Algorithmes

Définition

- Encyclopédie Larousse :

"Tout procédé systématique de calcul"

- Petit Larousse :

“Processus de calcul permettant d’arriver à un résultat final déterminé”

- Techniques de l'ingénieur :

"Ensemble de règles opératoires ou de procédés, définis en vue d'obtenir un résultat déterminé au moyen d'un nombre fini d'opérations".

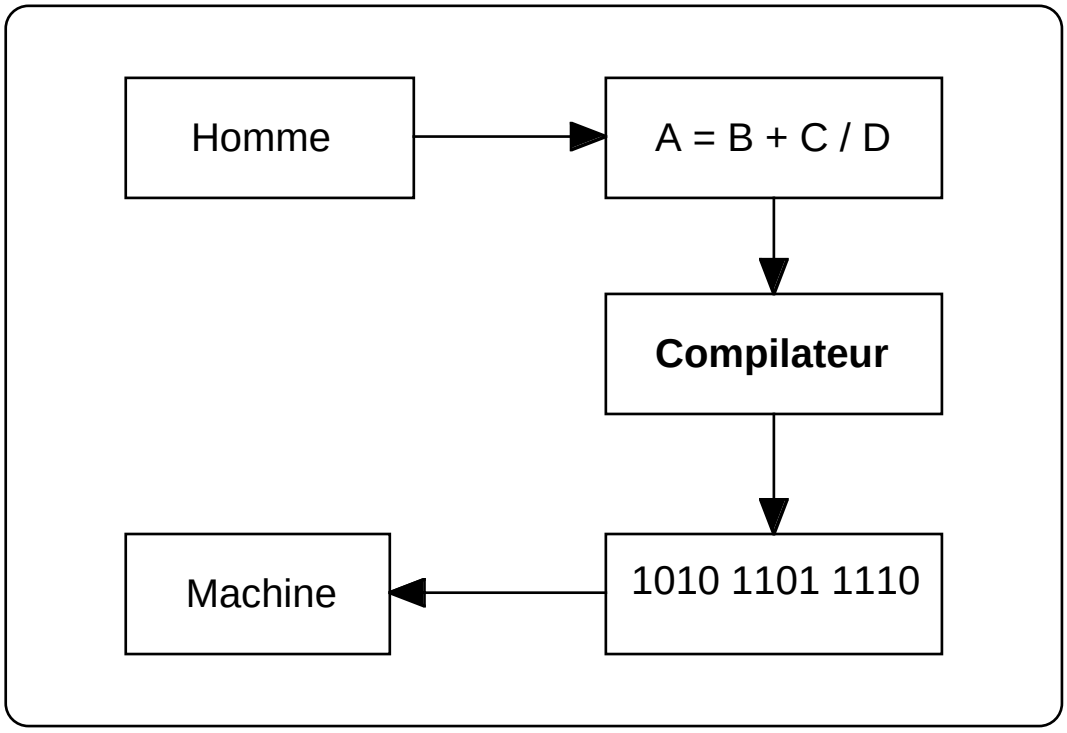
- M. LUCAS (dans «Algorithmique et représentation des données»)

« Nous définissons un algorithme comme la description d’un schéma de comportement exprimée à l’aide d’un répertoire fini d’actions et d’informations élémentaires identifiées, bien comprises et réalisables à priori .»

Ordinateurs - Langages.

Un ordinateur est capable de traiter des informations situées dans sa **mémoire** (centrale ou secondaire) sous forme numérique (format binaire 0 ou 1).

Le traitement à effectuer est décrit par une séquence de commandes, appelées **instructions**, qui constituent le **langage-machine** de l'ordinateur (suite de 0 et de 1).



L'expression d'un traitement à l'aide du langage-machine d'un ordinateur n'est pas très facile. C'est pourquoi, on a inventé des langages, dits **évolués**, qui permettent d'exprimer plus facilement, à l'aide d'une séquence d'instructions appelée **programme**, le traitement souhaité (le langage C, par exemple).

Afin que l'ordinateur puisse comprendre un programme en langage évolué, il existe des interfaces, appelées **compilateurs** qui traduisent les instructions du langage évolué en instructions de l'ordinateur (un compilateur C par exemple).

Le besoin d'algorithmes

Pour indiquer à une personne le chemin à suivre pour aller à l'endroit demandé, on remarque :

— qu'il est nécessaire de connaître le lieu même ou l'on se trouve (on ne peut fournir le renseignement si l'on est soi-même perdu). C'est la connaissance de l'**état initial**;

— que le demandeur doit être capable de **réaliser un certain nombre d'actions** (reconnaître la droite et la gauche, lire le nom des rues…);

— qu'il est nécessaire de connaître l'endroit auquel la personne désire se rendre : c'est la connaissance de l'**état final**.

Le processeur (humain dans le cas présent) doit comprendre et exécuter un certain nombre d'instructions.

Un algorithme est un ensemble de règles opératoires rigoureuses, ordonnant à un processeur particulier d'exécuter dans un ordre déterminé un nombre fini d'opérations élémentaires, pour résoudre tous les problèmes d'un type donné.

Les opérations élémentaires sont données pour un processeur bien spécifique possédant un jeu d'opérateurs bien défini.

Autres analogies

Une **recette de cuisine** est un algorithme. Elle conduit (du moins en théorie) à un résultat certain, pourvu que la personne qui cuisine soit capable d'effectuer chacune des opérations élémentaires décrites dans la recette : casser un œuf, mélanger la farine et le sucre, etc.
La **méthode traditionnelle de multiplication** de nombres de plusieurs chiffres est un algorithme. Elle conduit au bon résultat de manière certaine pourvu que l'exécutant connaisse ses tables de multiplications pour tous les nombres de 1 à 9 et sache faire des additions.
Tout algorithme a **des ingrédients** (les produits alimentaires pour une recette de cuisine, chiffres pour la multiplication), une **entrée** (œuf, sucre et fruits pour la recette, les deux nombres à multiplier pour la multiplication), une **sortie** (la tarte pour la recette de cuisine, le résultat pour la multiplication) et des **étapes** ou sous-algorithmes (faire la pâte, garnir la pâte, faire cuire la tarte pour la recette de cuisine, multiplier le nombre du haut par chacun des chiffres du bas puis additionner tous les résultats pour la multiplication).

Nécessité

La description et l'analyse du problème se font en langage naturel : le français.

On choisit de décrire le traitement d'un problème dans un langage relativement universel, comportant des instructions de base dont la signification est connue, et pouvant évoluer selon les besoins.

On appelle ce langage universel et extensible **langage algorithmique**. Les descriptions exprimées dans ce langage sont des **algorithmes**.

L'expression de l'algorithme est primordiale :

— pour la communication entre utilisateurs : la résolution d'un problème peut être menée conjointement ou successivement par des personnes différentes, qui doivent pouvoir être mises au courant avec précision du travail effectué. Une normalisation de l'expression des traitements est nécessaire afin que chacun ait le même langage.

— pour la clarté : la programmation, étape ultérieure, ne supporte pas l'ambiguïté.
Les langages de programmation ont une syntaxe stricte. Ceci est nécessaire pour que le compilateur puisse reconnaître et traduire correctement les différentes instructions. Or, cette rigueur n'est pas nécessaire dans la description du traitement d'un problème et elle peut même nuire à la clarté de la description.

— Un langage de programmation peut exister sur un ordinateur et pas sur un autre.
Le choix d'opter pour un langage de programmation pour décrire une solution, oblige à traduire sa description dans un autre langage en cas de changement d'ordinateur, si le nouvel ordinateur ne possède pas ce langage de programmation.

— Le choix du bon algorithme pour effectuer un traitement donné est plus important qu'une bonne mise en œuvre de l'algorithme dans le langage cible.
Optimiser la mise en œuvre d'un algorithme améliore peut améliorer le temps de calcul ou diminuer l'espace mémoire requis, mais les gains restent très inférieurs à ceux procurer par l'utilisation de l'algorithme le plus adapté.

Les règles à suivre

Avant d'être programmé, un problème doit être résolu.

Il faut que l'on ait trouvé une méthode de traitement informatique permettant de répondre au problème posé.

Une connaissance approfondie du langage est nécessaire.

Il faut bien connaître le mode de fonctionnement du langage, en plus de sa syntaxe; cette connaissance représente la meilleure alliée du programmeur.

Notions de Pseudo-code

Le Pseudo-code se rapproche d'un langage de programmation, mais aucun compilateur ou interpréteur ne peut transformer ce pseudo-code en programme exécutable. Il permet de définir des traitements de façon très structurée. L'expression même de la structure est déjà une forme de commentaire.

Les structures de contrôle du pseudo-code :

- séquence
- alternative
- itérative

La séquence

Il s'agit d'exprimer la juxtaposition, ou la mise en séquence de plusieurs actions.

```
DEBUT
        action 1;
        action 2;
        action 3
FIN
```

DEBUT et FIN repèrent les points d'entrée et de sortie d'une suite d'actions, appelée également module.

L'alternative

C'est l'expression d'un choix, gouverné par une «condition»ou test.

Forme complète :

```
SI condition
        ALORS
                DEBUT
                        action 1;
                        action 2;
                        action 3
                FIN
        SINON
                DEBUT
                        action 4;
                        action 2
                FIN
FINSI;
```

Forme dégradée :

```
SI condition
        ALORS
                DEBUT
                        action 1
                        action 3;
                FIN
;       /* FIN de SI */
```

Commentaire

On utilisera la forme :

```
/* commentaire */
```

pour commenter une ou plusieurs instructions (ces commentaires sont utiles pour compléter et clarifier un algorithme).

Exemple

Une personne veut aller au théâtre.

Soit elle a déjà un billet, auquel cas elle se rend directement au théâtre.

Soit elle n'en a pas. Il lui faut alors retirer de l'argent à la banque, réserver sa place par téléphone, aller retirer son billet à la caisse une heure au plus tard avant le spectacle.

Dans les deux cas, il ne lui reste plus qu'à rechercher sa place et s'y installer.

Une solution

```
DEBUT           /*   aller au théâtre   */

        SI possède-billet
                ALORS
                        se-rendre-au-théâtre-heure-H
                SINON
                        DEBUT     /* partie réservation */
                                passer-à-la-banque;
                                réserver-sa-place;
                                se-rendre-au-théâtre-heure(H-1);
                                retirer-le-billet;
                                attendre-heure-H
                        FIN             /* partie réservation  */
        ;
        rechercher-sa-place
        s'installer

FIN             /*   aller au théâtre   */
```

Alternative imbriquées

Une entreprise offre une prime aux employés suivant le nombre de leurs enfants :

- si pas d'enfant, pas de prime
-si 1 enfant, 500 F
- si 2 enfants, 750 F
- si 3 enfants, 1500 F
- si 4 enfants, 2600 F
- si plus de 4 enfants : 4000 F

Représenter la structure algorithmique qui permet d'exprimer la prime offerte **prime** en fonction du nombre d'enfants **nbenfants.**

Une solution

```
SI nbenfants = 0
        ALORS prime <-- 0
        SINON
                SI nbenfants = 1
                        ALORS prime <-- 500
                        SINON
                                SI nbenfants = 2
                                        ALORS prime <-- 750
                                        SINON
                                                SI nbenfants = 3
                                                        ALORS prime <-- 1500
                                                        SINON
                                                                SI nbenfants = 4
                                                                   ALORS prime <--
2600
                                                                   SINON prime <-- 4000
                                                                ;
                                                ;
                                ;
                        ;
                ;
;
```

Variante

```
SI nbenfants = 0
        ALORS prime <-- 0
;
SI nbenfants = 1
        ALORS prime <-- 500
;
SI nbenfants = 2
        ALORS prime <-- 750
;
SI nbenfants = 3
        ALORS prime <-- 1500
;
SI nbenfants = 4
        ALORS prime <-- 2600
;
SI nbenfants >4
        ALORS prime <-- 4000
;
```

Avantage : meilleure lisibilité.

Inconvénient : moins performant.

Remède : utilisation de la structure « en rateau » : le CHOIX.

LE CHOIX

Formulation :

```
CHOIX <expression> SELON
        CAS1 : <instruction 1>;
        CAS2 : <instruction 2>;
        CAS3, CAS4 : <instruction 3>;
        …
        CASN : <instruction n>
        SINON <instruction n>
FCHOIX;
```

Le test porte sur la valeur de l'expression, qui doit être de même type que les valeurs proposées en CASn.

La ligne commençant par SINON peut être omise si l'ensemble des valeurs que peut prendre <expression> est fini.

Notre exemple devient alors :

```
CHOIX nbenfants SELON
        0 : prime <-- 0 ;
        1 : prime <-- 500 ;
        2 : prime <-- 750 ;
        3 : prime <-- 1500 ;
        4 : prime <-- 2600 ;
        SINON prime <-- 4000
FINCHOIX;
```

L'ITÉRATIVE

La structure itérative ou boucle, est utilisée lorsque l'on doit effectuer le même traitement plusieurs fois.

Plusieurs possibilités se présentent :

Forme de 1 à n occurrences (le traitement est exécuté au moins une fois) :

```
REPETER
        DEBUT
                action 1;
                action 2
        FIN
JUSQUA condition-d'arrêt-vérifiée;
```

Forme de 0 à n occurrences (le traitement peut ne jamais être exécuté) :

```
TANTQUE non-condition-d'arrêt
FAIRE
        DEBUT
                action 1;
                action n
        FIN
FINTANTQUE;
```

Forme où le nombre d' occurrences est fixe :

```
POUR compteur VARIANT DE mini A maxi
FAIRE
        DEBUT
                action 1;
                action 3
        FIN;
```

EXEMPLE

Le travailleur de force.

Voici ses consignes :

- remplir son seau de sable à coups de pelletées jusqu'à ce qu'il soit plein

- quand le seau est plein, aller le vider dans la benne

- une fois que la benne est pleine, monter sur le tracteur et aller vider le sable au lieu convenu.

On suppose :

- que la benne peut contenir plusieurs seaux

- que la benne et le seau sont vidés au départ

- qu'il y a plus de sable que ne peut en contenir une benne.

UNE SOLUTION

```
DEBUT   /* travailleur de force   */
        REPETER
                REPETER
                        DEBUT   /* pelletée   */
                                charger-pelle;
                                vider-pelle-dans-seau
                        FIN   /* pelletée   */
                JUSQUA seau-plein;
                verser-seau-dans-benne
        JUSQUA benne-pleine;
        aller-vider-la-benne-en-tracteur
FIN    /* travailleur de force   */
```

UNE VARIANTE

On peut supposer que la quantité de sable est supérieure à la valeur de plusieurs bennes.

```
DEBUT   /* travailleur de force   */
        REPETER
                REPETER
                        REPETER
                                DEBUT   /* pelletée   */
                                        charger-pelle;
                                        vider-pelle-dans-seau
                                FIN   /* pelletée   */
                        JUSQUA seau-plein;
                        verser-seau-dans-benne
                JUSQUA benne-pleine;
                aller-vider-la-benne-en-tracteur
        JUSQUA tas-de-sable-vide;
        téléphoner-au-chef-que-le-travail-est-terminé
FIN    /* travailleur de force   */
```

UNE AUTRE VARIANTE

Notre travailleur de force relaye un collègue. Ce dernier peut avoir laissé un seau plein à l'heure de la débauche, et une benne partiellement remplie. On aura alors :

```
DEBUT   /* travailleur de force   */
        REPETER
                TANTQUE benne-pas-pleine
                FAIRE
                        TANTQUE seau-pas-plein
                        FAIRE
                                DEBUT   /* pelletée   */
                                        charger-pelle;
                                        vider-pelle-dans-seau
                                FIN   /* pelletée   */
                        FINTANTQUE;
                        verser-seau-dans-benne
                FINTANTQUE
                aller-vider-la-benne-en-tracteur
        JUSQUA tas-de-sable-vide;
        téléphoner-au-chef-que-le-travail-est-terminé
FIN    /* travailleur de force   */
```

NOTATION ALGORITHMIQUE

Les structures de contrôle

alternative :

```
SI <condition>
        ALORS <instruction>
;

SI <condition>
        ALORS <instruction 1>
        SINON <instruction 2>
;
```

choix :

```
CHOIX <expression> SELON
        Cas1 : <instruction 1>;
        Cas 2 : <instruction 2>;
        ...
        Cas n : <instruction n>
        SINON <instruction z>
FINCHOIX;

CHOIX <expression> SELON
        Cas1 : <instruction 1>;
        Cas 2 : <instruction 2>;
        ...
        Cas n : <instruction n>
FINCHOIX;
```

répétitive 1 à n occurences :

```
REPETER
        <instruction>
JUSQUA <expression>;

POUR variable VARIANT DE <valeur1> A <valeur2>
FAIRE <instruction>;
```

répétitive 0 à n occurences :

```
TANTQUE <condition>
        FAIRE <instruction>
FINTANTQUE;
```

La variable

Une variable est une donnée du problème. Elle correspond à l'association d'un **identificateur** (un nom) et un **contenu** (une valeur). Son identificateur permet de manipuler son contenu, qui peut ou non changer au cours du traitement réalisé par l'algorithme.

Il est obligatoire de connaitre la liste complète de toutes les variables utilisées dans l'algorithme.

VARIABLES: nom, n, m, delta;

L'affectation : <--

La variable située à gauche du signe <-- reçoit la valeur de l'expression située à droite du signe. Cela peut être une constante, le résultat d'un calcul, la valeur retournée par une fonction, etc...

Pi <-- 3,15159; N <-- M + 1;

L'incrémentation

L'incrémentation est une affectation spéciale mettant en jeu une seule variable à la fois à gauche et à droite du symbole d'affectation "<--".
Cette forme d'instruction est largement utilisée en programmation.

A <-- A + 1;

Pour bien comprendre cette instruction il faut considérer que la machine travaille en deux temps :

— Premier Temps :
La machine ignore totalement la partie à gauche du <--, son travail ne porte que sur la partie droite pour calculer la valeur de A +1,

— Second Temps :
Considérant la variable à gauche du signe <--, la valeur calculée dans le premier temps lui est affecté.

A prend donc pour nouvelle valeur, l'ancienne valeur de A, augmentée de une unité, qui est ici la valeur du **pas d'incrémentation**.

Les expressions numériques

Elles sont obtenus par combinaison d'opérateurs, de valeurs et de variables, ainsi que de fonctions.

opérateurs mathématiques :
+ , **-** , **/** (division), *****, **mod** (modulo ou reste de la division entière), **div** (division entière).

Exemples : 14 div 4, donne pour résultat 3
14 mod 4, donne pour résultat 2
A est multiple de B si A mod B égale à 0.

les fonctions
abs() (valeur absolue), **sin**(), **cos**(), **SQRT**() (racine carrée), …

Exemples : A <-- abs(B) + 3;

Les expressions logiques

Une expression logique (ou booléenne) est soit VRAIE, soit FAUSSE. Cette valeur de vérité est le résultat de l'expression logique est peut être affecté à une variable.
Les expressions logiques sont employées en tant que condition par certaines instructions :

SI condition ALORS …

Elles sont obtenues par combinaison d'opérateurs de comparaisons, d'opérateurs booléens, de valeurs, variables ou bien fonction.

opérateurs booléens :
et, ou, non.

opérateurs de comparaison :
<, <=, = (égalité), >, >=, <> (différent de)

Exemples : SI A = 1 ALORS …

PlusGrand <-- A > B;

Selon les valeurs de A et B, la variable PlusGrand va prendre pour valeur VRAI si A est supérieur à B ou FAUX si A n'est pas supérieur à B.

Les entrées-sorties

La lecture

La lecture est représentée dans l'algorithme par le mot clé «lire». La lecture d'une variable communique à cette variable une valeur issue d'un organe d'entrée (clavier ou autre fichier d'entrée).

```
lire (A); /* saisie au clavier de la valeur a affecter à A */
lire (A, B, C); /* de même pour A, B et C */
```

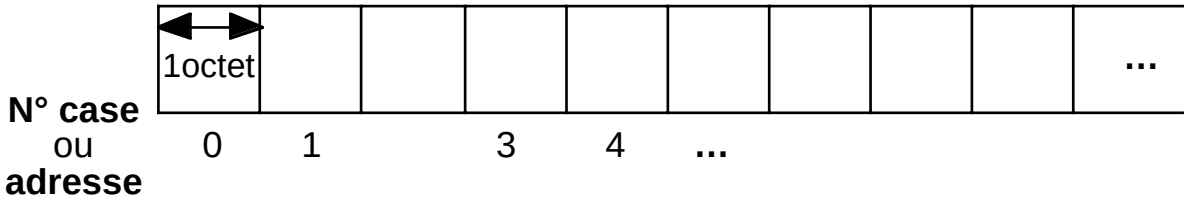
L'écriture

Elle est représentée par le mot «écrire». L'écriture d'une variable communique la valeur de cette variable à un organe de sortie (écran ou imprimante ou fichier de sortie).

```
écrire («Donner une valeur entre 0 et 100»);
écrire (N); /* Affiche le contenu de la variable N */
```

Représentation de la mémoire de l'ordinateur

La mémoire d'un ordinateur est une suite de cases contiguës de tailles identiques correspondant chacune à un **octet** ou 8 **bits** (permettant le codage de 256 valeurs).



Définition et utilisation des données

Une donnée informatique est représentée par son ***identificateur***, un **contenu**, et une **adresse en mémoire**.

Son *contenu* peut évoluer au cours de l'exécution du traitement ou rester fixe.

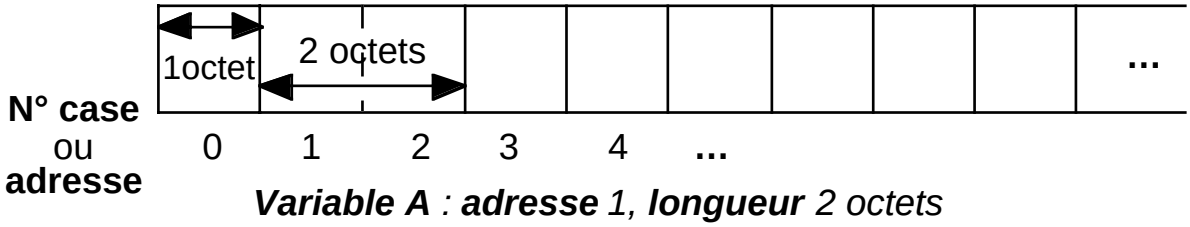
— Les *constantes* gardent une valeur fixe durant l'exécution et sont déclarées en tout début d'algorithme

```
Exemple : CONSTANTE   PI <-- 3.14159 ;
                      NOM <-- "TOTO" ;
```

— Les *variables* ont une valeur qui évolue au cours du temps. Cette variable a une **adresse en mémoire** et **une longueur** qui dépend du **profil** de la donnée : il faut donc lui donner un **type** au moment de la déclaration.

```
Exemple : VARIABLE    A :          ENTIER;
                          /* par exemple sur 2 octets */
                      réponse :    CARACTERE;
```

Représentation en mémoire :



***Variable A** : **adresse** 1, **longueur** 2 octets*

Pour les constantes, c'est la valeur qui détermine le type.
Remarque : Toute variable doit obtenir une valeur lors de sa première apparition dans le corps du programme.

Types de données

Exemples de définition de variables :

Les premières lignes correspondent au libellé complet des objets dans le langage naturel, et les secondes lignes définissent l'identificateur, le type et un exemple de valeur de chaque objet.

Lettre de l'alphabet
caract **CARACTERE** 'b'

A est plus grand que B (A>B)
asupb **BOOLEEN** FAUX

Hauteur de l'ascenseur en mètres
hteur **RÉÉL** 5.72

Vitesse de l'ascenseur en m/s
vitesse **RÉÉL** 1.2

Numéro de l'étage
etage **ENTIER** 3

Types scalaires :

Les valeurs appartiennent à un ensemble fini et ordonné :

— types scalaires prédéfinis :

ENTIER, BOOLEEN, CARACTERE

Types non scalaires :

L'ensemble des valeurs est indénombrable. Ce sont les RÉÉLS, CHAÎNES DE CARACTÈRES.

Types structurés :

Ils **collectionnent** ou **assemblent** des données de type définis plus haut : il s'agit de TABLEAUX ou de STRUCTURES.

Type entier

Un ordinateur ne peut représenter qu'un nombre fini d'entiers, dépendant de la taille mémoire associée à ce type.

Si l'on a une taille mémoire de 2 octets pour un entier, alors toute valeur comprise entre -32768 et + 32767 est valide pour ce type.

Exemple :

```
CONSTANTE   TAILLELIGNE <-- 80 ;
VARIABLE    numéro, compteur, ligne : ENTIER ;
```

Les expressions suivantes sont correctes :

```
numéro + compteur
numéro + ligne DIV TAILLELIGNE
numéro - 100
compteur MOD TAILLELIGNE + 1
```

Une expression est correcte si elle calcule un résultat de type déterminé à partir de valeurs d'un types donnés.

Exemple :

```
X : ENTIER;

X <-- 1/2;          /* X contiendra 0 et non 1,5 */

X <-- 1,5 +4,3      /* INCORRECT */
```

Si l'on a la possibilité de travailler sur des entiers codés sur 4 octets, on peut utiliser un éventail de valeurs allant de - 2 milliards à + 2 milliards.

Type RÉÉL

Les variables réelles sont caractérisées par l'étendue des valeurs qu'il est possible de représenter, et par la précision avec laquelle elles peuvent être représentées.

Il est approximativement vrai de dire que dans un ordinateur qui a une précision de dix chiffres significatifs, deux nombres dont les valeurs diffèrent à partir du onzième chiffre significatif ont la même représentation en mémoire.

Lors de longs enchaînements de calculs, on peut donc perdre de la précision, et arriver à des résultats inexacts.

Exemple : CONSTANTE LIMITE <-- 6.23E+7;
VARIABLE x, y : RÉÉL;

Exemple : 2, +2, -7,2E+24 sont valides.

Dans un programme, on utilisera la notation 2.0 pour indiquer que ce n'est pas 2 en valeur entière, mais 2 en valeur réélle, et ce de façon à obtenir des calculs correctes.

Opérateurs sur ENTIERS et RÉÉLS

Il est important de vérifier le type des valeurs utilisées en entrée ou en sortie des différents opérateurs et fonctions.

— Sur les nombres entiers :

+, - (inversion de signe ou soustraction), * ,
/ (avec résultat réel), DIV, MOD.

—Sur les nombres réels :

+, -, * ,/.

— Fonctions de calcul standard :

Formulation	Type de l'argument	Type du résultat
ABS(x)	ENTIER,RÉÉL	Idem argument
SQR(x) (carré)	ENTIER,RÉÉL	Idem argument
SIN(x)	ENTIER,RÉÉL	RÉÉL
COS(x)	ENTIER,RÉÉL	RÉÉL
ARCTAN(x)	ENTIER,RÉÉL	RÉÉL
SQRT(x) (racine car.)	ENTIER,RÉÉL	RÉÉL
LN(x)	ENTIER,RÉÉL	RÉÉL
EXP(x)	ENTIER,RÉÉL	RÉÉL
TRUNC(x)	PE pour x >= 0 PE(|x|), x < 0 PE : partie entière	ENTIER
ROUND(x)	TRUNC (x + 0,5) pour x >= 0 TRUNC(x - 0,5) pour x < 0	ENTIER

Priorité de opérations

Les règles d'évaluation sont les suivantes :

— + et - unaires.

— en l'absence de parenthèses, les multiplications et divisions (*, /, DIV et MOD) sont traitées avant les additions et soustractions.

— en cas d'égalité des priorités, l'évaluation se fait de gauche à droite.

— S'il y a des parenthèses, les expressions qu'elles contiennent sont évaluées en premier. Les fonctions dont les arguments se donnent entre parenthèses suivent cette règle et sont évaluées en premier.

Exemple : x * y / z --> (x * y) /z

x + y * z --> x + (y * z)

Type BOOLÉEN

Une variable de type booléen ne peut prendre que l'une des deux valeurs parmi {vrai, faux}.

Exemple :

```
VARIABLE        adesdents  : BOOLÉEN ;
```

Puis, à l'initialisation :

```
adesdents ( vrai ;      /* s'il s'agit d'un orque */
adesdents ( faux ;      /* s'il s'agit d'une baleine */
```

Les opérateurs booléens sont :

Opérateur ET

A	B	**A ET B**
FAUX	FAUX	FAUX
FAUX	VRAI	FAUX
VRAI	FAUX	FAUX
VRAI	VRAI	VRAI

Opérateur OU

A	B	**A OU B**
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	VRAI

Opérateur NON

A	**NON A**
FAUX	VRAI
VRAI	FAUX

NON est évalué en premier, puis ET et OU on la même priorité

```
A < B <= C              incorrect
A < B ET B <= C         incorrect
(A<B) ET (B<= C )       correct
```

Type CARACTÈRE

Le domaine de validité pour un CARACTÈRE est l ‘ensemble des valeurs permises par le code de représentation interne (ASCII en général)

C’est une ensemble fini (nous prendrons 256 valeurs qui correspondent aux possibilités de codages sur un octet), et ordonné. Les constantes de type CARACTÈRE seront notées entre ‘ .

Les opérateurs de comparaison <, >, =, <=, >=, <>, peuvent être appliqués sur les caractères, puisque ceux-ci appartiennent à un ensemble ordonné (ils correspondent à des valeurs entières).

```
VARIABLE      numcode : ENTIER ;
              réponse :  CARACTERE ;


    numcode <-- ‘c’;  /* correcte, numcode reçoit le rang de c dans
                                le code ASCII */
    réponse <-- ‘a’ + 1;
    écrire (réponse);       /* affiche le caractère suivant de a : b */
```

Type structure

Une variable de type structure correspond à une juxtaposition de plusieurs variables de type différents.

```
TYPE STRUCTURE INFOPLANETE :
                    numero : ENTIER;
                    visible : BOOLÉEN ;
                    diamètre, orbite : RÉÉL
               FIN ;
```

Ici INFOPLANETE regroupe 4 types d'information.

VARIABLE infop1 : INFOPLANETE ;

Pour accéder à une information de base, utiliser la notation suivante :

```
infop1.numero <-- 2367 ;
infop1.visible <-- vrai ;
infop1.diamètre <-- 120104 ;
infop1.orbite <-- 108.2 ; /* en gigamètres */
```

On peut aussi écrire :

```
infop1.visible <-- infop1.orbite < 4000 ;
```

Les noms des champs doivent être uniques à l'intérieur de chaque structure.

Type tableau

— Un tableau est une collection ordonnée de variables de même type.

— Un tableau est un ensemble d'éléments en nombre fixé, mémorisés dans une zone contiguë et accessibles par un, *ou plusieurs*, indices.

indice

*Lorsque l'indice est unique, le tableau est dit à **une dimension**.*

*Lorsque N indices sont nécessaires pour repérer une valeur, le tableau est dit à **N dimensions**.*

Cet indice permet d'accéder directement à chacun de ses éléments par l'intermédiaire de l'indice de cet élément.

— On réfère au **i**-ième élément du tableau **T** par **T[i]**.
Il est nécessaire de bien vérifier le contenu de la case avant d'y accéder (contient-elle une valeur correcte ?).

— Le nombre d'éléments du tableau doit être connue à l'avance.

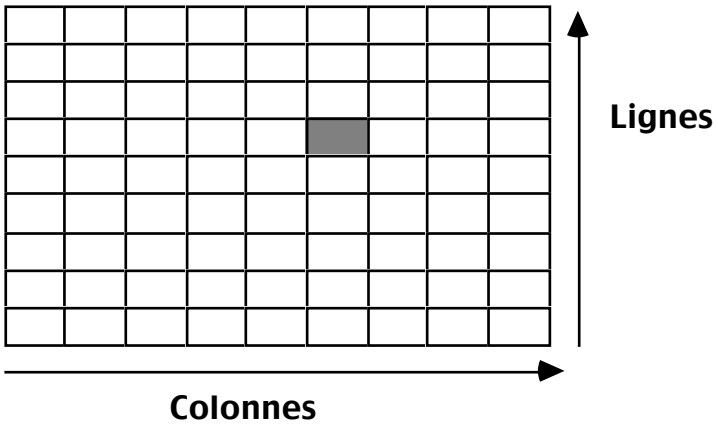
— Le type de la variable de base, qui détermine la taille du tableau.

```
valeurs : TABLEAU_DE n ENTIER;
valeurs [ 1 ] <-- 2;
valeurs [ 2 ] <-- 3;
valeurs [ i ] <-- valeurs [ i-1] + 2;

matrice : TABLEAU_DE n FOIS m RÉÉL;
matrice [ i ] [ j ] <-- x + 4,5;
```

C'est une structure de donnée fondamentale car il existe une correspondance directe avec la gestion de la mémoire d'un ordinateur.

*Pour accéder au contenu d'un mot mémoire en langage machine, il faut une **adresse**. Il est possible de se représenter la mémoire entière d'une machine comme un tableau dans lequel l'adresse mémoire joue le rôle d'indice.*

Un tableau à **deux dimensions** est organisé sous forme de **lignes** et de **colonnes**.



Exemple : une table de notes d'étudiants pour une matière pourrait contenir une ligne par étudiant et une colonne par contrôle.

Le calcul de la moyenne sur un contrôle s'effectue en divisant la somme des notes de la colonne par le nombre de colonnes.

Type CHAINE de CARACTERES

Tous les types définis ci-dessus peuvent être combinés. Il en est un plus particulier, très souvent utilisé : la chaîne de caractères.

Elle peut se comporter comme un tableau.

```
VARIABLE        ligne : TABLEAU_DE 80 CARACTÈRES ;

        ligne <-- "Ceci est une chaîne de caractères";
        /* charge le contenu de la chaîne indiquée entre guillemets dans le tableau défini précédemment à raison d'un caractère par case */

        écrire (ligne [ 0 ]);  /* affiche C */
```

Attention :

"123" n'est pas nombre
"+" n'est pas un opérateur

Opérations sur tableaux

Principaux traitements effectués sur les tableaux :

- définition
- chargement ou initialisation
- recherche
- tris

CHARGEMENT

Cas 1

Les données de la table sont des constantes : elles sont connues et doivent être chargées dans la table en début de programme :

```
VARIABLE      notes      :      TABLEAU_DE 6 RÉÉLS ;
              i          :      ENTIER ;
```

Dans le programme, on aura :

```
/*  ---  initialisation de la table  ---  */

notes [ 1 ] <-- 12,5;
notes [ 2 ] <-- 15,5;
…
notes [ 6 ]  <-- 10,5;
```

Il est plus courant d'initialiser les tables à partir d'une saisie directe. On a alors l'algorithme suivant :

```
POUR i VARIANT de 1 A 6 FAIRE
      DEBUT
            LIRE(valeur) ;
            notes [ i ] <-- valeur;
      FIN ;
```

Cas 2

Les données du tableau ne sont pas connues à l'avance, elles doivent être saisies.

```
CONSTANTE    maxjoueur <-- 5 ;

TYPE STRUCTURE  Joueur :
                    nom  :      TABLEAU_DE 15 CARACTÈRES ;
                    âge        :      ENTIER ;
                    score      :      ENTIER
                    FIN ;
VARIABLES     i            :      ENTIER ;
              tabjou       :      TABLEAU_DE maxjoueur  Joueur ;

/*  ---  initialisation de la table  ---  */

POUR i VARIANT de 1 A maxjoueur FAIRE
      DEBUT
            ECRIRE("nom du joueur : ") ;
            LIRE(tabjou [ i ].nom) ;
            ECRIRE(" âge du joueur : ") ;
            LIRE(tabjou [ i ].âge) ;
            tabjou [ i ].score <-- 0
      FIN ;
```

Variante : écrire une partie de l'algorithme permettant d'afficher le contenu de la table.
Variante :

```
VARIABLES     i            :      ENTIER ;
              c            :      CARACTERE;
/*  ---  initialisation de la table  tabjou définie plus haut ---  */
i <-- 1 ;
REPETER
      DEBUT
            ECRIRE(" nom du joueur : ") ;
            LIRE(tabjou [ i ].nom) ;
            ECRIRE(" âge du joueur : ") ;
            LIRE(tabjou [ i ].âge) ;
            tabjou [ i ].score <-- 0 ;
            i <-- i + 1 ;
            ECRIRE("Y a t-il encore un joueur ?") ;
            LIRE (c)
      FIN
JUSQUA (i > maxjoueur) OU (c <> 'O') ;
```

Complément :

```
POUR i VARIANT DE 1 A maxjou FAIRE
        ECRIRE ("Joueur n° ", i, " : ", tabjou [ i ].nom, " -  ",
                        tabjou [ i ].âge, " ans, score :" , tabjou [ i ].score);
```

Que se passe-t-il si les joueurs ne sont pas au nombre de 5 ?

Une solution : marquer le dernier élément . Si on saisit les informations sur 3 joueurs, la quatrième case doit contenir des caractères représentant une valeur nulle pour la case.

On rajoute alors en fin de boucle d'initialisation de type variante :

```
SI i <= 5 ALORS tabjou[ i ].nom <-- " * " ;
```

Notre programme d'affichage, du type de complément devient alors :

```
TANTQUE (i < maxjoueur) ET (tabjou[i].nom <> " * " ) FAIRE
        DEBUT
                ECRIRE (" Joueur n° ", i, " : ", tabjou [ i ] .nom, " -  ",
                        tabjou [ i ].âge, " ans, score :"  , tabjou[ i ].score);
                i <-- i + 1
        FIN
FINTANTQUE ;
```

Une méthode de résolution de problèmes.

L’analyse descendante.

Le principe de l’analyse descendante est le suivant :

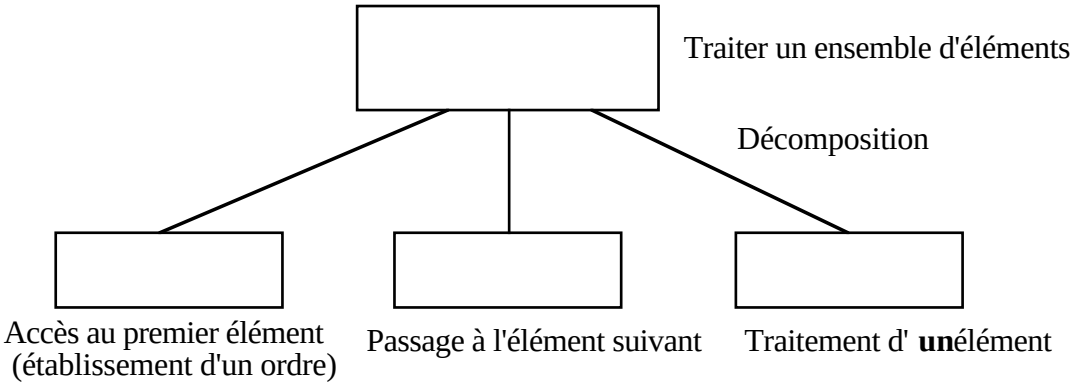
— Définir un certain nombre de problèmes simples que l’on sait traiter.

— Si le problème à résoudre peut être plaqué sur un de ces problèmes connus, lui appliquer la solution appropriée.

Sinon, décomposer le problème en un ensemble de problèmes plus simples, auxquels on appliquera à nouveau le principe de l’analyse descendante.

Un exemple simple d’analyse descendante est le cas de traitement des éléments d’un ensemble. Si les éléments de l’ensemble ne sont pas ordonnés, on établit un ordre entre ces éléments.
Pour pouvoir traiter un ensemble d’éléments, il suffit de pouvoir accéder au premier élément de l’ensemble, de pouvoir passer d’un élément à l’élément suivant et de savoir traiter **un** élément.

Le traitement d’un élément peut nécessiter à son tour la poursuite de l’analyse descendante.



Fonctions

Un long algorithme qui ne serait composé que de nombreuses déclarations et instructions écrites les unes derrière les autres serait très difficile à écrire et impossible à comprendre et à maintenir.
Il y a un moyen de **contrôler la complexité** des grands programmes : l'écriture de fonctions.

L'analyse descendante, qui décompose un problème initial en problèmes plus simples à résoudre, s'appelle également la méthode par raffinement graduel.

L'algorithme final sera constitué d'un ensemble de modules, organisés de manière hiérarchique, correspondant chacun à une décomposition du problème initial.

Une **fonction** est un module algorithmique, dédié à la résolution d'un sous-problème quelconque. On l'appelle encore procédure ou routine suivant les langages. Elle peut renvoyer une valeur, et dans ce cas, elle est assimilée à une <expression>.

Déclaration

Une fonction a un **nom**, des **variables** qui lui sont propres et son propre **jeu d'instructions**, comme un programme. Elle peut utiliser des variables globales à tout le programme principal (connues dans tout le programme), et accessibles aux autres fonctions.
Elle peut recevoir de l'information au moment de l'appel, ce seront les **paramètres**, et renvoyer une **valeur de retour**.
Structure de la fonction :

Entête :
FONCTION nom-de-la-fonction (liste de paramètres) :
TYPE de la valeur renvoyée par le fonction.

Déclaration des variables locales à la fonction :
CONSTANTE
VARIABLE

Corps de la fonction :
```
DEBUT           /*  ---  point d'entrée de la fonction  ---  */
        instructions ;
        RETOUR (valeur)          /*   attention au type   */
FIN ;       /*  ---  fin de la fonction  --- */
```

Appel de la fonction :

```
x <-- nom-de-la-fonction(n, m, ...) ;
        /* Où x est de type compatible avec celui de la fonction */

fonction (n, m, ...);   /* fonction sans valeur de retour */
```

Une fonction qui s'appelle elle-même est dite **récursive**.

Exemple

Voici une fonction sans argument et ne retournant aucune valeur, appelée dans un programme ne connaissant qu'une seule fonction.

```
PROGRAMME inverse ;
CONSTANTE    N <-- 50 ;   /* longueur de texte maxi */

FONCTION renverse : VIDE;
                        /*indique que la fonction ne retourne rien*/
VARIABLES      texte : TABLEAU_DE N CARACTÈRES ;
                        i               :       ENTIER ;
                        c               :       CARACTERE ;
DEBUT  /*  ---  début fonction renverse  ---  */
        texte <-- "" ;
        i <-- N;
        TANTQUE i >= 0 FAIRE
                DEBUT
                        LIRE(c) ;
                        texte [ i ] <-- c ;
                        i <-- i - 1;
                FIN ;
        ECRIRE (texte)
FIN ;   /*  ---   fin fonction renverse    ---  */
/*-----------------------------------------------------*/

DEBUT  /*  ---  début programme inverse  ---  */
        ECRIRE("Donnez votre texte : ") ;
        renverse (VIDE); /*indique qu'il n'y a pas d'arguments */
FIN ;    /*  ---    fin programme inverse     ---  */
```

Les paramètres

Si l'on désire paramétrer le nombre de fois où l'on imprime le texte, on rajoutera une variable fois, initialisée dans le programme principal et passée en argument à la fonction renverse.

```
PROGRAMME inverse ;
CONSTANTE    N <-- 50 ;     /* longueur de texte maxi */

VARIABLE fois    :    ENTIER ;  /* nombre de fois */
/*-----------------------------------------------------*/
FONCTION renverse (f : ENTIER) : VIDE ;
VARIABLES      texte          :       TABLEAU_DE N CARACTÈRES ;
               i              :       ENTIER ;
               c              :       CARACTERE ;
DEBUT  /*  ---  début fonction renverse  ---  */
       texte <-- "" ;
       i <-- N;
       TANTQUE I >= 0 FAIRE
              DEBUT
                     LIRE(c) ;
                     texte [ i ] <-- c;
                     i <-- i - 1
              FIN ;
       TANTQUE (f > 0) FAIRE
              DEBUT
                     ECRIRE (texte) ;
                     f <-- f - 1
              FIN
       FINTANTQUE
FIN ;   /*  ---   fin fonction renverse    ---  */
/*-----------------------------------------------------*/
DEBUT  /*  ---  début programme inverse  ---  */
       ECRIRE ("Nombre désiré de saisies et d'affichages ? ") ;
       LIRE (fois) ;    /* contrôler la saisie */
       ECRIRE("Donnez votre texte : ") ;
       renverse (fois);
FIN ;    /*  ---    fin programme inverse     ---  */
```

Commentaires

La valeur de **fois** est recopiée dans **f**, qui est une **variable locale** à la fonction renverse. **Fois** est une variable globale à tout le programme. À l'appel de la fonction renverse, f vaut la valeur de fois, puis est décrémentée jusqu'à la valeur zéro. La valeur de fois est inchangée par renverse (la fonction a travaillé avec une **copie**).

On peut donner le même nom aux variables f et fois, puisqu'elle représente la même réalité, mais elles ne correspondent pas aux mêmes adresses en machine. En cas d'homonymie, c'est la dernière déclaration qui compte.
Si l'on a :

```
PROGRAMME transmet ;
VARIABLE fois    :   ENTIER ;
/*-----------------------------------------------------*/
FONCTION incrémente (fois: ENTIER) :VIDE;
DEBUT  /*  ---  début fonction incrémente   ---  */
        fois <- fois + 1 ;
        ECRIRE(fois)
FIN ;   /*  ---   fin fonction incrémente    ---  */
/*-----------------------------------------------------*/
DEBUT  /*  ---  début programme transmet  ---  */
        ECRIRE ("Donnez un nombre ") ;
        LIRE (fois) ;    /* contrôler la saisie */
        ECRIRE("J'incrémente : ") ;
        incrémente (fois) ;
        ECRIRE("Je reviens au programme principal : ") ;
        ECRIRE (fois) ;
FIN ;    /*  ---    fin programme transmet    ---  */
```

Ici la variable globale fois est inchangée. Le premier affichage donne la valeur de fois plus 1. Le résultat du calcul est perdu lorsque l'on sort de la fonction. C'est la valeur saisie initialement qui est affiché lors de l'exécution de la dernière instruction.

Variante

Une autre façon de voir les choses et de travailler avec «fois» comme **variable globale** seulement, en faisant attention de garder intacte sa valeur, si l'on l'utiliser plus tard.

```
PROGRAMME transmet ;
VARIABLE fois    :     ENTIER ;
/*-----------------------------------------------------*/
FONCTION incrémente :VIDE;
DEBUT  /*  ---  début fonction incrémente   ---  */
        fois <- fois + 1 ;
FIN ;   /*  ---   fin fonction incrémente    ---  */
/*-----------------------------------------------------*/
DEBUT  /*  ---  début programme transmet  ---  */
        ECRIRE ("Donnez un nombre ") ;
        LIRE (fois) ;    /* contrôler la saisie */
        ECRIRE("J'incrémente : ") ;
        incrémente(VIDE);
        ECRIRE (fois) ;
FIN ;    /*  ---    fin programme transmet    ---  */
```

Ici la valeur de fois a changé après l'appel à la fonction incrémente.

Le travail avec un maximum de variables en global n'est pas conseillé pour une question d'économie de place : seule les variables globales ont une durée de vie égale à celle du programme, et donc une certaine quantité de mémoire allouée pendant tout le programme.

Enfin, cette façon de procéder est propice aux **effets de bords** : des fonctions modifient incidemment des variables globales, ce qui n'était pas forcément voulu au départ. Les effets de bord obscurcissent la structure du programme et rendent l'algorithme plus difficile à comprendre.

Valeur de retour de la fonction

Une fonction sans valeur de retour (type VIDE) est assimilable à une instruction. Une fonction retournant une valeur est assimilable à une expression, dont le type est déterminé lors de la déclaration de la fonction.

Il est préférable de travailler avec des fonctions recevant des paramètres et renvoyant une valeur.

Exemple :

```
PROGRAMME transmet ;
VARIABLE fois    :     ENTIER ;
/*-----------------------------------------------------*/
FONCTION incrémente (f : ENTIER) :ENTIER;
DEBUT  /*  ---  début fonction incrémente   ---  */
        f <- f + 1 ;
        RETOUR (f )
FIN ;   /*  ---   fin fonction incrémente    ---  */
/*-----------------------------------------------------*/
DEBUT  /*  ---  début programme transmet  ---  */
        ECRIRE (“Donnez un nombre “) ;
        LIRE (fois) ;    /* contrôler la saisie */
        ECRIRE(“J’incrémente : “, incrémente (fois)) ;
        ECRIRE(“Valeur de fois après l’appel  “, (fois))
FIN ;    /*  ---    fin programme transmet    ---  */
```

La valeur de fois est inchangée.

C’est la valeur (fois + 1) qui est affichée lors de l’exécution de l’avant-dernière instruction.

Paramètres formels : passage par adresse ou par valeur ?

Passage par valeur (utilisé jusqu'à présent) :

- à chaque paramètre est associé un espace mémoire appartenant à l'espace alloué à la fonction.
- au moment de l'appel, il y a recopie de la valeur des paramètres effectifs dans l'espace mémoire associé aux paramètres formels correspondant.
- toute modification de la valeur d'un paramètre formel de type valeur est sans effet sur le paramètre effectif correspondant.
- les paramètres effectifs associés aux paramètres formels de type valeur peuvent être des variables, constantes ou des expressions.

Passage d'une adresse :

On utilise des **pointeurs.**
On fait précéder les déclarations des paramètres formels par ADRESSE
pour spécifier que la variable que l'on déclare va contenir l'adresse d'une autre variable.
Il est nécessaire que la variable contenant l'adresse possède un type adapté à celui de la variable dont elle doit contenir l'adresse.

Pour obtenir le contenu de la variable dont on posséde l'adresse, on utilise CONTENU sur la variable contenant l'adresse.

- il n'y a pas d'espace mémoire associé au paramètre dans l'espace alloué à la fonction.
- au moment de l'appel, le paramètre formel de type variable prend l'adresse du paramètre effectif correspondant.

Si l'on veut modifier directement un des arguments de la fonction on doit transmette l'adresse de cet élément. En fait, il y a une copie de l'adresse transmise en paramètre à la fonction.
C'est sur cette copie que va travaillé la fonction, mais grâce à cette copie de l'adresse, elle peut **directement** utiliser la variable indiquée et donc la **modifier**.

Pointeur

Un pointeur est une **variable** dont le contenu est l'adresse de celui d'une autre variable.

Pour définir une variable de type pointeur on utilise le mot clé POINTEUR_DE en plus du type de variable dont on veut stocker l'adresse.

Pour obtenir l'adresse d'une variable on utilise le mot clé ADRESSE appliqué sur l'identificateur de la variable dont on veut l'adresse.

Pour obtenir le contenu de la variable dont on possède l'adresse on utilise le mot-clé CONTENU sur la variable de type pointeur.

Exemple :

```
VARIABLE calcul : entier;
            acces_direct : POINTEUR_DE ENTIER;
calcul <-- 26;

acces_direct <-- ADRESSE calcul;

CONTENU acces_direct <-- 34;

ECRIRE (calcul);            /* affiche 34 */
```

On peut définir des pointeurs sur tout type de donnée, y compris ceux de structure :

```
TYPE STRUCTURE FICHE_PERSONNE :
            nom : TABLEAU_DE 10 CARACTÈRES;
            age : ENTIER;
      FIN;

VARIABLE personne_un : FICHE_PERSONNE;
          acces_direct : POINTEUR_DE FICHE_PERSONNE;

personne_un . nom <-- “Jean-Paul”;
acces_direct <-- ADRESSE personne_un;
(CONTENU acces_direct) . nom <-- “Jacques”;
ecrire(personne_un . nom); /*Affiche Jacques */
```

Exemple

```
PROGRAMME transmet ;
VARIABLE fois    :     ENTIER ;
/*-------------------------------------------------------*/
FONCTION incrémente (f : POINTEUR_DE ENTIER) :ENTIER;
DEBUT  /*  ---  début fonction incrémente   ---  */
        CONTENU (f) <- CONTENU (f) + 1 ;
        RETOUR (CONTENU ( f ))
FIN ;   /*  ---   fin fonction incrémente    ---  */
/*-------------------------------------------------------*/
DEBUT  /*  ---  début programme transmet  ---  */
        ECRIRE (“Donnez un nombre “) ;
        LIRE (fois) ;    /* contrôler la saisie */
        ECRIRE(“J’incrémente : “, incrémente (ADRESSE fois)) ;
        ECRIRE(“Après l’appel : “, fois) ;
FIN ;    /*  ---    fin programme transmet    ---  */
```

A l’appel de la fonction incrémente, on travaille sur l’entier correspondant à l’adresse de la variable globale “fois“. L’incrémentation de fois s’est effectuée, et la valeur initiale est perdue.

La façon de passer les arguments par adresse permet surtout de passer une série de valeurs, un tableau par exemple.

Dans la liste d’arguments de la déclaration d’une fonction, on peut mélanger les différents types de passage.

Par exemple :

```
FONCTION coupe-et-colle
        (à-coller : POINTEUR_DE TABLEAU_DE 80 CARACTÈRES,
        à-couper : TABLEAU_DE 80 CARACTÈRES ) : BOOLEEN;
```

Exemple de passage par adresse d'un tableau

Voici par exemple une fonction permettant de lire les éléments d'un tableau, et renvoyant le tableau initialisé.

```
FONCTION lectable (table : POINTEUR_DE TABLEAU_DE 10
                                                ENTIER) : VIDE ;
VARIABLE        i : ENTIER ;
DEBUT
        ECRIRE("Lecture des valeurs du tableau") ;
        POUR i VARIANT DE 1 A 10 FAIRE
                DEBUT
                        ECRIRE("Valeur  ", i," ?  ") ;
                        LIRE ((CONTENU table) [ i ])
                FIN
FIN ;
```

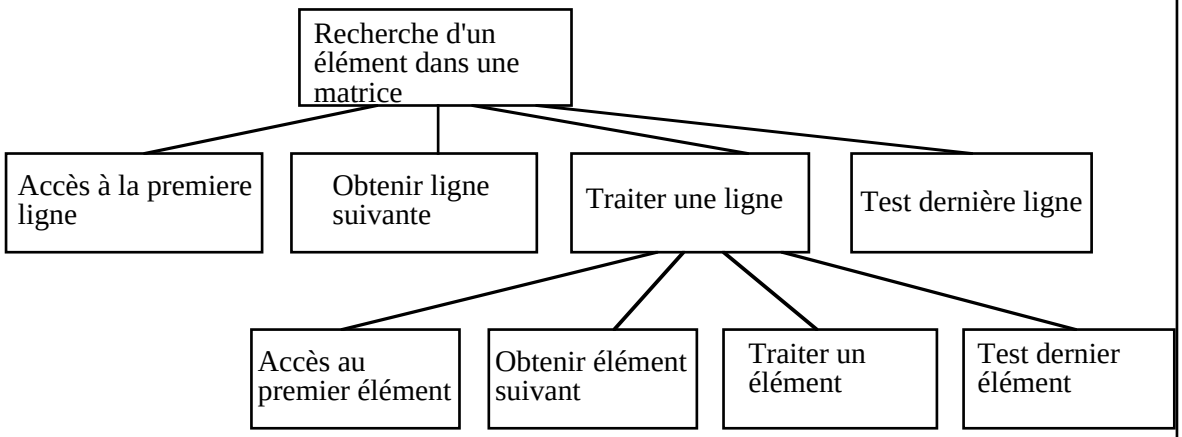
On peut avoir dans le corps du programme des appels de fonctions du type :

```
DEBUT  /*  ---  point d'entrée du programme  ---  */
        lectable(T) ;
        tritable (T) ;
        editable (T)
FIN ;
```

Un tel découpage en fonctions facilite la lecture et la maintenance des programmes.

Recherche d'un élément dans un tableau

Dans le cas d'une matrice rectangulaire, la recherche d'un élément consiste à parcourir toutes les lignes de la matrice et à traiter chacune d'elles. Le traitement d'une ligne de la matrice consiste à parcourir tous ses éléments et à les traiter. L'analyse descendante de la recherche d'un élément dans une matrice peut être résumée par le schéma suivant.



L'algorithme de recherche est écrit de la façon suivante :

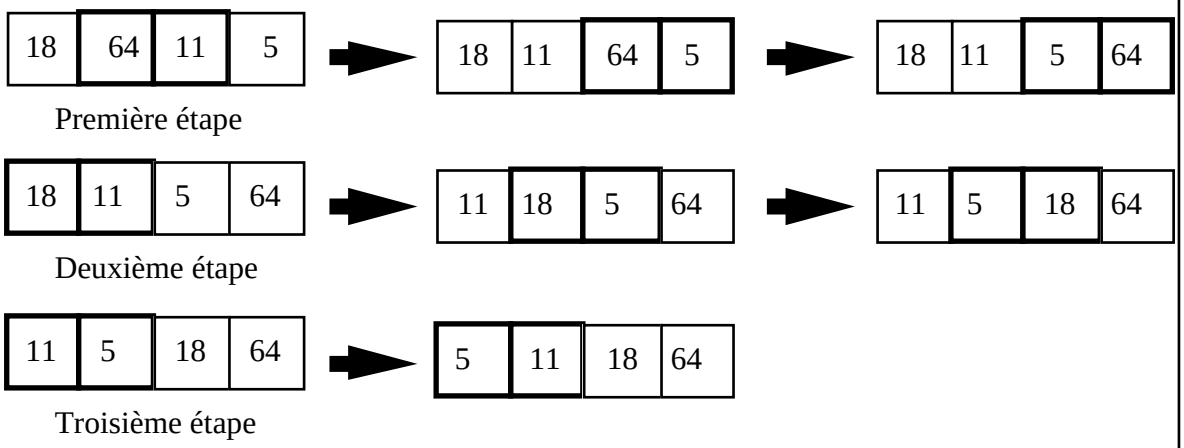
Fonction Recherche_d'un_élément_dans_une_matrice (ElémentRecherché, Trouvé, Ligne, Colonne)
    Initialisations
    Accès à la première ligne
    Fin <-- **faux**
    tantque non Fin **faire**
        Traiter ligne (ElémentRecherché, Trouvé, Ligne, Colonne)
        Obtenir ligne suivante
    fin tant que
Fin Fonction

Fonction Traiter ligne (ElémentRecherché, Trouvé, Ligne, Colonne)
    Accès au premier élément
    tantque Elément non final **faire**
        Traiter élément (ElémentRecherché, Trouvé, Ligne, Colonne)
        Obtenir élément suivant
    fin tant que
Fin Fonction

```
    Fonction Traiter élément (ElémentRecherché, Trouvé, Ligne, Colonne)
        si Matrice [NoLigne, NoColonne] = ElémentRecherché alors
            Trouvé <-- vrai
            Ligne <-- NoLigne
            Colonne <-- NoColonne
        fsi
    Fin Action

    Fonction Accès à la première ligne
        NoLigne <-- IndiceDébut1
    Fin Fonction

    Fonction Accès au premier élément
        NoColonne <-- IndiceDébut2
    Fin Fonction

    Fonction Elément non final résultat logique
        Elément non final <-- ((non Trouvé) et (NoColonne ≤ IndiceFin2))
    Fin Fonction

    Fonction Fin résultat logique
        Fin <-- (Trouvé ou (NoLigne > IndiceFin1))
    Fin Fonction

    Fonction Initialisations
        Trouvé <-- faux
    Fin Fonction
    Fonction Obtenir ligne suivante
        NoLigne <-- NoLigne + 1
    Fin Fonction

    Fonction Obtenir élément suivant
        NoColonne <-- NoColonne + 1
    Fin Action
```

Tri par la méthode des bulles.

Principe : le plus grand élément du tableau est propagé par permutations successives en fin de tableau, alors que les valeurs les plus légères remontent à la « surface », comme les bulles d'un liquide.



Traitement : (n connu entre 1 et 1000)

```
...
   VARIABLE       i , fin, temp : ENTIER ;
                  t : TABLEAU_DE 1000 ENTIER ;
....
TANTQUE (n > 0) FAIRE
   DEBUT
      i <-- 1 ;
      fin <-- 0 ;
      TANTQUE (i < n) FAIRE
         DEBUT
            SI t[i] >= t[i+1]
                  ALORS
                        DEBUT
                              temp <-- t[i] ;
                              t[i] <-- t [i + 1] ;
                              t[i + 1] <-- temp ;
                              fin <-- i
                        FIN ;
            i <-- i + 1
      FINTANTQUE
      n <-- fin ;
FINTANTQUE ;
/*  ---  édition des résultats pour vérification  --- */
```

Pointeurs et allocation mémoire dynamique

Toutes les variables étudiées jusqu'à présent étaient pour un programme donné, connues en nombre, nommées et déclarées en début d'algorithme.

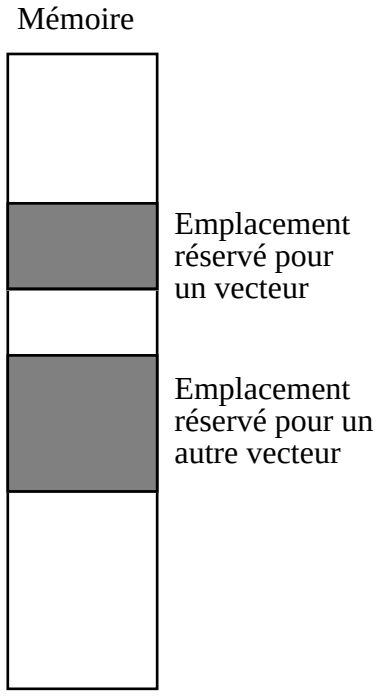
Elles occupent la mémoire de façon **statique** : une adresse en mémoire et une taille leur était alloués, pour la durée de vie du programme dans le cas de variables globales ou de la fonction dans le cas de variables locales.

Cas typique : les tableaux.

L'utilisation de tableaux en mémoire est très pratique pour le programmeur car elle permet de profiter de la souplesse de cette structure de données et des possibilités d'accès direct à ses éléments.

Certaines informations ont un nombre d'occurences indéfini : il faut alors réserver le maximum de place en mémoire.

À chaque déclaration de tableau, une zone en mémoire est réservée :



Ceci a deux inconvénients :

- D'une part la place réservée ne peut pas évoluer et elle doit donc être suffisamment large pour avoir des chances de permettre le traitement de la plupart des problèmes. Ceci a pour conséquence qu'une partie de la mémoire est réservée inutilement.

- D’autre part, la place réservée le reste jusqu’à la fin de l’exécution du programme, même si le tableau n’est plus utilisé par le programme.

L’utilisation de pointeurs permet d’apporter une solution à ces problèmes.

La possibilité de créer de nouvelles variables, pendant l’exécution du programme, est donnée par l’utilisation de variables **dynamiques**, gérées en fonction des besoins.

Les opérations sur les pointeurs permettent de créer en mémoire des objets, initialement vides, du type associé à chaque pointeur, d’accéder à ces objets et, éventuellement, de les détruire.

Pour un pointeur déclaré de la façon suivante :

NomPointeur POINTEUR_DE Type

les primitives de base sont les suivantes :

- **ALLOUER** (NomPointeur)

Cette instruction alloue en mémoire la place nécessaire pour la création d’un objet de type Type. Un objet vide de ce type est créé.

- **LIBÉRER** (NomPointeur)

Cette instruction libère la place occupée en mémoire par un objet de type Type, accessible à partir de NomPointeur.

En dehors de ces instructions spécifiques, les pointeurs peuvent être utilisées dans n’importe quelle instruction et permettent d’accéder à un objet de type Type créé par la primitive **Allouer**.

L’objet désigné par un pointeur peut varier après une instruction **Allouer** ou une instruction d’affectation.

ATTENTION : Il ne faut utiliser un pointeur (accéder à son contenu) que si l’on est sûr qu’il contient l’adresse d’une variable existante !
(définie de façon statique ou dynamique).

Règle : Toujours initialiser à NULLE un pointeur lors de sa déclaration.

Les fichiers

Toutes les variables rencontrées jusqu'à maintenant, qu'elles soient de type scalaire ou structurées, ont une existence en mémoire centrale pendant la durée d'exécution du programme dont elles font partie. Les résultats obtenus ne peuvent être réutilisés automatiquement dans un autre programme. Un fichier permet de mémoriser les données et de les transmettre à d'autres programmes.

Les enregistrement logiques sont disposés sur un support physique. Il s'agit d'établir une correspondance entre l'ensemble des enregistrements logiques et celui des emplacements physiques disponibles.
Différentes organisations sont possibles, qui dépendent des supports employés :

- l'organisation séquentielle
- l'organisation séquentielle indexée
- l'organisation relative

Différentes opérations sont possibles :

- l'accès direct
- l'accés séquentiel

Les traitements sur fichiers que nous verrons sont les suivants :

— création d'enregistrements :

  — écriture d'un nouveau fichier

  — écriture en fin de fichier

— lecture : lecture séquentielle

— mise-à-jour par lecture/écriture

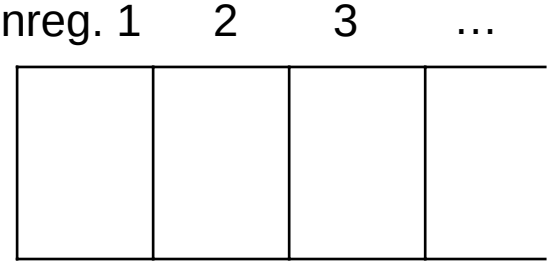
Organisation séquentielle

Les enregistrements logiques sont rangés consécutivement sur le support physique.
Très souvent on est amené à trier les enregistrements sur un critère logique pour traiter ces fichiers.
Les enregistrements peuvent être de longueurs variables.
La lecture du n-ième élément nécessite la lecture des (n-1) premiers enregistrements.

Exemple : fichier de type « texte », chaque enregistrement est de taille variable

Ceci est une phrase
Ceci est une autre phrase
…

Fichier de type « binaire », chaque enregistrement est de taille fixe (souvent correspondant à une variable de type structure)



Accès direct sur fichier à enregistrement de taille variable

Il s'agit d'accéder « directement » à un enregistrement en donnant son rang, sans lire les enregistrement qui précèdent.

Il est nécessaire d'associer à chaque enregistrement une « adresse » permettant de se positionner dessus directement.

L'association d'une adresse à un enregistrement doit être mémorisé (soit en mémoire, soit dans un nouveau fichier de type enregistrement de taille fixe).

Organisation relative

Cette organisation a l'avantage de permettre un accès séquentiel et un accès rapide aux enregistrements logiques du fichier.
L'adressage étant calculé en multipliant la longueur par la position relative de l'emplacement, il faut donc que tous les enregistrement aient la **même longueur**.

Par exemple, le n-ième enegistrement aura pour adresse l'adresse du début de fichier + (n - 1) * longueur de l'enregistrement.

Les traitements

Déclaration :

```
TYPE STRUCTURE  joueur :
                    nom :  TABLEAU_DE 10 CARACTÈRE ;
                    âge         : ENTIER ;
                    score       : ENTIER
                FIN ;

VARIABLE        fic 1:          FICHIER_DE REEL ;

                fic2 :          FICHIER_DE joueur ;
                bufjoueur :     joueur ;
```

Ouverture :

C'est à l'ouverture du fichier que se fait la relation descripteur/nom physique. C'est le traitement préliminaire à toute utilisation d'un fichier :

- le fichier n'existe pas et le programme crée le fichier
- le fichier existe, on se propose de l'ouvrir en lecture : le premier enregistrement est alors accessible
- le fichier existe ou est créé, on veut y écrire des informations : c'est sur le premier enregistrement que l'on écrira ou ré-écrira.
- on peut enfin ouvrir un fichier en mode « append » : on se positionne après le dernier enregistrement, près à écrire « à la suite ».

OUVRIR (fic, “nom du fichier”, lecture seulement) ;
OUVRIR (fic, “nom du fichier”, mode Lec/Ecr) ;
OUVRIR (fic, “nom du fichier”, mode ajout) ;

Fermeture :

C’est l’opération symétrique de l’ouverture. Il faut systématiquement fermer les fichiers non utilisés pour libérer des ressources d’entrées/sorties qui y sont associées.

FERMER(fic) ;

Lecture :

La **lecture séquentielle** permet de passer d’un enregistrement à l’autre à chaque lecture logique. Le contenu de l’enregistrement est recopié dans un « buffer » ou zone-tampon, et est alors accessible au programme. Chaque nouvelle lecture écrase le contenu du buffer par la valeur du nouvel enregistrement.

L’opération de lecture gère un indicateur de fin de fichier (EOF : End-Of-File), de type booléen. EOF est positionné à faux à l’ouverture du fichier (s’il n’est pas vide) et à vrai lors de la lecture du dernier enregistrement. Il faut absolument tester cet indicateur afin de ne pas risquer vouloir lire un enregistrement qui n’existerait pas en fin de fichier.

```
TANTQUE (NON EOF)  FAIRE .....
        LIRE(fic, bufjoueur) ;
        .....
```

La **lecture en accès direct**, pour un fichier d’enregistrements de taille fixe, permet d’accéder tout de suite à l’information désirée : il suffit de se positionner sur un enregistrement donné par son rang, ce qui a pour effet de déplacer le pointeur d’adresse associé au fichier, et de lire tout de suite après l’enregistrement désiré.

```
Positionnement :      ALLER_A(rang_enregistrement)
La lecture :          LIRE(fic, bufjoueur) ;
Puis :      SI (bufjoueur.nom = nom_cherché)
                              ALORS... /* enregistrement trouvé */
```

En accès direct, le test sur EOF n’est pas utile.

L’écriture :

Sur les fichiers à organisation séquentielle, l'écriture des enregistrements se fait en suivant, du premier jusqu'au dernier. Une insertion d'enregistrement n'est pas possible.

Par contre on peut se positionner en fin de fichier, soit en lisant tous les enregistrements, soit en ouvrant le fichier en mode append, et ensuite écrire la série d'enregistrements qui complètent le fichier.

ECRIRE (fic, bufjoueur) ;

Une mise à jour doit pouvoir permettre de ré-écrire un enregistrement en le modifiant : le score du joueur peut avoir changé par exemple. La ré-écriture opère sur l'enregistrement qui vient immédiatement d'être lu (elle n'est possible que dans le cas d'enregistrement de taille fixe).

```
LIRE(fic, bufjoueur) ;
/* modification de bufjoueur.score */
RE-ECRIRE (fic, bufjoueur) ;
```

L'écriture n'utilise pas EOF.

L'étape de lecture est nécessaire avant toute ré-écriture.

Lors de ces opérations d'entrées-sorties sur fichiers, le programme passe la main au système d'exploitation qui gère les périphériques associés. Celui ci tente de réaliser l'opération demandée et renvoie un **code** permettant de s'assurer du succès de l'opération, et le cas échéant, de traiter le cas d'erreur.
L'étude de ces **codes** se fera de façon plus fine lors de la programmation.

Création de fichier :

```
OUVRIR(fic, mode création) ;
REPETER
        DEBUT
                /* acquisition des données */
                ECRIRE("Donnez le nom du joueur ") ;
                LIRE(bufjoueur.nom) ;
                ECRIRE("Donnez l'âge du joueur ") ;
                LIRE(bufjoueur.âge) ;
                bufjoueur.score <-- 0 ;
                /*   écriture dans le fichier : */
                ECRIRE(fic, bufjoueur) ;
                ECRIRE("Y a-t-il encore un joueur ? ") ;
                LIRE(réponse)
JUSQUA (réponse <> 'O') ;
FERMER(fic) ;
```

Le fichier fic est créé.
Dans un programme, à l'ouverture, une correspondance doit être établie entre le **descripteur**, et le nom physique.
fic <-- open("JOUEUR.DATA" , mode lecture).
puis LIRE(fic, bufjoueur);

sortie dans un fichier :

```
OUVRIR(fic, mode lecture seulement) ;
/* lecture initiale, positionne EOF */
LIRE(fic, bufjoueur) ;

TANTQUE (NON EOF(fic)) FAIRE
        DEBUT
                LIRE(fic, bufjoueur);
                ECRIRE(bufjoueur.nom) ;
                ECRIRE(bufjoueur.score) ;
        FIN ;
FERMER(fic) ;
```

Extension de fichier :

```
OUVRIR(fic, mode append) ;
```

```
REPETER
        DEBUT
                /* acquisition des données */
                ECRIRE(“Donnez le nom du joueur “) ;
                LIRE(bufjoueur.nom) ;
                ECRIRE(“Donnez l’âge du joueur “) ;
                LIRE(bufjoueur.âge) ;
                bufjoueur.score <-- 0 ;
                /*   écriture dans le fichier : */
                ECRIRE(fic, bufjoueur) ;
                ECRIRE(“Y a-t-il encore un joueur ? “) ;
                LIRE(réponse)
JUSQUA (réponse <> ‘O’) ;
FERMER(fic) ;
```

Bibliographie

— L'algorithmique, de la pratique à la théorie
G .Chaty et J Vicard
Editions Cedic/Fernand Nathan 1985

— Initiation à l'analyse et à la programmation
J.-P. Laurent
Editions Dunod informatique 1982

— Exercice commentés d'analyse et de programmation
J.-P. Laurent et J. Ayel
Editions Belin 1985

— Algorithmique et représentation des données
M. Lucas, J.P. Peyrin, P.C. Scholl
Editions Masson 1985

— Algorithmes en langage C
Robert Sedgewick
Editions InterEditions 1991.