeur et le retour de cette valeur dans l'algorithme appelant. Ce calcul dépend des arguments de la fonction et des variables globales auxquelles la fonction fait appel.

Lorsque la fonction modifie des variables définies à l'extérieur du corps de la fonction (elle fait alors autre chose que de simplement calculer et retourner une valeur), on dit qu'elle a des effets de bords. Les effets de bord sont des effets secondaires par rapport au calcul du résultat.

Exemple

```
fonction valide(entier p) : booléen
début
        si p≠0 et p<50
                alors valide retourne vrai
        sinon   écrire(« La valeur entrée n'est pas valide »)
                e<-0
                valide retourne faux
        finsi
fin fonction
```

Cette nouvelle version de la fonction *valide* présente deux effets de bord. Le premier est relativement anecdotique : c'est une instruction d'écriture qui affiche un message à l'écran. Le deuxième est la modification de la variable globale *e*. Dans le cas où la valeur donnée par l'utilisateur n'est pas valide, la fonction a donc deux effets en plus du calcul de la valeur de retour (faux, dans ce cas) : l'affichage d'un message à l'écran et la modification de la variable e, ce qui provoque la sortie de boucle dans la fonction principale.

Les deux grands types d'effets de bord sont les instructions d'écriture et de lecture d'une part et la modification des variables globales d'autre part.

Les effets de bord rendent l'algorithme plus complexe à maîtriser parce qu'ils compliquent l'interface de la fonction avec l'algorithme appelant. Pourtant les effets de bord permettent des astuces de programmation et sont parfois très utiles. On veillera à les utiliser avec parcimonie, à éviter tout effet de bord accidentel et à l'inverse à documenter les effets de bord voulus d'une fonction. Le commentaire d'une fonction doit ainsi décrire, outre la description de sa fonction principale, l'ensemble de ses effets de bord si elle en a.

3.1.3 Appel de fonction

a. Syntaxe d'un appel de fonction

On appelle une fonction utilisateur exactement comme on appelle une fonction prédéfinie du langage considéré. La syntaxe est la suivante :

*<nom de fonction>**(**<par 1>, <par 2>,...,<par n>**)***

Dans une instruction, on utilise une fonction comme une expression. Comme toute expression, cette fonction est évaluée au moment de son utilisation. Le résultat de cette évaluation est la valeur retournée par la fonction.

On peut ainsi utiliser une fonction dans une expression complexe ou dans une instruction de lecture.

Exemple

n <- fact(6) % (4)
écrire(« La factorielle de », e, « est », fact(e))

Lors de l'appel d'une fonction, on indique entre parenthèses ses arguments, des expressions – éventuellement complexes – qui sont évaluées au moment de l'appel. Le nombre et le type de ces arguments doivent correspondre à la liste des paramètres figurant dans l'en-tête de la fonction.

On dit que les arguments avec lesquels la fonction est appelée sont ses *paramètres effectifs*. On les distingue des *paramètres formels* qui figurent dans la définition (en-tête et corps) de la fonction. Ils ne sont pas de même nature : les paramètres formels sont des variables, alors que les paramètres effectifs peuvent être des expressions plus complexes qui retournent une valeur.

Dans l'exemple de l'algorithme *liste de factorielles*, p est le paramètre formel des deux fonctions *valide* et *fact* (ils pourraient avoir des noms différents cela ne changerait rien, ces deux noms sont en réalité complètement indépendants). C'est l'expression *e* qui est le paramètre effectif au moment de l'appel.

b. Passage des paramètres par valeur

Aux valeurs des paramètres effectifs de la fonction correspondent donc (dans le même ordre) les paramètres formels figurant dans l'en-tête de la fonction.

Lors de l'exécution de la fonction, chaque paramètre formel prend la valeur du paramètre effectif qui lui correspond. Les instructions de la fonction et le calcul de sa valeur de retour sont ensuite exécutés avec ces valeurs initiales des paramètres formels. On dit dans ce cas que les arguments ont été passés *par valeur*, car c'est la valeur des paramètres effectifs qui est transmise aux paramètres formels.

Ainsi, la valeur du paramètre effectif est recopiée dans le paramètre formel. Tout se passe comme si le paramètre formel était une variable locale de la fonction et que, à l'appel de la fonction, on avait une série d'affectations du type :

<i ème paramètre formel> <- <i ème paramètre effectif> ;

Si la valeur du paramètre formel est modifiée dans le corps de la fonction, cette nouvelle valeur n'affectera pas le paramètre effectif (s'il s'agissait d'un identificateur de variable) et ne sera plus accessible après l'exécution de la fonction. Le paramètre effectif n'est en effet pas modifié lors d'un passage de paramètre par valeur. Il ne sert qu'à fournir la valeur initiale du paramètre formel avant l'exécution de la fonction et aura donc encore a priori cette valeur à la fin de l'exécution de la fonction (à moins que l'on ait un effet de bord dans le corps de la fonction).

Exemple

Voici un algorithme qui calcule la puissance n ième d'un entier.

```
Algorithme calcul

1 variable d'entrée réelle : r
1 variable d'entrée entière : p

fonction puissance(réel base; entier puis):réel;
{calcule la puissance puis de la base.  puis est un entier quelconque,
 s'il est négatif, on utilise  sa valeur absolue}
        1 variable entière auxiliaire : i
        1 variable auxiliaire réelle :  res
début
        si (puis=0)
           alors res<-1
        sinon   si (puis<0) alors puis<-abs(puis)
                finsi
                res<-base
                pour i variant de 2 à puis faire
                      res<-res*base
                fin pour
                puissance retourne res
        finsi
fin fonction puissance

fonction principale
début
        écrire(« Donnez un nombre : »)
        lire(r);
        écrire(« Quelle puissance de », r, « voulez-vous ? »)
        lire(p);
        écrire(« La puissance »,p, « de », r, « est »,puissance(r,p))
fin fonction principale
fin  algorithme
```

3.1.4 Au-delà de la valeur de retour d'une fonction

Considérons une fonction f appelée dans le bloc d'instructions qui compose le corps de la fonction F. Nous dirons que *F* est la fonction *appelante* et *f* la fonction *appelée*. Nous avons vu qu'une fonction est un sous-programme qui opère un calcul à partir des valeurs de ses paramètres et renvoie le résultat de ce calcul sous la forme d'une valeur de retour unique, participant au calcul d'une instruction de la fonction appelante. On a donc une interface claire entre la fonction appelée et la fonction appelante : en entrée de la fonction, plusieurs valeurs transmises par les différents paramètres calculés depuis la fonction appelante et, en sortie de la fonction une valeur unique (le résultat de la fonction appelée) qui s'intègre dans le calcul d'une expression de la fonction appelante.

Dans certains cas, ces règles sont trop contraignantes et l'on souhaiterait qu'une fonction puisse transmettre plus d'informations que le simple retour d'une valeur dans la fonction appelante.

Exemple

On pourrait par exemple vouloir écrire une fonction qui prend un ensemble de valeurs en arguments et qui retourne à la fois le minimum et le maximum de ces valeurs plutôt que d'écrire deux fonctions pour calculer l'un et l'autre. En effet, cela éviterait de parcourir deux fois l'ensemble des valeurs, pour trouver, une première fois, le minimum et une deuxième fois, le maximum.

On peut cependant contourner cette contrainte de deux manières : en jouant sur les effets de bords, ou en jouant sur le mode de passage des arguments (voir plus loin).

La première méthode consiste à utiliser des variables globales que la fonction modifie. On utilise ainsi un effet de bord.

Exemple

Dans l'exemple ci-dessous, la fonction *minMax* est définie comme une fonction booléenne. Elle retourne la valeur vraie si les valeurs passées en paramètres ne sont pas toutes nulles. Elle retourne la valeur faux dans le cas contraire. Mais la fonction ne se contente pas de calculer cette valeur de retour. Elle a aussi comme effet d'initialiser les variables globales *valMin* et *valMax* dans le cas où la valeur de retour serait vrai.

Algorithme min-max

```
3 variables d'entrée entières :e1, e2, e3 ;
2 variables de sortie réelles : valMin et valMax
1 variable auxiliaire booléenne : b

fonction minMax(3 entiers p1, p2, p3):booléen
début
        valMin<-p1
        valMax<-p1
        si p1=0 et p2=0 et p3=0
            alors minMax retourne faux
        sinon
            si p2<valMin alors valMin<-p2
            si p2> valMax alors valMax<-p2
            si p3<valMin alors valMin<-p3
```

```
            si p3> valMax alors valMax<-p3
            minMax retourne vrai
        finsi
fin fonction minMax

fonction principale
début
        écrire(« Entrez 3 entiers : »)
        lire(e1, e2, e3)

        b<-minMax(e1, e2, e3)
        si b est vrai
            alors écrire(« Minimum : », valMin, «. Maximum : »,valMax)
        sinon écrire(« Valeurs nulles »)
        finsi
fin fonction principale
fin algorithme
```

3.1.5 Simulation des fonctions

La simulation des fonctions que nous utilisons ici évalue lors de l'appel de la fonction ses paramètres effectifs pour initialiser ses paramètres formels. À la fin de la fonction, elle indique quelle est la valeur retournée par la fonction, évalue ses paramètres formels et réévalue les paramètres effectifs. On pourra ainsi constater, si les paramètres ont été modifiés, l'écart entre les uns et les autres.

Simulation de l'appel d'une fonction

<Simulation du début du programme appelant>
Appel de la fonction <nom de fonction>(<eval par. eff.>) <eval par. for.>
<simulation du corps de la fonction avec les paramètres formels >
Retourne la valeur <valeur>
fin fonction <eval. par. for.> <eval. par. eff.>
<Simulation de la suite du programme appelant>

Où <eval par. eff.> <eval par. for.> sont les listes des évaluations des paramètres effectifs et formels respectivement.

Simulation de `calcul`

Ecriture de «Donnez un nombre : »
Lecture de 5 {`r`=5}
Ecriture de «Quelle puissance de 5 voulez-vous ? »
Lecture de -3 {`p` =-3}
Appel de la fonction puissance{`r`=5, `p`=-3} {`base`=5; `puis`=-3}
{(`puis`=0)=faux}
{(`puis`<0)=vrai; `puis`=3}
{`res`=5}
Début-for{`i`=2, puis=3, (`i`<=`puis`)=vrai }
{`res`=5*5=25}
Suite-for {`i`=3, puis=3, (`i`<=`puis`)=vrai }
{ `res`=25*5=125}
Suite-for {`i`=4, puis=3, (`i`<=`puis`)=faux}
Fin-for

Retourne la valeur 125}
fin fonction puissance{base=5; puis=3}{r=5; p=-3}
Ecriture de «La puissance -3 de 5 est 125. »

3.2 Procédures

Les procédures constituent le deuxième type de sous-programmes. Elles se distinguent des fonctions en ce qu'elles ne retournent pas de valeur au programme appelant. Une procédure se contente de regrouper une série d'instructions qui peuvent être considérées comme formant un ensemble cohérent.

On a déjà vu avec les instructions de lecture et d'écriture avec les procédures prédéfinies *lire* et *écrire*. Les appels de procédures sont de manière générale des instructions.

De même qu'on peut définir de nouvelles fonctions pour répondre à des besoins spécifiques, on peut écrire de nouvelles procédures pour rendre un programme plus modulaire et éviter des répétitions de code.

3.2.1 Exemple d'utilisation de procédures

Dans l'exemple suivant, on se propose d'imprimer les solutions réelles d'une équation du second degré : $a\,x^2 + b\,x + c$. Le programme demande d'entrer les coefficients a, b et c, calcule le discriminant $b^2 - 4ac$, et imprime les solutions en fonction du discriminant.

```
Algorithme Résolution
{ Résolution dans R d'une équation du second degré }

3 variables d'entrée entières : a1, a2, a3   {les coefficients}

procédure entrezCoeff()
{ Procédure qui permet d'entrer les coefficients a1, a2 et a3 }
début
    écrire(« entrez les coefficents a, b et c »)
    écrire(« de l'équation  à résoudre :»)
    lire(a1, a2, a3)
fin procédure entrezCoeff

fonction discriminant(entier a ; entier b ; entier c) : réel ;
{ fonction calculant le discriminant }
1 variable entière réelle :  res
début
    res <- b*b  -  4*a*c
    discriminant retourne res
fin fonction discriminant

procédure imrimeSolution(réel  dis)
début
    si dis<0
       alors  écrire(« Pas de solution réelle »)
    sinon
       si dis=0
          alors écrire(« une seule solution : », - a2 / 2*a1)
       sinon
          écrire(« deux solutions : », - (a2 + racine(dis)) / 2*a1 )
```

```
            écrire( « et », racine(dis) – a2 / 2*a1  )
        finsi
    finsi
fin procédure imrimeSolution

fonction principale
1 variableréelle auxiliaire : delta      {le discriminant}
début
    entrezCoeff()
    delta <- discriminant(a1,a2,a3)
    imrimeSolution(delta)
fin fonction principale
fin algorithme
```

On constate que le corps de la fonction principale est très concis. Une part importante du code est reportée dans les sous-programmes. C'est l'un des mérites de la programmation modulaire.

3.2.2 Définition d'une procédure utilisateur

a. Définition de procédure

On voit sur l'exemple ci-dessus qu'on définit une procédure comme on définit une fonction, à cette différence qu'on n'indique pas le type du résultat puisqu'une procédure ne retourne pas de valeur. Une définition de procédure indique donc :

- Le mot clé *procedure* ;
- Le nom de la procédure ;
- Les paramètres formels éventuels ;
- Le type des paramètres formels ;

Il faut veiller à définir les procédures et fonctions dans un ordre qui assure que toute fonction ou procédure *f* appelée dans une fonction ou procédure *g* est définie avant cette dernière. On peut par ailleurs définir une procédure ou une fonction *f* à l'intérieur d'une autre procédure ou fonction *g* : *f* est alors locale à *g* et ne peut être utilisée que dans *g*.

Comme pour les fonctions, il importe de bien commenter les procédures, leur objet, le rôle de leurs différents paramètres ainsi que leurs effets de bord (il y en a forcément, car sinon, la procédure ne ferait rien !).

b. Corps d'une procédure

On écrit une procédure exactement comme on écrit une fonction, à cette différence qu'une procédure ne comporte pas d'instruction de retour.

Le principe de la portée des variables et celui du passage des paramètres est identique pour les procédures et les fonctions.

3.2.3 Appel de procédure

L'appel d'une procédure définie par l'utilisateur se fait comme l'appel d'une procédure prédéfinie (ex. *écrire*, *lire*). Un appel de procédure est une instruction qui peut être insérée dans le corps d'un algorithme comme n'importe quelle instruction.

Comme pour les fonctions, le nom des arguments lors de l'appel est indépendant des identificateurs des paramètres formels avec lesquels la procédure a été définie. Les arguments d'appel (paramètres effectifs) ne sont d'ailleurs pas nécessairement des noms : ce sont des expressions qui retournent une valeur, et ils ne correspondent pas toujours à un identificateur de variable. Mais le nombre et le type des valeurs des paramètres effectifs doivent correspondre à ceux des paramètres formels.

3.3 Les fonctions en C

Le langage C ne fait pas la distinction entre les procédures et les fonctions comme d'autres langages de programmation. En C, tous les sous-programmes sont des fonctions. Une procédure est une fonction qui retourne une valeur vide, dont le type est noté `void`. En revanche, il y a deux manières d'utiliser les fonctions en C : une utilisation fonctionnelle ou procédurale selon qu'on utilise ou non la valeur de retour de la fonction.

3.3.1 Utilisation fonctionnelle des fonctions

Traduisons en C l'algorithme *liste de factorielles*.

Exemple

```
# include <stdio.h>
# define TAILLEMAX 50

typedef enum booleen {faux,vrai};

/* définition de fonctions */
enum booleen valide(int p) {
/* teste si la valeur p est valide ou non */
      if (p > 0 && p < TAILLEMAX)
        return(vrai);
      else
        return(faux);
}

int fact(int p) {
/* calcule factorielle de p */
      int i, res;
      res=1;
      for (i=2; i<=p; i=i+1)
          res= res*i;
      return(res);
}

int main () {
      int e;

      do {
          printf("Entrez un entier (zéro pour quitter): ");
```

```
            scanf("%d", &e);
            if (valide(e) == vrai)
              printf("\nFactorielle de %d = %d\n", e, fact(e));
        } while (e!=0);
}
```

a. Syntaxe

La syntaxe d'une définition de fonction est la suivante

```
<type résultat> <nom de fonction>(<type1> <par1>, ... ,
                                          <typeN><parN>) {
<bloc d'instructions>
}
```

Cette syntaxe est celle qui est définie par la norme ANSI (1988). La syntaxe antérieure, à la norme ANSI, d'une définition de fonctions était :

```
[<type résultat>] <nom de fonction>(<par1>, ...<parN>)
     <type1> <par1> ;
...
 <typeN><parN> ;
{
<bloc d'instructions>
}
```

La liste des paramètres formels figure aussi entre parenthèses mais la liste de leurs déclarations (avec leurs types) apparaît entre l'en-tête de la fonction et le début du bloc d'instruction. Cette ancienne syntaxe est encore reconnue de certains compilateurs.

b. Paramètres

La liste des paramètres notée entre parenthèses détermine le nombre des paramètres formels de la fonction, leur nom et leur type. Une fonction peut avoir un nombre quelconque de paramètres.

Exemple

La fonction `valide` admet par exemple un unique paramètre.
La fonction `somme` ci-dessous additionne deux réels et prend deux arguments :

```
float somme(float arg1, float arg2) {
          return(arg1 + arg2) ;
}
```

On peut dans certains cas avoir besoin de fonction sans paramètre. Dans ce cas, le C suit la syntaxe suivante :

```
<type résultat> <nom fonction> (void) {
            <bloc d'instructions>
}
```

`void` est un type prédéfini qui désigne l'ensemble vide.

Exemple

La fonction `chiffreAleatoire` n'a pas de paramètre. Elle choisit de manière aléatoire un entier compris entre 0 et 9. Pour ce faire, elle fait appel à une fonction qu'on suppose définie `heureEnSeconde` qui n'a elle-même aucun paramètre. `heureEnSeconde` récupère l'heure

du processeur, calcule le nombre de secondes écoulées depuis le début de la dernière heure et retourne cette valeur (un entier compris entre 0 et 3600). La fonction `chiffreAleatoire` retourne le chiffre des unités de cette valeur arbitraire de secondes.

```
int chiffreAleatoire(void) {
          int s ;
          s=heureEnSeconde() ;
          return(s%10) ;
}
```

Il existe enfin un type assez particulier de fonctions qui admettent un nombre variable de paramètres. C'est le cas, en particulier de la fonction `printf` qui prend n+1 paramètres, selon le nombre n de spécifications de format que comporte la chaîne de contrôle constituant le premier argument. Nous ne développons pas cet aspect ici.

c. Valeur de retour et type de fonction

Le type d'une fonction est celui de sa valeur de retour. Une fonction peut être de n'importe quel type (entier, réel, caractère, etc.). Dans les exemples précédents, les fonctions `fact` et `chiffreAleatoire` sont de type `int`. La fonction `valide` est de type `booleen` où `booleen` est un type énuméré défini par l'utilisateur.

Quand aucun type n'est spécifié dans la déclaration, une fonction est, par défaut, de type `int`. Mais nous préférons déclarer toujours explicitement le type de retour.

Une fonction correctement définie en C doit comporter une instruction de retour dont la syntaxe est
`return(<expression>) ;`

Le type de l'expression doit correspondre au type avec lequel la fonction a été définie. Cette instruction permet de quitter le corps de définition de la fonction et retourne dans la fonction appelante au point d'évaluation de l'expression qui contenait l'appel à la fonction, avec cette valeur de retour.

Au cas où cette instruction serait manquante, ce que nous déconseillons formellement, la valeur de retour sera la valeur calculée par la dernière instruction exécutée par le corps de la fonction.

d. Effets de bord

Le corps d'une fonction a donc pour premier effet de calculer une valeur qui sera retournée au programme appelant. Cependant, une fonction peut effectuer d'autres actions et avoir des effets de bord.

Exemple

```
int e;  /* variable globale */


enum booleen valideAvecEffetsBord(int p) {
/* valideAvecEffetsBord teste si la valeur passée */
/* en paramètre est valide ou non. Si la valeur */
/* n'est pas valide, un message est adressé à */
/* l'utilisateur et la valeur est remplacée par 0 */
```

```
/* pour sortir du programme. */

        if (p!=0 && p<50)
          return(vrai);
        else {
          if (p!=0) {
                printf("Valeur non valide");
                e=0;
          }
          return(faux);
        }
}
```

e. Appel de fonction

Un appel de fonction s'utilise comme n'importe quelle expression. Les exemples suivants montrent qu'un appel de fonction peut figurer dans une expression comme argument d'une autre fonction, notamment d'un `printf`.

Exemples

```
comb= fact(n)/(fact(n-m)*fact(m)) ;
for (i=1 ; i<fact(n) ; i=i+1) {
          ...
}
```

Dans les exemples ci-dessus on trouve différents appels de fonction :

```
if (valide(e)==vrai) {...}
printf("Factorielle de %d = %d.\n", e, fact(e));
s=heureEnSeconde() ;
```

L'appel d'une fonction sans paramètre respecte la syntaxe suivante :

```
<nom de fonction> ()
```

Un exemple figure dans la fonction `chiffreAleatoire` qui appelle la fonction `heureEnSeconde`.

f. La fonction `main`

Un programme en C est constitué d'un ensemble de définitions de constantes, de types et de variables et d'un ensemble de définitions de fonctions. Il comporte nécessairement une fonction principale appelée `main` dont le corps constitue, comme nous l'avons déjà vu, le corps du programme.

Jusqu'ici, les fonctions `main` de nos programmes avaient la forme suivante :

```
int main() {
...
}
```

La syntaxe de la définition de la fonction `main` suit celle des définitions de fonction. La fonction `main` ne comporte ici aucun paramètre. L'en-tête indique le type `int` de la fonction. Dans les programmes que nous allons écrire maintenant la valeur retournée sera 0 pour indiquer que le programme s'est exécuté normalement. Traditionnellement, on considère qu'une valeur de retour négative signale une erreur dans l'exécution du programme.

La valeur de retour d'un programme peut en effet être utilisée par l'interprète de commande de l'ordinateur. Un programme, une fois compilé, est une commande comme une autre, dont l'exécution peut être lancée par l'utilisateur. Si l'utilisateur sait comment créer des commandes complexes, un appel à son programme pourra figurer dans une telle commande, et sa valeur de retour pourrait alors être utilisée. Ce cas de figure est cependant relativement rare.

Par contre, il n'est pas rare de vouloir passer des arguments à un programme, comme par exemple, un nom de police de caractères , un nom de fichier, ou autres. Nous verrons plus loin comment déclarer des arguments à un programme.

3.3.2 Utilisation procédurale des fonctions

On utilise parfois en C des fonctions comme des procédures. Cela signifie qu'on fait appel à des fonctions mais qu'on ne les utilise pas pour leur valeur de retour, mais comme des blocs d'instructions ayant des effets de bord.

Voici en C un exemple de calcul de moyennes de notes d'étudiants dans deux matières différentes : l'informatique et les maths. La procédure afficheMoyenne permet d'afficher leur moyenne dans ces deux matières. Un tableau de dimension Nx3 permet de stocker, dans la cellule i,0 le numéro d'inscription d'un étudiant, et dans les cellules i,1 et i,2, ses notes dans les deux matières.

```
# include <stdio.h>
#define TAILLEMAX 100

float tableau[TAILLEMAX][3] ; /* variable d'entrée */

void afficheMoyenne(int e) {
      /* procédure qui affiche la moyenne de l'étudiant
          dont le numéro d'inscription est e */

          float m ;
          int num;

          m=(tableau[e][1]+tableau[e][2])/2;
          num=(int)tableau[e][1];
          printf("\tEtudiant %d : %f \n", num, m);
}

int main () {
        int taille ; /*taille effective du tableau*/
        int i ;

              /* initialisation du tableau */
        printf("Combien y a-t-il d'étudiants ? ") ;
        scanf("%d", &taille);
        if (taille > TAILLEMAX) {
          printf("Nombre trop grand\n") ;
          return(0) ;
        }

        for (i=0 ; i<taille ; i=i+1) {
          printf("Entrez le numéro du %d ème étudiant : ",i+1);
          scanf("%f", &tableau[i][0]);
```

```
            printf("\nEntrez sa note  d'informatique : ");
            scanf("%f", &tableau[i][1]);
            printf("\nEntrez sa note de math : ");
            scanf("%f", &tableau[i][2]);
          }
               /* Affichage des moyennes */
          printf("Liste des moyennes :\n");
          for (i=0 ; i<taille ; i=i+1)
            afficheMoyenne(i) ;
          return(0);
}     /* fin de définition de main */
```

a. Syntaxe

Le C n'ayant pas de procédure, la syntaxe de la définition d'une procédure est celle d'une définition de fonction dont la valeur de retour aurait le type `void` :

```
void <nom de fonction>(<type1> <par1>, ...
                                        <typeN><parN>) {
      <bloc d'instructions>
}
```

Une procédure peut avoir un nombre quelconque de paramètres.

b. Type `void`

Comme une procédure ne retourne aucune valeur au programme appelant, on lui associe le type `void` qui est le type « sans valeur ». On obtient alors une définition de fonction comme celle de `afficheMoyenne`.

Définir une fonction comme étant de type `void` a le mérite d'indiquer explicitement dès sa définition qu'elle sera utilisée comme procédure.

En théorie, il ne devrait pas être nécessaire d'indiquer une valeur de retour dans les procédures puisque cette valeur n'est pas utilisée. En pratique, de nombreux compilateurs émettent un signal d'alerte quand ils rencontrent une fonction sans instruction de retour (dans ce cas on mettra `return();`)

c. Appel des procédures

Le C ne comportant pas de procédure, la syntaxe de la définition des fonctions et des procédures est identique.

Nous l'avons vu au niveau algorithmique, un appel de procédure constitue une instruction en tant que telle. La syntaxe d'un appel de procédure est donc :

```
<nom de procédure>(<liste des paramètres effectifs>) ;
```

où le « ; » est la marque de fin d'instruction du C.

Les paramètres effectifs sont des expressions (éventuellement complexes) dont la valeur doit être d'un type compatible avec celui du paramètre formel correspondant.

Exemple

Dans l'exemple ci-dessus, l'instruction `afficheMoyennes(i)` ; constitue un appel de procédure et les instruction de calcul de la moyenne et de l'affichage de la valeur obtenue sont déléguées à ce sous-programme, ce qui simplifie d'autant la fonction principale et la rend plus lisible.

3.3.3 Règles de visibilité

a. Structure d'un programme C

En C, la définition d'une fonction ne peut pas figurer à l'intérieur de la définition d'une autre fonction. La structure d'un programme C est une liste de définitions ou déclarations de variables, types et fonctions, qui doit contenir la définition de la fonction `main`.

L'ordre de définition des éléments (types, variables, fonctions) peut varier mais un élément doit toujours avoir été déclaré avant d'être utilisé. Si on définit une fonction f qui fait appel à une fonction g avant que g n'ait été définie, il y aura erreur de compilation. (En fait, le compilateur considèrera que la fonction g a le type entier, et on aura ensuite des erreurs de compilation lors de la définition ou de l'utilisation de g, si celle-ci n'a pas le type `int`.)

Pour résoudre ce problème d'ordre des définitions, on distingue la *définition* d'une fonction de sa *déclaration*. Si on prend soin de déclarer au préalable tous les objets utilisés, on peut ensuite les définir dans un ordre quelconque.

La déclaration d'une fonction s'appelle son *prototype* : c'est l'en-tête de la fonction suivie d'un point virgule. Un prototype suit la syntaxe suivante dans la norme ANSI :

```
<type du résultat> <nom de fonction> (<liste des paramètres>) ;
```

où `<liste des paramètres>` comporte le nom et le type de chacun des paramètres

```
<type par1> <par1>, <type par2> <par2>,… ,<type parN> <parN>) ;
```

Par exemple, on peut déclarer la fonction `discriminant` avec la déclaration :

```
float discriminant (int a, int b, int c);
```

et utiliser un appel à `discriminant` avant qu'elle n'ait été définie.

b. Variables locales vs *variables globales*

Comme on l'a déjà vu, C fait la distinction entre les variables globales et les variables locales. Les variables définies dans le bloc de définition d'une fonction sont locales à cette fonction tandis que les variables définies hors de tout bloc de fonction sont des variables globales.

Exemple

Dans l'exemple précédent du calcul d'une liste de factorielles. La variable `e`, qui représente l'entier donné par l'utilisateur et dont on calcule la factorielle, est utilisée dans la fonction `main`. Elle est définie dans le bloc de définition de cette fonction. Elle est donc locale à la fonction main.

En revanche, si on modifie la définition de la fonction valide pour que cette fonction modifie la valeur de l'entier donné par l'utilisateur (effet de bord), il faut que la variable `e` soit accessible depuis la fonction `valideAvecEffetsBord` et donc qu'elle soit globale. C'est pourquoi, elle est déclarée ci-dessous hors de tout bloc de fonction et avant la définition de `valideAvecEffetsBord` dans le programme.
Rappelons que l'utilisation d'une variable globale est ici indispensable car le paramètre p de `valideAvecEffetsBord` est passé en valeur et sa modification dans le corps de `valideAvecEffetsBord` n'a aucune incidence sur la valeur du paramètre d'appel dans fonction appelante.

```
# include <stdio.h>
# define TAILLEMAX 100

typedef enum booleen {faux,vrai};

int e;

enum booleen valideAvecEffetsBord(int p) {
      /* valideAvecEffetsBord teste si la valeur passée */
      /* en paramètre est valide ou non. Si la valeur */
      /* n'est pas valide, un message est adressé à */
      /* l'utilisateur et la valeur est remplacée par 0 */
      /* pour sortir du programme. */

      if (p!=0 && p<50)
         return(vrai);
      else {
         if (p!=0) {
          printf("Valeur non valide");
          e=0;
          }
          return(faux);
       }
}

int fact(int p) {
               /* calcule factorielle p */
      int i, res;
      res=1;

      for (i=2; i<=p; i=i+1)
         res= res*i;
      return(res);
}

int main () {
      do {
          printf("Entrez un entier : ");
          scanf("%d", &e);
          if (valideAvecEffetsBord(e)==vrai)
              printf("Factorielle de %d = %d.\n", e, fact(e));
      } while (e!=0);
}
```

Voilà ce que donne l'exécution de ce programme si l'utilisateur entre successivement 6, 4 et 51 comme entiers. 51 n'étant pas une valeur acceptable, le message (`Valeur non valide`) est adressé à l'utilisateur et le programme quitte la boucle while.

```
Entrez un entier : 6
Factorielle de 6 = 720.
Entrez un entier : 4
Factorielle de 4 = 24.
Entrez un entier : 51
Valeur non valide
```

De manière plus générale, les définitions des fonctions ou variables constituant un programme peuvent être réparties dans plusieurs fichiers. La portée d'une variable globale ou d'une fonction est normalement celle du fichier où elle est définie. Mais on peut étendre la définition de la variable à un autre fichier : une déclaration précédée du mot clé `extern` permet d'indiquer que la définition de la variable ou fonction utilisée figure dans un autre fichier. Mais dans ce cas, il faudra prendre garde à n'avoir défini la variable que dans un seul fichier (dans les autres, il n'y aura que la déclaration `extern`). Les différents fichiers peuvent être alors compilés séparément, puis liés ensemble pour obtenir un code exécutable. (Pour obtenir un code effectivement exécutable, il faut cependant avoir défini quelque part la fonction `main`.).

c. Visibilité des fonctions et prototypage

Les fonctions définies en C sont toujours globales et peuvent donc être utilisées n'importe où, pourvu qu'elles aient été déclarées au préalable en tête de fichier (avec la mention `extern` si elles ne sont pas définies dans ce fichier).

Exemple

Voici une nouvelle version (`valide2`) de la fonction `valide` présentée plus haut. Cette fois, la fonction `valide2` appelle elle-même la fonction `fact`. Il faut donc impérativement que la fonction `fact` ait été déclarée avant la fonction `valide2` et qu'elles soient toutes deux déclarées avant la fonction `main`.

```
enum booleen valide2(int p) {
                /* valide2 teste si la valeur passée
                   en paramètre est valide ou non */
      if (p!=0 && fact(p)<300)
          return(vrai);
      else
          return(faux);
}
```

Voici ce que l'on obtient pour le programme du calcul d'une liste de factorielle avec cette nouvelle version de la fonction `valide`. On pourrait inverser les deux prototypes des fonctions `valide2` et `fact`, puisque toutes les déclarations (prototypes) sont regroupées ici avant la liste de définitions des fonctions :

```
#include <stdio.h>
#define TAILLEMAX 100

typedef enum booleen {faux,vrai};
```

```
/* Prototypes des fonctions */

int fact(int p);
enum booleen valide2(int p);

/* définitions des fonctions */

enum booleen valide2(int p) {
        /* valide2 teste si la valeur passée
            en paramètre est valide ou non */
      ...
}

int fact(int p) {
          /* calcule factorielle p */
      ...
}

int main () {
int e;
do {
      printf("Entrez un entier : ");
      scanf("%d", &e);
      if (valide2(e)==vrai) {
        printf("Factorielle de %d = %d.\n", e, fact(e));
        }
      }
      while (e!=0);
}
```

d. Bibliothèques et fichiers d'en-tête

Ces problèmes d'interdépendance entre fonctions sont d'autant plus importants qu'en C, du fait de l'utilisation de bibliothèques de fonctions, un programme est le plus souvent réparti dans plusieurs fichiers.

Bibliothèques

En C, en effet il n'existe pas de fonction prédéfinie par le langage pour gérer les entrées et sorties, les calculs mathématiques, la manipulation de chaînes de caractères, etc. En revanche, on peut utiliser des fonctions définies dans des *bibliothèques* (ou *librairies*). Ces bibliothèques sont des extensions du langage C. On indique au moment de la compilation quelles sont les bibliothèques utilisées pour que le compilateur puisse avoir accès à la définition de ces fonctions.

Il existe depuis la norme ANSI une bibliothèque de C standard, indépendante de l'implémentation et du système d'exploitation. Les déclarations de ces fonctions sont réparties dans des fichiers dont les principaux regroupent :

- La bibliothèque standard qui constitue l'interface de programmation avec les primitives de bas niveau du système d'exploitation ;
- La bibliothèque standard de gestion des entrée/sortie qui contient les primitives de lecture et d'écriture de caractères, flot d'entrée ou fichiers ;

– La bibliothèque standard des fonctions de manipulation de chaînes, copie de chaîne, etc. ;

– La bibliothèque mathématique qui contient les fonctions mathématiques usuelles.

Fichiers d'en-tête

Pour pouvoir utiliser dans un programme une fonction, il faut qu'elle soit déclarée, que ce soit une fonction utilisateur ou la fonction prédéfinie d'une bibliothèque.

Pour éviter d'avoir à taper les prototypes (ce qui serait laborieux et source d'erreurs), il existe des *fichiers d'en-tête* (déclarant les prototypes de ces fonctions) dont l'extension est `.h` (pour *header*) qui regroupent ces déclarations de fonctions par familles.

Le C ANSI définit une liste de 15 fichiers d'en-tête standards. On trouve notamment :

- `stdio.h` : gestion des entrées sorties (`io` pour *Input*/*Output*) ;
- `stdlib.h` : traitements courants ;
- `ctype.h` : fonctions de tests sur les caractères ;
- `string.h` : manipulation de chaînes de caractères ;
- `math.h` : traitements mathématiques ;
- `time.h` : manipulation des primitives de temps (date, heure, etc.) ;
- `limits.h` : caractéristiques de l'implémentation (valeurs extrêmes, etc.)

Préprocesseur

On n'a pas besoin de recopier tous les prototypes de fonctions de la bibliothèque (et plus généralement les définitions des éléments utilisés : types, pseudo-constantes...) que l'on utilise dans un fichier d'un programme. C'est le préprocesseur qui se charge de ce travail.

Le rôle du préprocesseur est de préparer un programme source pour le compilateur. En fonction des directives qu'il rencontre dans le fichier source, le préprocesseur crée un fichier intermédiaire utilisable pour le compilateur. En outre, le préprocesseur supprime les commentaires, les caractères d'espacement inutiles, les sauts à la ligne, etc.

C'est donc le préprocesseur qui se charge d'inclure dans le fichier intermédiaire les prototypes des fonctions déclarées dans un fichier d'en-tête. En fait, c'est comme si le texte du fichier inclus était littéralement introduit à cet endroit dans le texte du programme.

Les directives adressées au préprocesseur sont des lignes commençant par le signe # qui figurent en un endroit quelconque du fichier source. Par contre, le mot clé introduit par le signe # doit être impérativement collé au #.

Exemple

```
#define TAILLEMAX 100
#include <stdio.h>
```

Il faut souligner qu'une directive n'est pas une instruction et ne se termine pas par un point virgule. Il faut donner une seule directive par ligne. Si la directive ne tient pas sur une ligne, elle peut se poursuivre sur la ligne suivante si la première ligne se termine par le signe « \ ».

Différents types de directives peuvent être adressées au préprocesseur. Nous avons déjà rencontré les définitions de constantes et les inclusions de fichiers.

La syntaxe d'une définition de constante est la suivante :

```
#define <constante> <chaîne de remplacement>
```

Lorsqu'il rencontre une directive de définition de constante, le préprocesseur remplace toutes les occurrences de la constante dans le fichier source par la chaîne de remplacement dans le fichier intermédiaire. Le préprocesseur remplace toutes les constantes définies par leur chaîne de remplacement dans l'ordre où il trouve les définitions dans le fichier source.

La syntaxe d'une inclusion de fichiers est la suivante :

```
#include <nom de fichier>
```

Le préprocesseur recopie le contenu du fichier <nom de fichier> dans le fichier intermédiaire à l'endroit où figure la directive. C'est ainsi que l'inclusion d'un fichier d'en-tête décharge le programmeur de la nécessité de donner le prototype de toutes les fonctions qu'il utilise.

La mention du nom de fichier se note entre chevrons (« < » et « > ») quand il s'agit d'un fichier d'en-tête standard. Dans le cas où le fichier inclus est un fichier créé par l'utilisateur, on donne son nom entre guillemets. Le nom spécifie le chemin d'accès au fichier. Il peut alors être donné relativement au répertoire de travail de l'utilisateur, ou bien « complet », c'est-à-dire donné sous la forme du chemin complet d'accès à ce fichier depuis la racine.

Exemple

```
#include <string.h>  /* inclusion du fichier standard string.h */
          /* inclusion de fichiers personnels  du programmeur */
#include "monFichier.h"
#include "MonRep/monFichier2.h"
#include "/users/rcln/nazarenk/MonRep/monFichier2.h"
```