Pré-requis ---- Les notions de base de l'algorithmique

A. Introduction

1. Éléments caractérisant l'algorithmique séquentielle

Dans le chapitre 1 nous avons introduit la notion d'algorithme. Un algorithme est **un schéma de résolution d'un problème**. Ici nous définissons ce qui caractérise un algorithme et son écriture.

Trois éléments caractérisent généralement tout algorithme séquentiel :

1. **un glossaire** spécifiant les "données" utilisées par l'algorithme (pour une recette, il s'agit des ingrédients). En français, le glossaire correspond en général aux *noms communs* (œuf, farine, etc.).
2. **un ensemble d'instructions élémentaires** (battre les oeufs, réchauffer de l'eau, ...). En français, la liste d'instructions correspond aux *verbes*.
3. **un ordre sur l'exécution** (d'abord, ensuite, ...). En français, l'ordre est exprimé à l'aide de *conjonctions*.

Le cœur de tout algorithme réside dans le choix de **l'ensemble des instructions élémentaires**. Cet ensemble peut être plus ou moins riche : très riche, c'est mettre à disposition un nombre important d'instructions. Pauvre, à l'inverse, c'est n'avoir que très peu d'instructions (c'est le cas d'algorithmes étudiés pour être traduits en *assembleur* par exemple). Pour cette raison, l'algorithmique se trouve toujours plus ou moins "liée" à la richesse du langage de programmation que l'on utilisera pour la résolution du problème. On parle de langage *support*.

Toutefois, l'algorithmique tente d'éviter - au plus possible - un formalisme se rapprochant trop d'un langage de programmation particulier. Ceci peut être difficile dans certains cas, car bien souvent l'algorithme même dépend de la richesse du langage de programmation utilisé. C'est ce qui explique pourquoi il n'existe pas réellement de formalisme universel en algorithmique. "Chacun écrit ses algorithmes un peu à sa façon", mais néanmoins toujours en adoptant une formulation mathématique de rigueur.

L'écriture algorithmique que nous adopterons dans ce cours n'est donc de fait pas universelle. Mais, même si d'autres établissements emploient d'autres écritures algorithmiques, on retrouve toujours des points communs et une formulation globalement similaire, avec à peu de choses près les mêmes instructions élémentaires.

2. Instructions élémentaires

Les instructions utilisées par un algorithme correspondent à des commandes informatiques de base (cad à des instructions que peut exécuter un processeur) :

Généralement, on distingue trois types d'instructions élémentaires :

- traitement des données (opérations mathématiques, calculs, etc.) ;
- structures de contrôles ;
- communication avec les périphériques (affichage, lecture, etc.).

Les opérations de base sont en général les suivantes :

calculs	**ordonnancement**	**e/s**
+, -, *, /, mod	si < cond > alors	saisir / afficher
sin, cos, tan, exp, etc.	sinon	lire / écrire
affectation	tant que < cond> faire	ouvrir
	répéter ... jusqu'à <cond>	fermer
	pour chaque... etc....	etc.....

Afin de traiter une information (une **donnée**), il faut pouvoir **l'identifier**. On lui attribue donc un nom (un peu comme en mathématiques, on parle de x, y, etc.). La donnée outre son nom, a également une valeur (x vaut 3 par exemple). Notons que comme en mathématiques, les données traitées en informatique ont toujours un type particulier : il s'agit d'un entier, d'une matrice, d'un texte, d'une image, d'un son, etc. Pour les différentes opérations, les types ne peuvent être mélangés.

Les structures de contrôle permettent de gérer l'ordonnancement de l'algorithme. Selon une condition donnée (un état qui dépend par exemple de l'utilisateur du programme), l'exécution n'est pas la même : exemple, "si dimanche alors dormir jusqu'à 12h sinon debout à 6h !"

Les opérations d'entrée / sortie (e/s ou, en anglais, i/o, pour input/output) permettent de communiquer avec l'utilisateur du programme ou bien avec les périphériques qui sont connectés sur le bus (imprimante, disque, réseau, haut-parleurs, écran, etc.).

Voici un exemple d'algorithme permettant de calculer l'aire d'un cercle. Le programme demande à l'utilisateur le rayon, puis affiche l'aire en fonction de ce rayon :

```
// Calculer l'aire d'un cercle à partir du rayon

programme AireCercle
début
        avec        rayon :   réel        // rayon du cercle saisi par l'utilisateur
                    aire :    réel        // aire du cercle calculée grâce à rayon
```

afficher « Saisir le rayon du cercle »
saisir rayon
*aire := 3.14159*rayon*rayon*
afficher « L'aire du cercle vaut : », aire
à la ligne
fin AireCercle

On note sur cet exemple :

- l'existence d'un glossaire où sont déclarées les données *rayon* et *aire* ;
- un ensemble d'instructions élémentaires : *afficher*, *saisir*, *affectation* (symbolisée par ":="), *multiplication* (symbolisée par "*") et *à la ligne* ;
- un ordre sur les instructions : on lit de haut en bas (et de gauche à droite).

3. Développer un logiciel

Une fois le problème posé (en définissant un **cahier des charges**), et une solution algorithmique trouvée (**par analyse**), il faut réaliser le logiciel. La réalisation se fait en traduisant l'algorithme (écriture pseudo-française) en un langage de programmation, en utilisant des outils systèmes (compilateur, interpréteur, atelier de génie logiciel, etc.). Cependant, le programme n'est pas uniquement constitué d'algorithmes.

De nos jour, un logiciel se compose à :

- environ 75% d'une interface souvent graphique 2D (widgets, fenêtres, etc.) ;
- environ 20% de traitements liés à la gestion des erreurs et des cas particuliers ;
- environ 5% d'algorithmes proprement dits ;

Lors de la phase d'analyse (définition de l'algorithme), on ne se préoccupe pas, en général, de l'interface. La gestion des exceptions n'est également qu'abordée de manière marginale. Cependant, il est tout à fait essentiel de considérer dès le départ (dès l'analyse), tous les cas particuliers. Ces derniers doivent être impérativement traités.

Ici, nous nous préoccupons donc surtout des 5%, représentant le *cœur* de tout programme ou logiciel (le reste étant souvent appelé le *squelette*). Pour ce qui concerne l'interfaçage, nous ne considérons dans un premier temps que des saisies et affichages élémentaires (pas de *graphisme ni de fenêtrage* !)

4. Langages de programmation

Il existe différentes catégories de langages de programmation séquentielle :

• interprétés => l'exécution se fait au cours de la lecture;
• de compilation => le prog. est traduit une fois pour toute en langage machine.

Le programmeur procède en trois étapes pour les langages de compilation :

```
          compilation              édition de lien
source ----------->  objet  ----------->  exécutable
```

Certain langages peuvent être mixtes (ils utilisent alors un processeur virtuel): Java ou Python.

Il existe également différentes catégories de langages de programmation, comme par exemple les langages dits :

• impératifs comme Fortran, C, Pascal ;
• fonctionnels ou de lambda calcul comme Lisp ;
• logiques comme Prolog ;
• modulaires ou plus récemment orientés objets comme Smalltalk, C++, Java ;

En dehors d'une écriture séquentielle ou parallèle, on distingue également la programmation dite **événementielle** liée aux interfaces graphiques : le programme réagit en fonction d'évènements, comme le mouvement de la souris par exemple. Toutefois, ce style de programmation se rapproche de la programmation séquentielle habituelle et ne pose pas de réel problème d'un point de vu algorithmique. Notons que ce style est devenu tout à fait incontournable lors d'un développement sur une plate-forme de type Windows ou X11.

B. La variable informatique

L'essentiel du travail d'un ordinateur consiste à traiter des **données**. Comme pour tout objet de la vie courante, il faut attribuer un nom à chacune de ces données, de façon à pouvoir l'identifier sans ambiguïté.

La variable matérialise une donnée. Elle a un **nom**, un **type**, une **valeur** et une **zone mémoire** pour stocker sa valeur.

1.1 Le nom d'une variable informatique

Le nom de la variable permet de **l'identifier** et de lever toute ambiguïté, car il est obligatoirement unique dans un même corps de programme. Le nom est composé d'une suite de caractères ou chiffres (sans commencer par un chiffre). Il doit être choisi de façon à éclaircir l'utilité de la variable au sein de l'algorithme (comme le prénom d'une personne indique son sexe).

Dans l'exemple précédent, les variables avaient les noms *rayon* et *aire*. Ces deux noms indiquent parfaitement leur rôle respectif au sein de l'algorithme. Mais, le nom à lui seul, est parfois insuffisant. Il est donc préférable de donner de plus amples précisions sous la forme de **commentaires**.

Voici des exemples de noms de variables valides et invalides :

Rayon	x	// trop peu précis
Couleur	2plan	// ne pas commencer avec un chiffre

Nombre1 Est_valide Nombre2 Le_mot_le_plus_long	vert+ // pas de symbole (sauf le "_")

1.2 Le type d'une variable informatique

Le type (entier, réel, matrice, vecteur, texte, image, son, ...) permet d'identifier la **nature** de la donnée. Il permet à l'ordinateur de spécifier (comme en mathématique) l'ensemble des *opérations* qui sont permises : par exemple, des entiers ou réels peuvent être additionnés, un texte imprimé, une image affichée. Notons qu'en mathématiques les variables sont également typées, même si le type n'est pas toujours explicitement donné. En effet, il reste souvent implicite au contexte. Certains langages de programmation (par exemple Smalltalk) n'obligent pas le programmeur à donner un type unique au départ. Toutefois, dans notre cas le type d'une variable sera toujours spécifié en tête de l'algorithme (dans le glossaire), et il sera unique tout le long de ce dernier.

Le type permet également d'associer un codage binaire à la donnée. En informatique, il existe un certain nombre de types prédéfinis que l'on appelle les **types de base**. Nous les verrons dans la suite de cette section. Les autres types de données, par exemple une image ou un son, sont des **types composés** à l'aide des types de base. Ils sont définis par le programmeur et ajoutés au langage, en particulier grâce à la notion de **structure** et plus récemment **d'objet**.

1.3 La valeur d'une variable informatique

Toute variable informatique a **toujours et à tout moment** une valeur **finie**. Des écritures comme en mathématiques, du type : *« quelque soit x »* ne sont jamais possibles en informatique. La valeur est **finie**, il n'est donc pas non plus possible d'écrire : *« avec E : ensemble des entiers naturels »*. En effet, cet ensemble a un cardinal infini. Si la valeur n'est pas indiquée par le programmeur au moment de la déclaration, c'est une valeur par défaut qui sera attribuée.
Attention, car pour certains langages, cette valeur par défaut est choisie de façon purement aléatoire !

Il est nécessaire d'initialiser une variable avant de l'utiliser. La saisie est une forme particulière d'initialisation. Evidemment la valeur d'une variable n'est pas figée. Elle peut évoluer (changer) au cours de l'algorithme, d'où le nom de *variable*. Notons, qu'il est possible de définir pour un algorithme donné des constantes, par opposition aux variables (sera vu en TD et TP).

Dans l'exemple précédent, la variable *rayon* possède une valeur avant même la saisie. Cette valeur est aléatoire (pas nécessairement 0 !). Il est donc interdit de l'utiliser avant de l'avoir saisie. La variable *aire* est initialisée par affectation d'une expression arithmétique. Avant cette affectation, elle possède une valeur aléatoire (en fait, plutôt indéterminée).

Note : Ne pas initialiser la valeur d'une variable avant de l'utiliser est une source majeure d'erreur de programmation. Il est formellement interdit d'utiliser une variable

avant de l'avoir initialisée ! L'initialisation s'effectue soit sous forme de saisie, soit en affectant une valeur directement à la déclaration, soit encore, en affectant une valeur qui peut être calculée à partir d'autres variables toutes (sans exception) déjà initialisées, c'est le cas de la variable *aire*.

1.4 La zone mémoire d'une variable informatique

La zone mémoire correspond à l'endroit "physique" où sera stocker la valeur de la variable en utilisant un codage binaire (cf. cours **d'architecture**).

Astuce : Une variable peut être vue (intuitivement) comme un *tiroir* dans lequel on place quelque chose. Ce tiroir n'est jamais vide et contient toujours le même genre de donnée (un nombre entier, une image, etc.). On peut placer à tout moment une nouvelle donnée dans ce tiroir. Chaque tiroir a un nom (une étiquette) qui permet de l'identifier.

Le nombre de tiroirs **n'est pas illimité**. Il dépend de la quantité de mémoire disponible sur la machine. Il dépend également de la nature de l'information : un tiroir de type "image" utilise plus de mémoire qu'un tiroir de type "nombre entier".

2. Instructions élémentaires

2.1 L'affectation

Cette opération permet d'attribuer à une variable une certaine **valeur**. Pour une variable de type entier, appelée par exemple *nb_entier*, on peut écrire :

```
Nb_entier := 3
```

Ceci se lit : *« affecter à la variable nb_entier la valeur 3 »*, et correspond, en reprenant l'exemple des tiroirs, aux actions suivantes : *« ouvrir et remplacer dans le tiroir nommé nb_entier, la valeur qui s'y trouve actuellement par la valeur 3. L'ancienne valeur est oubliée et disparaît complètement »*.

La valeur affectée dépend évidement du type de la variable. Il n'est pas possible d'affecter à une variable de type entier une valeur réelle : par exemple, il n'est pas possible d'affecter à *nb_entier*, la valeur 3.1415.

On se propose ici d'"encoder" l'affectation par ":=", pour ne pas confondre cela avec le "=" des mathématiques, qui n'a strictement rien à voir, même si beaucoup de langages de programmation utilisent ce symbole pour encoder l'affectation.

Dans le cas numérique, la valeur affectée peut correspondre à une expression arithmétique pouvant faire intervenir d'autres variables, des symboles (+, -, *, etc.) et des parenthèses.

Exemples:

```
x := 3          y := x+1          a := a+2
z := -2         toto := 3.14159   max := 6.0
```

Il est possible d'affecter directement une valeur constante donnée à une variable au moment de sa déclaration. On parle alors **d'initialisation**.

avec ma_var:=12 :entier

Certains langages n'obligent pas à déclarer explicitement le type d'une variable celui-ci étant implicite à l'affectation. C'est le cas du langage Python par exemple.

Certains langages permettent également de faire des affectations multiples:

a:= b:= c:= 1
ou
a,b,c := 1,2,3

2.2 La lecture / écriture de valeurs au travers d'une console

Une console est tout simplement un couple écran / clavier. Avec une console, l'écran n'est pas graphique, c'est-à-dire qu'il ne permet que d'afficher du texte. La lecture et l'écriture sont les opérations de base qui permettent de communiquer avec l'utilisateur du programme au travers d'une console.

2.2.1 La lecture

saisir ou bien lire permettent de lire une valeur au clavier et d'affecter celle-ci à une variable dont le nom suit directement la commande, exemple :

lire ma_var

Cette instruction exécute automatiquement les fonctionnalités suivantes :

- bloquer le programme
- lire des caractères au clavier et attendre un retour chariot (touche *entrée*)
- convertir les caractères saisis en valeur selon le type de la variable *ma_var*
- affecter cette valeur à la variable concernée (ici *ma_var*)
- débloquer le programme pour qu'il puisse continuer.

Il faut toujours fournir à la commande saisir des noms de variables. En effet, *saisir x+1* n'a pas de sens !

2.2.2 L'écriture

Cette instruction correspond à l'inverse de la saisie. afficher ou écrire permet donc d'afficher des valeurs à l'écran texte, exemple :

Afficher "La variable vaut", (ma_var)

Cette instruction exécute les fonctionnalités suivantes :

- afficher le texte *“La variable vaut”*
- convertir la valeur de *ma_var* en codes caractères
- afficher ces caractères à l'écran

Afficher peut traiter directement des expressions arithmétiques :

Afficher "la variable x vaut:", (x), " et son carré ", (x)*(x)

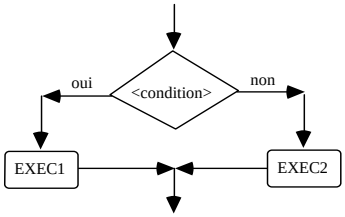
Les " " permettent de définir un texte (une chaîne de caractères). Les parenthèses autour de la variable indiquent que l’on prend son contenu. Elles sont optionnelles.

2.3 Branchements conditionnels simples

Généralement, un programme s'exécute de façon différente selon un contexte donné. Le contexte dépend, la plupart du temps, des choix et actions de l’utilisateur. De façon formelle, un contexte est spécifié par le biais d’expressions mathématiques **booléennes**. En algorithmique, on utilise la commande de **branchement conditionnel** “si alors sinon” pour tenir compte d’un contexte donné.

La séquence algorithmique prend la forme suivante :

si <condition> alors
  | | EXEC1
sinon
  | | EXEC2
finsi



Les expressions booléennes permettant d’exprimer une **condition** sont étudiées au chapitre suivant. Elles utilisent des opérateurs de comparaison entre variables numériques (entiers et réels) tels que =, <, >, ≤, ≥, ≠ ainsi que des opérateurs booléens (non, et, ou).

2.4 Exemple d'algorithme avec branchement conditionnel

Calculer le maximum entre deux nombres :

```
Programme Maximum2nombres
Début
  avec        max :          réel    // le maximum entre deux nombres
              nb_a, nb_b :   réel    // les deux nombres à saisir et comparer

    saisir nb_a, nb_b
    si a>b alors
        max := a
    sinon
        max := b
    finsi
    afficher "le maximum vaut ",max
fin Maximum2nombres
```

3. Les types de base

3.1 Booléen

En algorithmique, la déclaration d'une variable *b* de type booléen se fait de la manière suivante :

avec b : booléen.

On peut affecter à une variable de type booléen deux constantes : *vrai* ou *faux*
exemple: *b:= faux*

Les opérations booléennes sont les suivantes: *non*, *et*, *ou*

• le non permet d'inverser la valeur du booléen, vrai devient faux et vis versa.
exemple: *b:= non a*
si a vaut vrai alors b vaudra faux, par contre si a vaut faux alors b vaudra vrai.

• le et permet de vérifier si deux booléens sont en même temps vrais
exemple: *c := a et b*
c n'est vrai que si a et b sont tous deux vrais, autrement c vaut faux.

• le ou permet de vérifier si l'un des deux booléens vaut vrai.
exemple: *c := a ou b*
c ne vaut faux que si a et b valent faux tous les deux, autrement c vaut vrai.

En algèbre de Boole, on note souvent le ou par +, le et par *, le non par une barre, le vrai par 1 et le faux par 0 (c'est une écriture dite algébrique).

Propriétés élémentaires (non démontrées, cf. cours de **mathématiques**):

a*0=0 ; a*1=a ; a*a=a; a+0=a ; a+1=1 ; a+a=a ; a*$\bar{a}$=0 ; a+$\bar{a}$=1
a*(b+c)=(a*b)+(a*c) ; a+(b*c)=(a+b)*(a+c) => distributivité
$\overline{a+b}=\bar{a}*\bar{b}$; $\overline{a*b}=\bar{a}+\bar{b}$ => Morgan
a+ab = a =>absorption

Il existe des opérateurs de comparaison sur les valeurs numériques entières et réelles permettant de calculer des expressions booléennes: =; <; >; ≤; ≥; ≠

ex: (a<b) et (b<c/2) où a, b et c sont tous des entiers ou tous des réels.

Exemple concret : Exprimer la condition de réduction du prix d'entrée, sachant qu'elle est accordée aux enfants de moins de 18 ans, aux étudiants et aux personnes âgées de 65 ans et plus.

Solution : *(age<18) ou est_étudiant ou (age≥65)* ,
avec age : entier
est_étudiant : booléen

Des affectations du type : *b:= x<y ou s* , avec b, s : booléens, x, y : entiers (ou réels) sont évidemment possibles.

3.2 Entier

Comme nous l'avons déjà vu, la déclaration d'une variable *a* de type entier se fait de la façon suivante :

avec a : entier

On peut affecter à une variable de type entier une constante numérique, comprise dans un certain intervalle. Les bornes dépendent du nombre d'octets avec lequel est codé l'entier (cf. cours **d'architecture**) :

```
x:= 25
y:= -32
```

Les opérations permises sur les entiers sont les opérations mathématiques usuelles, c'est-à-dire: +, -, *, / (division Euclidienne) et mod (modulo, reste de la division).

```
ex:   x := a mod 2          => indique la parité
      y := a*4+1
      z := 3/10             => attention vaut toujours 0!
```

Note: la plupart des langages permettent de définir la taille des entiers (courts ou long). Par exemple, pour des entiers longs en Python, on ajoute un L derrière le chiffre.

3.3 Réel

La déclaration d'une variable *r* de type réel se fait de la façon suivante :

avec r : réel

On peut leur affecter des constantes numériques comprises dans un intervalle. En réalité ce ne sont pas des réels au sens mathématique du terme, mais uniquement des nombres décimaux (pas même fractionnaires). Les nombres réels sont dits en informatique à *virgule flottante* (voir cours **d'architecture**).

```
ex:   x:=3.1415
      w:=5*10^6
```

Les opérations usuelles sont : +, -, *, /, avec en plus des opérations et fonctions de base comme: sqrt, pow, cos, sin, tan, exp, log, etc.

```
ex:   valeur := pow(x,cos(a+b*sqrt(z)))        // équivaut à x^cos(a+b*sqrt(z))
```

Dans le cas d'opérations comme la division (y compris pour les entiers) ou la racine carrée, il faut faire attention au domaine de définition, afin d'éviter des erreurs fatales

(racine carrée d'un nombre négatif par exemple). Une erreur fatale conduit à l'arrêt immédiat du programme (si cela arrive, on parle de **"bug"** dans le programme).

3.4 Caractère

La déclaration d'une variable *c* de type caractère se fait de la manière suivante :

avec c : car

On peut affecter des constantes à une variable de type caractère.
ex: c:= 'A'
c:= '='
s:= 'c' // à ne pas confondre avec s:=c ! ! !

Le caractère placé entre les deux apostrophes est unique (contrairement à la chaîne de caractères que nous avons vu au niveau de l'affichage). L'écriture suivante n'est donc pas autorisée : *c:='toto'*.

En fait un caractère est un entier "caché" derrière un codage particulier. Le codage le plus usuel est le codage dit *ASCII*. Par exemple le caractère 'A' est codé par le chiffre 65, le 'a' correspond à 97, le '0' à 48, etc.

Il n'y a pas vraiment d'opérations sur les caractères sauf dans certains cas le + ou le -. Par exemple, *c:= 'a'+1* affecte à la variable *c* le caractère suivant la lettre 'a'. Il s'agit de 'b'. Mais pas tous les langages de programmation ne supportent ce type d'écriture.

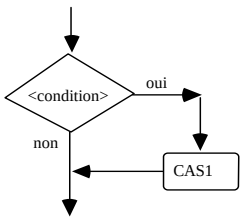
C. Les structures de contrôle

1. La sélection simple

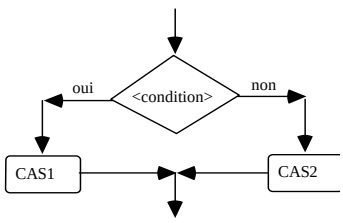
1.1 Le si alors sinon

L'exécution d'un programme dépend d'un contexte donné. Ce contexte est exprimé à l'aide **d'expressions booléennes**.

si <cond> alors
| CAS1
[sinon
| CAS2]
finsi



cas simple sans sinon



cas complet

Les expressions booléennes sont obtenues par des variables booléennes, par des opérations logiques (non, ou, et) ou par des opérations relationnelles =, ≠ etc.

Deux exemples simples, sans *sinon*:

```
si age<18 alors
        | afficher "mineur"
finsi
```

```
si n mod 2 = 1 alors
        | afficher "impair"
finsi
```

Un exemple avec *sinon*:

```
si n mod 2 = 1 alors
        | afficher "impair"
sinon
        | afficher "pair"
finsi
```

En guise d'exercice, commenter les expressions suivantes:

si non (age < 18) alors , si non (couleur=vert ou couleur=bleu) alors
si v=3 ou v≥2 alors , si k=1 ou (k=2 et k=3) alors
si r≤3.5 et r≥3.0 alors

Attention, des écriture comme celles qui suivent ne sont pas valides:

si 1≤x≤8 alors	il faut écrire	si 1≤x et x≤8 alors
si x=1,2 ou 3 alors	il faut écrire	si x=1 ou x=2 ou x=3 alors

1.2 Imbriquer des conditions

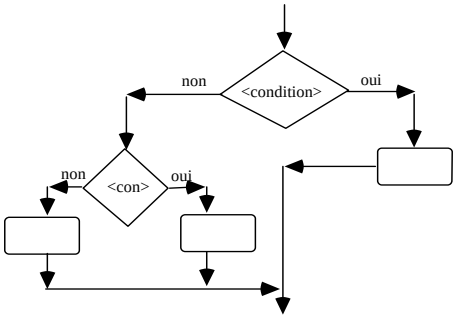
Il se peut qu'en traitant un cas particulier, il devienne nécessaire de faire une seconde sélection.

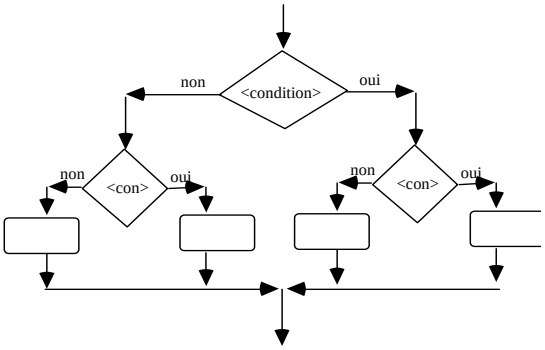
Exemple: choix d'un tarif réduit. 0-18 ans tarif -50%, 19-27 ans -20% et plus de 28 ans plein tarif.

```
Programme tarif
début
        avec    age:    entier,         // age de la personne
                prix:   réel            // prix à payer

        saisir age, prix
        si age≤18 alors
                prix:=prix*0.5
        sinon
                si age≥19 et age≤27 alors
                        prix:=prix*0.8
                finsi
```

```
        finsi
        afficher prix
fin tarif
```

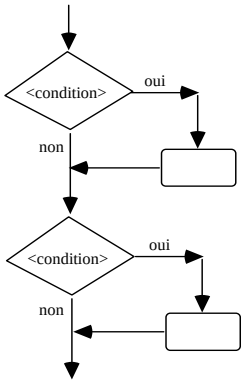
Dans ce cas, on parle d'écriture **imbriquée**:





Autre façon d'écrire (non imbriquée):

```
Programme tarif
Début
        avec age:entier, prix:réel

        saisir age, prix
        si age≤18 alors
                prix:=prix*0.5
        finsi
        si age≥19 et age≤27 alors
                prix:=prix*0.8
        finsi
        afficher prix
fin tarif
```



On parle d'écriture **linéaire**

De façon générale, il est permis d'imbriquer autant de fois que voulu. Le facteur le plus important reste la **lisibilité du programme**. En général, une écriture imbriquée est plus efficace, alors que l'écriture linéaire est plus claire.

2. La sélection multiple

Selon un certain nombre de cas donnés, les actions sont différentes:

```
selon x faire
        cas 1 :   TRAITEMENT1
        cas 2 :   TRAITEMENT2
        ...
        défaut :   TRAITEMENT PAR DEFAUT (si aucun des cas précédents n'a abouti)
fin faire
```

En général, le test se porte sur une variable ou une expression. Chaque cas ne correspond normalement qu'à une seule valeur et non à des ensembles de valeurs. Cependant, il arrive d'écrire: *cas 1 à 20* par exemple. Mais pas tous les langages de programmation, ne permettent une transcription directe de ce type d'écriture. Dans certains cas, il faut alors se ramener à des *si* habituels. L'usage de ou, non et et est interdit, il en est de même pour les opérateurs de comparaison. **Des écritures comme : cas <18 ne sont généralement pas permises**.

Exemple tarif:

```
Programme tarif
début
        avec age:entier, prix:réel

        saisir age, prix
        selon age faire
                cas 0 à 18 : prix:=prix*0.5
                cas 19 à 27 : prix:=prix*0.8
                défaut : rien
        fin faire
        afficher prix
fin tarif
```

Attention: pas tous les langages de programmation ne supportent ce type de branchement conditionnel.

3. Les boucles conditionnelles

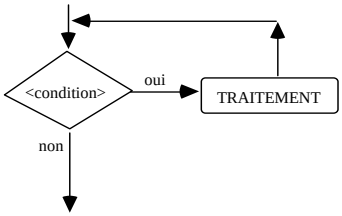
3.1 Itération simple

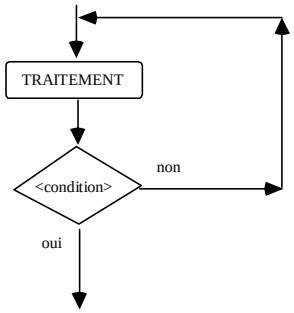
Il arrive fréquemment que des parties de programme doivent se répéter un certain nombre de fois. Le nombre de répétitions (**itérations**) est fonction d'une condition donnée. Il existe deux types de boucles conditionnelles :

```
tant que <cond> faire
    TRAITEMENT
fin faire
```

```
répéter
    TRAITEMENT
jusqu'à <cond>
```

Attention la sémantique n'est pas la même dans ces deux cas (l'un s'arrête lorsque la condition n'est plus vérifiée, l'autre lorsqu'elle le devient). Les conditions sont exprimées, comme pour les branchements conditionnels, à l'aide d'une expression booléenne.





La différence entre les deux approches est la suivante : dans le cas du tantque le TRAITEMENT peut ne pas être effectué du tout, alors que dans le cas du répéter le TRAITEMENT s'effectue toujours au moins une fois.

Exemple d'utilisation d'une boucle conditionnelle. Afficher les nombres de 1 à *n*, avec *n* saisi par l'utilisateur :

```
Programme AffNombres
Début
        Avec n, i :entier

        Saisir n
        i := 1
        tant que i<=n faire
                afficher i
                i := i+1
        fin faire
fin AffNombres
```

```
Programme AffNombres
début
        avec n, i :entier

        saisir n
        i := 1
        répéter
                afficher i
                i := i+1
        jusqu'à i>n
fin AffNombres
```

Autre exemple, calculer la somme des *n* premiers entiers

```
Programme Somme
début
        avec s, n, i :entier

        saisir n
        i := 1
        s := 0
        répéter
                s := s+i
                i := i+1
        jusqu'à i>n
        écrire s
fin Somme
```

Pour pouvoir comprendre un algorithme, **il faut le dérouler**, c'est-à-dire suivre l'exécution pas à pas. Supposons que dans l'exemple précédent la valeur 5 est saisie pour n:

	avant boucle	1ière Itération	2ème Itération	3ème Itération	4ème Itération	5ème Itération	après boucle
n	5	5	5	5	5	5	
i	1	2	3	4	5	6	
s	0	1	3	6	10	15	écrire 15
i>n		faux	faux	faux	faux	vrai	

3.2 Utilité des boucles en algorithmique

• Les boucles s'utilisent en algorithmique pour le contrôle des données saisies

```
lecture                              répéter
tantque non valide faire                 lecture
    lecture                          jusqu'à valide
finfaire
```

Exemple: saisir un nombre nécessairement positif

```
répéter
    saisir n
jusqu'à n>=0
```

• Les boucles s'utilisent fréquemment lorsque le programme propose plusieurs fonctionnalités à choisir dans un menu.

```
répéter
    écrire "1 - ...."
    écrire "2 - ...."
    écrire "3 - ...."
    écrire "4 - fin"
    écrire "Quel est votre choix: "
    saisir choix
    selon choix faire
       cas 1 : ....
       cas 2 : ....
       cas 3 : ....
       défaut : rien
    fin faire
jusqu'à choix=4
```

• Répétition sans connaître la fin à l'avance. Par exemple, pour calculer une moyenne sur des nombres saisis au fur et à mesure:

```
avec   somme, nbre_tot, nb_saisi :   entier,
       reponse :                     car
```

```
somme := 0
nbre_tot := 0
répéter
        écrire "Encore un nombre (o/n) ?"
        saisir reponse
        si reponse = 'o' ou reponse = 'O' alors
                saisir nb_saisi
                somme := somme+nb_saisi
                nbre_tot := nbre_tot+1
        finsi
jusqu'à rep='o' ou rep ='O'
si nbre_tot≠0 alors
        écrire "moyenne=", réel(somme)/réel(n)
sinon
        écrire "aucun chiffre saisi!"
finsi
```

• Traitement de données rangées dans des listes.

• Résolution de systèmes par méthodes numériques itératives.

etc.

4. Les compteurs

Dans de nombreux cas, le nombre d'itérations est connu “à l’avance” (avant de commencer la boucle) et on utilise une variable supplémentaire qui sert à compter (par exemple pour afficher les *n* premiers entiers). Cette variable est appelée un compteur.

Il existe une forme d'écriture compacte, lorsque la boucle effectue un comptage. Exemple : afficher les *n* premiers entiers:

```
pour i de 1 à n faire
        afficher i
fin faire
```

Ceci est équivalent à écrire:

```
i := 1                  -- initialisation
tant que i<=n faire
        afficher i
        i := i+1        -- incrémentation
fin faire
```

Le pour est caractérisé par trois éléments:

• un compteur, initialisé à une valeur de départ
• une boucle, jusqu'à atteindre (ou dépasser) une valeur finale
• une incrémentation automatique

pour <var> *de* <départ> *à* <arrivée> *faire*
    TRAITEMENT
fin faire

Si la valeur de départ est supérieure à la valeur d'arrivée, alors pour n'exécute pas le traitement. Il arrive que l'on veuille compter à l'envers ou changer le pas (par défaut 1). Il suffit d'écrire alors

pour <var> *de* <départ> *à* <arrivée> *pas* <pas> *faire*
    TRAITEMENT
fin faire

Exemples:

pour i de 1 à 10 pas -1 faire ...
pour j de 2 à 100 pas 2 faire ...

Certains langages ne permettent pas de faire varier le pas. Par contre compter à l'envers est toujours possible. En général, la variable compteur est du type entier, mais elle peut a priori également être d'un autre type:

pour col de ROUGE à BLEU faire ...
pour c de 'a' à 'z' faire ...
pour c dans <liste> faire ...

Remarque: ne jamais modifier le contenu de la variable compteur à l'intérieur de la boucle. Si une modification est nécessaire, alors il faut utiliser une boucle conditionnelle usuelle. Lorsque l'on a besoin d'interrompre le compteur avant d'atteindre la valeur d'arrivée, c'est qu'il aurait fallu utiliser une boucle conditionnelle, avec un test de fin plus élaboré.

Exemple de programme faisant la somme des *n* premiers entiers:

```
Programme Somme
début
    avec somme, nbre, i :entier

    saisir nbre
    somme := 0
    pour i de 1 à nbre faire
        somme := somme+i
    finfaire
    écrire somme
fin Somme
```

Exemple de programme permettant de tracer une ligne horizontale, sur un écran graphique, sachant que la commande allumer(x,y) permet d'afficher un pixel de coordonnées x et y dans cet écran.

```
Programme Ligne
début
```

```
        avec xmin, xmax, y :entier

        saisir xmin, xmax, y
        pour i de xmin à xmax faire
                allumer(i, y)
        finfaire
fin Ligne
```

De façon générale, les boucles pour sont utilisées lorsque le nombre exact d'itérations est connu à l'avance et lorsque celui-ci reste constant. Les boucles conditionnelles s'emploient dans tous les autres cas.

5. Imbriquer des boucles

Comme pour les *si alors sinon*, les boucles peuvent également s'imbriquer.
Exemple, afficher un rectangle plein:

```
pour i de xmin à xmax faire
        pour j de ymin à ymax faire
                allumer(i, j)
        fin faire
fin faire
```

Attention: lorsque des boucles *pour* sont imbriquées, il faut utiliser des compteurs différents (ici *i* et *j*)

Exemple de programme permettant de saisir continuellement des entiers *n* et d'afficher la somme des carrés jusqu'à ce nombre. Le programme termine lorsque *n* saisi vaut 0.

```
Programme carré
début
        avec n, i, somme : entier

        répéter
                saisir n
                si n > 0 alors
                        somme := 0
                        pour i de 1 à n faire
                                somme := somme+i*i
                        fin faire
                        afficher s
                fin si
        tant que n≠0
fin carré
```

D. Algorithmique descendante

1. Décomposition descendante

1.1 Une méthode hiérarchique

Il est souvent difficile de connaître le déroulement d'un programme dans tous ses détails à l'avance. Le problème est alors hiérarchiquement décomposé en sous-partie ou “blocs” de programmes, qui sont résolus au fur et à mesure.

Exemple : le programme “somme des carrés” décrit au chapitre précédent est décomposé en parties de programme pour faciliter sa construction:

```
Programme Carre
Début
Avec n: entier

répéter
                traite nouveau n
tant que n>0
fin Carre
```

On sait qu'il va falloir répéter un certain traitement concernant n tant que n est positif. On est pas obligé de connaître immédiatement ce traitement, donc on écrit simplement son nom.
“traite nouveau n” est une sous-partie ou "bloc" de programme qui sera résolu par après.

```
Sous-Partie traite nouveau n
début
        saisir n
        si n>0 alors
                faire la somme
        finsi
fin
```

Là encore, une fois n saisi et positif, on sait qu'on a à effectuer un certain traitement qui consiste à calculer et afficher la somme des carrés des n premiers entiers

```
Sous-Partie faire la somme
Début
Avec i, somme: entier

        somme := 0
        pour i de 1 à n faire
                somme:=somme+i*i
        fin faire
        afficher somme
fin
```

Des variables supplémentaires sont introduites pour effectuer le traitement. Ces variables sont toutes regroupées à la fin et placer dans le glossaire commun à l’ensemble du programme.

L'algorithme final devient:

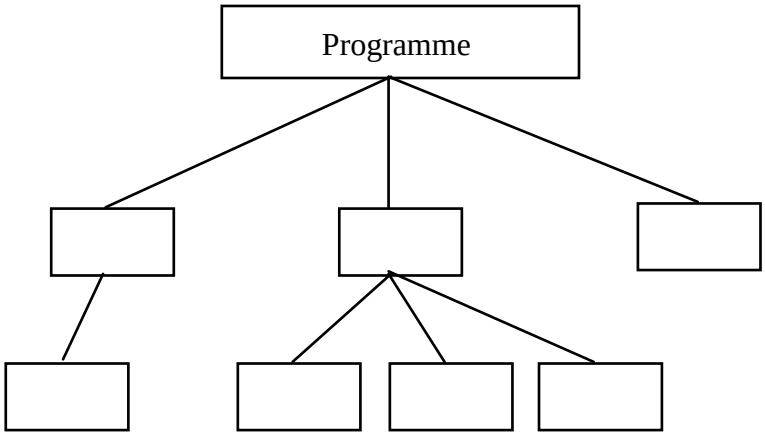
Programme carré

```
début
        avec    n :             entier
        avec    i, somme:       entier

        répéter
                saisir n
                si n > 0 alors

                        somme := 0
                        pour i de 1 à n faire
                                somme := somme+i*i
                        fin faire
                        afficher somme
                fin si
        tant que n>0
fin carré
```

Une sous-partie est directement transcrite dans l'algorithme final. C'est une *procédure* mais nous n'aborderons celles-ci que plus tard. Les sous-parties permettent simplement de développer plus facilement un algorithme, sans avoir à en connaître à l'avance tous les détails.

Noter qu'une sous-partie peut déclarer de nouvelles variables locales, mais il est préférable de toutes les rassembler dans le glossaire en tête.

La hiérarchie des sous-parties engendre un arbre:



1.2 Un autre exemple de décomposition hiérarchique

Ecrire un programme qui permet de saisir continuellement des articles, d'en saisir la quantité, le prix à l'unité et d'ajouter cela à un prix final. Le programme se termine lorsque l'utilisateur répond 'n' à la question encore un article et affiche alors la somme totale.

```
Programme Vente
Début
Avec   somme:        réel
       Reponse :     car

       Somme:= 0
       répéter
              autre article
       tant que reponse='o'
       afficher somme
fin Vente
```

```
Sous-partie  autre article
Avec   qantite :     réel
       prix:         réel

       lire prix
       saisir et contrôler quantite
       saisir et contrôler prix
       somme:=somme+prix*qantite

Fin
```

```
Sous-partie saisir et contrôler quantite
-----------------------------------------
       | |    répéter
       | |      | | lire qantitte
       | |      | | si qantite ≤0 alors
       | |      | |   | |  écrire ...
       | |      | | finsi
       | |    jusqu'à qantite >0
```

```
Sous-partie saisir et contrôler prix
-----------------------------------------
répéter
          | | écrire " encore ..."
          | | saisir reponse
jusqu'à reponse='o' ou reponse='n'
```

L'algorithme final devient donc:

```
Programme vente
début
       avec   somme:          réel,
              reponse:        car
              prix :          réel
              quantite :      entier


       somme:=0
       répéter
              lire prix
              répéter
                     lire quantite
                     si quantite ≤0 alors
                            écrire "Entrer un nombre positif!"
                     finsi
              tant que quantite ≤0
              somme := somme+prix*qte
              répéter
                     écrire "Encore un article (o/n) ?"
                     saisir reponse
              jusqu'à reponse='o' ou reponse='n'
       jusqu'à rep='o'
       afficher somme
fin vente
```

2. Preuve d'algorithme

Une fois un algorithme écrit, il faut effectuer sa preuve, c'est-à-dire démontrer son bon fonctionnement.

Pour cela il faut démontrer deux choses:

- la terminaison (pas de bouclage infini)
- le résultat obtenu est **dans tous les cas** conforme à la spécification (cahier des charges par exemple)

En pratique, une telle démonstration est difficile. On effectue alors des démonstrations par "petits" bouts qui elles-mêmes peuvent être démontrées par récurrence ou par l'absurde.

• Vérification de la terminaison

Il faut vérifier la terminaison de chaque boucle du programme. Si toutes les boucles se terminent alors nécessairement le programme termine aussi (attention à la récursivité qui sera abordée plus tard).

En pratique, la vérification se fait par exemple en considérant certaines questions:

ex: tant que a≠b faire ... fin faire

est-ce que dans la boucle a et b sont modifiés?
existe-t-il un cas dans le traitement où la condition a=b n'est jamais vérifiée?
etc...

• Vérification du résultat

La vérification du résultat se fait en déroulant le programme selon tous les cas de figure possibles. Comme ce nombre est généralement très grand, on se sert de valeurs clé et de plages de valeurs pour lesquels il est facile de démontrer que le traitement sera identique.

```
ex: tant que i≠n faire
        TRAITEMENT;
        i:= i+1
    fin faire
```

Ici la valeur clé est i inférieur ou supérieur à n+1. Dans le second cas cette boucle ne termine pas.

Comme en pratique une démonstration rigoureuse s'avère très lourde et difficile, on se limite à faire des jeux d'essai judicieusement choisis et des déroulements "à la main" de programmes. Néanmoins le manque courant (dans l'industrie par exemple) de preuve de programme, fait qu'il arrive fréquemment de découvrir des mal fonctionnements, que l'on appelle dans le jargon informatique le "bug". Actuellement la recherche tente de

développer des langages plus avancés permettant d'effectuer de façon automatique des preuves de programme par le biais de systèmes de réécriture et de moteurs d'inférence.

E. Les arrangements linéaires (indices)

1. Généralités

Dans de nombreux cas, on a besoin d'avoir recours à des variables indicées x_i par exemple, surtout lorsqu'il s'agit de gérer un nombre important de données. Par exemple pour gérer le nombre de jours qu'il y a dans un mois:

31 27 31 30 31 30 31 31 30 31 30 31

En informatique les indices n'existent pas. On se sert de tableaux, de vecteurs ou de listes. Un tableau remplace une série de variables et peut être vu d'une certaine façon comme l'équivalent des indices en mathématiques :

x_i s'écrit x[i]

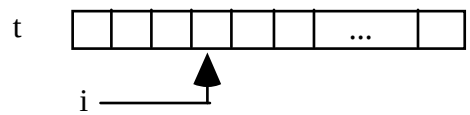
Le tableau est ce qui **se rapproche conceptuellement le plus de la mémoire physique d'un ordinateur**. Etant donné qu'un tableau correspond à un arrangement linéaire en mémoire (la mémoire elle-même peut être vue comme un tableau de "registres" de bits, chaque registre étant identifié par une "adresse"), il a toujours une taille (un nombre de case ou une dimension) finie fixée à l'avance une fois pour toute. Cette taille est constante tout le long du programme c'est une zone contiguë de la mémoire. En général, les indices varient de 1 à n (pour certains langages de 0 à n-1), n étant une constante.

Comme pour les variables ordinaires chaque élément du tableau a toujours et à tout moment une valeur. Un tableau n'est ni une liste (car une liste peut avoir un nombre variable d'éléments) ni un ensemble (car dans un ensemble on ne trouve pas deux occurrences d'une même valeur).

Le tableau a un type donné: entier, réel, car, booléen. Cela signifie que toutes les cases qui peuvent être vues chacune comme une variable individuelle, ont ce même type.

```
exemple:    avec   tabint[100] :   entier      // tableau de 100 entiers
                   tabreel[20] :   réel        // tableau de 20 réels
                   nom[50] :       car         // tableau de 50 caractères
```

Pour accéder à un élément du tableau, on indique le numéro de case (l'indice):



exemple: tabint[5] := 23 tabint[i] := 50 tabint[i+1] := x*y+2

Dans l'exemple précédent i est une variable spéciale qui représente un indice ou une "adresse". C'est aussi ce que l'on peut appeler un "pointeur".

Exemple de programme: mesurer l'évolution du poids d'une personne au cours d'une semaine et afficher la moyenne.

• *écriture sans tableau*

```
avec      pl, pm, pme, pj, pv, ps, pd, moy : réel

ecrire ("Poids lundi : ")
lire pl
ecrire ("Poids mardi : ")
lire pm
...
ecrire ("Poids dimanche : ")
lire pd
moy := (pl+pm+pme+pj+pv+ps+pd)/7
ecrire ("Poids moyen : ", moy)
```

• *ecriture avec tableau*

```
avec      pds[7], moy : réel
          i : entier

pour i de 1 à 7 faire
          ecrire ("Poids du jour ", i, " : ")
          lire pds[i]
fin faire
moy := 0
pour i de 1 à 7 faire
          moy := moy + pds[i]
fin faire
moy := moy/7
ecrire ("Poids moyen : ", moy)
```

2. La gestion de listes

Un tableau sert à recueillir et rassembler des données. Exemple: saisir dix nombres, les afficher et calculer la moyenne

```
avec      tabint[10], i:      entier
          moy :               réel;

pour i de 1 à 10 faire
          écrire "Entrer la valeur d'indice", i, " :"
          saisir t[i]
fin faire
moy := 0.0
pour i de 1 à 10 faire
          ecrire t[i]
          moy := moy + réel(t[i])
fin faire
moy := moy /10.0
écrire moy
```

Mais souvent le nombre n'est pas connu à l'avance. Exemple: idem avec un nombre n quelconque. Le problème est alors une déclaration du type :

avec tab[n], i, n: entier

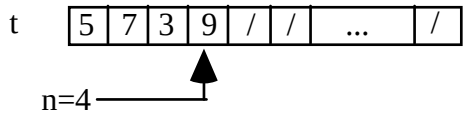
ceci n'est pas possible. En effet, n n'est pas connu. Or un tableau a toujours un nombre constant de cases. Ce nombre est fixé à l'avance et reste le même une fois pour toute.

Une solution consiste alors à déclarer un tableau assez grand à l'avance et de n'utiliser que certaines cases (les autres sont ignorées):

```
avec    t[100], i, n:    entier    // on alloue 100 cases, mais on en utilisera que n
        moy :            réel;

saisir n
contrôler n>0 et n<100
pour i de 1 à n faire
        ecrire "Entrer la valeur", i
        saisir t[i]
fin faire
moy := 0.0
pour i de 1 à n faire
        écrire t[i]
        moy := moy +réel( t[i])
fin faire
moy := moy /réel(n)
écrire moy
```

2.1 Insertion et suppression

Un tableau en plus d'un entier *n* permettent de "réaliser" une **liste** ou un **ensemble**:



Les cases au-delà de l'indice *n* sont simplement ignorées. Pour ajouter un élément *x* on peut ajouter cet élément à la position *n+1* puis incrémenter *n* de 1:

```
t[n+1] := x
n := n+1
```

Pour insérer un élément à une position *p*, telle que $1 \leq p \leq n+1$, il faut procéder à un décalage dans le tableau. Chaque case est décalée de 1 cran vers la droite à partir de *p*. Attention : il faut faire le décalage en décroissant autrement des valeurs sont perdues.

```
pour i de n à p pas -1 faire
        t[i+1] := t[i]
fin faire
t[p] := x
n := n+1
```

Pour supprimer une valeur on décale vers la gauche:

```
pour i de p à n-1 faire
        t[i] := t[i+1]
fin faire
n := n-1
```

On peut aussi prendre la dernière valeur dans le tableau et la placer à l'endroit où l'on veut éliminer la case. Ceci évite de faire un décalage.

2.2 Recherche d'un élément dans un tableau

On peut vouloir rechercher la position d'un élément dans un tableau ou le nombre d'occurrences d'un élément dans un tableau. Ceci se fait en parcourant le tableau.

Position d'un élément x:

```
i := 1
tant que i<n et t[i]≠x faire
        i := i+1
fin faire
si t[i]=x alors p:=i sinon p:= -1 finsi
```

Dans le pire des cas la valeur recherchée est en dernier.

Nombre d'occurrences d'un élément x:

```
occ:=0
pour i de 1 à n faire
        si t[i]=x alors occ := occ+1 finsi
fin faire
```

2.3 Rotations

Pour faire une rotation de 1 rang vers la droite ou la gauche:

```
x := t[n]
pour i de 1 à n-1 pas -1 faire
  t[i+1] := t[i]
finfaire
t[1] := x
```

```
x := t[1]
pour i de 1 à n-1 faire
  t[i] := t[i+1]
finfaire
t[n] := x
```

Pour effectuer une rotation de k rangs avec k inférieur à n, on peut utiliser les algorithmes précédents:

```
pour j de 1 à k faire
        x := t[n]
        pour i de 1 à n-1 pas -1 faire
                t[i+1] := t[i]
        finfaire
        t[1] := x
fin faire
```

L'algorithme suivant est "plus efficace". Mais, il ne fonctionne que si pgcd(n,k)=1

```
reste := n                  // nombre de cases restant à traiter
```

```
i := 1                          // case courante
buffer := t[1]                  // buffer
tant que reste > 0 faire
        p := (i+k)          -- nouvelle position (pour décalage à droite)
        si p>n alors p:= p-n finsi
        x := t[p]
        t[p] := buffer
        buffer := x
        i := p
        reste := reste-1
fin faire
```

3. La gestion d'ensembles

Un ensemble est une liste qui ne contient jamais deux fois le même élément. L'ordre dans lequel sont placés les éléments n'a pas d'importance.

• **Ajouter un élément** (on teste s'il existe déjà ou pas)

```
i:=1
tant que i<n et t[i]≠x faire
        i := i+1
fin faire
si t[i]=x alors
        t[n+1] := x
        n := n+1
finsi
```

• **Union de deux ensembles:**

On prend deux tableaux t1 et t2, munis de deux entiers n1 et n2. On remplit un troisième tableau avec tous les éléments de t1 et t2 (sans doublets)

```
pour i de 1 à n1 faire
        t3[i] := t1[i]
fin faire
n3 := n1
pour i de 1 à n2 faire
        j := 1
        tant que j<n3 et t2[i]≠t3[j] faire
                j := j+1
        fin faire
        si t2[i] ≠ t3[j] alors
                t3[n3+1]:=t2[i]
                n3:=n3+1
        fin si
fin faire
```

• **Intersection de deux ensembles :**

```
n3 := 0
pour i de 1 à n1 faire
        ad := faux
        pour j de 1 à n2 faire
                si t[i] = t[j] alors ad := vrai finsi
        fin faire
        si ad alors
```

```
                t3[n3+1]:=t1[i]
                n3:=n3+1
        fin si
fin faire
```

• Différence de deux ensembles :

Ajouter à t3 les éléments de t1 qui ne sont pas dans t2

```
n3 := 0
pour i de 1 à n1 faire
        ad := vrai
        pour j de 1 à n2 faire
                si t[i] = t[j] alors ad := faux finsi
        fin faire
        si ad alors
                t3[n3+1]:=t1[i]
                n3:=n3+1
        fin si
fin faire
```

4. Chaînes de caractères

Une chaîne de caractères (une ligne de texte) est un tableau de caractères. Dans certains langages le type chaîne n'existe pas en tant que type de base. On utilise alors un tableau avec une convention pour marquer la fin de la chaîne (il s'agit d'un caractère spécial, que nous noterons FIN). Dans d'autres cas le type chaîne existe comme s'il s'agissait d'un type de base. En réalité, il s'agit toujours d'un tableau, mais la gestion reste transparente au programmeur.

Déclaration: avec nom : chaîne[100] // chaîne d'au maximum 100 caractères

Pour certains langages, et en algorithmique dans notre cas, il est possible d'utiliser des opérations usuelles telle que l'affectation:

```
nom := "ceci est une chaîne"
```

D'autres opérations sont la concaténation +, la longueur et la partie de chaîne (les caractères sont obtenus directement par les indices):

```
nom := "Jean"+"-"+"Michel"          nom := ch1 + "rest"
long := longueur(nom)+1
nom := machaine[5 à 10]
```

Les opérateurs de comparaison sont également autorisés:

```
si ch1 <= ch2 alors ...          // l'ordre est lexicographique
```

Exemple : saisir un nom et l'afficher à l'envers

```
Avec nom: chaîne[50]

saisir nom
```

```
pour i de longueur(nom) à 1pas –1 faire
        ecrire nom[i]
fin faire
```

Lorsque les langages ne permettent pas d'utiliser directement l'affectation, la concaténation, la partie, la longueur et les opérateurs de comparaison, il faut les programmer soi-même:

• **l'affectation:** pour affecter à ch1 la valeur de ch2 (ch1 := ch2):

```
pour i de 1 à longueur(ch2)+1 faire
        ch1[i] := ch2[i]
fin pour                                    // un car de plus à cause de la marque de fin
```

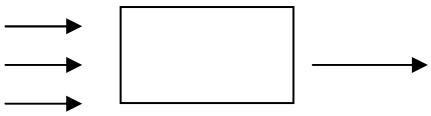
• **la concaténation:** ch1 et ch2 sont deux chaînes, le résultat est placé dans s

```
pour i de 1 à longueur(ch1) faire
        s[i] := ch1[i]
fin pour
pour i de 1 à longueur(ch2) faire
        s[i+longueur(ch1)] :=ch2[i]
fin pour
s[longueur(ch1)+longueur(ch2)+1] := FIN
```

F. Fonctions, Procédures et Modularité

1. Fonctions

Nous avons déjà rencontré préalablement des fonctions. En effet, les fonctions mathématiques telle que *sqrt()* ou *cos()* sont deux exemples. Une fonction permet de calculer une valeur à partir d'un ensemble de données fournies en entrée :



Toute fonction est caractérisée par :

- un certain nombre d'arguments : ce sont les paramètres d'entrée
- par une valeur de retour : c'est le résultat en sortie.

Les langages de programmation permettent de se définir des fonctions personnalisées, autres que les fonctions mathématiques disponibles sous forme de bibliothèque (comme cos, sin, exp, sqrt, etc.).

Exemple: fonction qui calcule factoriel n.

```
fonction factoriel(n: entier) retourne entier
début
```

```
        avec res, i: entier

        res:=1
        pour i de 2 à n faire
                res:=res*i
        finfaire
        retourne res
fin fact
```

Une fois la fonction définie, elle peut être utilisée comme n'importe quelle fonction mathématique, à condition de la déclarer auparavant. Exemple: un programme qui saisit des nombres et affiche leur factoriel.

```
programme AfficheFact
début
        avec    encore:         car
                i :             entier                  // l'entier saisi
                factoriel(entier) retourne entier       // inutile de préciser n ici

        répeter
                saisir encore
                si encore='o' alors
                        saisir i
                        contrôler i>0
                        afficher "Factoriel de ", i, " vaut :", factoriel(i)
                finsi
        jusqu'à encore/='o'
fin AfficheFact
```

Dans le cas des fonctions mathématiques prédéfinies (cos, sin, exp, etc.), il n'est pas nécessaire de déclarer la fonction avant de l'utiliser, car cette déclaration est déjà faite au niveau des inclusions. C'est précisément l'objet de ces inclusions.

Notez que la variable i dans la fonction n'a rien à voir avec celle du programme AfficheFact (cf. plus loin).

Une fonction peut avoir plusieurs paramètres de types différents. Exemple : rechercher l'indice du maximum dans un tableau d'entiers entre deux bornes a et b:

```
fonction RechIndiceMaxTab(t[]: entier, a, b: entier) retourne entier
début
        i:      entier
        imax:   entier

imax:=a
pour i de a+1 à b faire
        si t[i]>t[imax] alors
                imax:=i
        finsi
finfaire
retourne imax
```

fin RechIndiceMaxTab

1.1 Passage de paramètres

Il est important de faire la distinction entre :
- La définition de la fonction (elle sera placée dans un fichier à part)
- La déclaration de la fonction avant son utilisation : on donne un **profil** (ou **signature**)
- L'appel de la fonction (l'utilisation proprement dite)

Au niveau de la définition, le nom de chacun des paramètres est purement symbolique, c'est pourquoi au niveau de la déclaration il est omis. Seul le type des paramètres et leur ordre importe pour l'utilisation.

Les variables déclarées à l'intérieur d'une fonction n'existent que pour cette fonction. Là encore il ne s'agit que d'un symbole. Le i du programme principal n'a rien à voir avec celui de la fonction factorielle. On dit que la variable est **locale** à la fonction. Toutes les variables déclarées dans une fonction sont locales. Une fois la fonction terminée, toutes ses variables locales sont détruites. On dit que les variables ont **une portée** limitée à la fonction.. Inversement toute variable utilisée par la fonction doit obligatoirement soit faire partie des paramètres, soit être déclarée localement par la fonction.
Si on a besoin d'une variable du programme faisant appel à la fonction, il faut ajouter un paramètre, et faire passer cette variable au travers de ce paramètre.

Au niveau de l'utilisation, il est possible de faire appel aux fonctions avec des expressions:

```
x := factoriel(x+y)
x := factoriel((a+b)/2)
```

Au niveau de la définition, modifier la valeur de l'un des paramètres de la fonction, ne modifie pas la valeur dans le programme qui fait l'appel:

```
fonction incrementer(x: entier) retourne entier
début
        x := x+1
        retourne x
fin incrementer
```

Lorsque la fonction est utilisée dans un programme :

```
avec    a, v :          entier
        incrementer(entier) retourne entier

v := 10
a := incrementer(v)             // v n'est pas modifié !
afficher a,v                    // affichera 11 et 10
```

On dit qu'il y a passage **des paramètres par valeur**. En fait, au moment où l'on fait appel à la fonction, ce n'est pas la variable qui est passée en paramètre mais uniquement sa valeur. C'est ce qui fait que l'on peut utiliser des expressions arithmétiques :

```
avec    a, v :          entier
        incrementer(entier) retourne entier

v := 10
a := incrementer(v+5)          // v+5 n'est pas modifié !
afficher a,v                          // affichera 16 et 10
```

1.2 le retour

Normalement le retour peut être de tout type comme pour les paramètres. Mais beaucoup de langages interdisent des retours de type tableau ou structure. Ceci est lié au fait que le retour doit être recopié dans la variable réceptrice. Dans le cas de structures ou tableaux cela peut engendrer des traitements extrêmement coûteux en temps de calcul et en espace mémoire.

2. Décomposition hiérarchique et procédures

Dans la décomposition hiérarchique développée dans les premiers chapitres, nous avons décomposé un programme en plusieurs parties ou blocs de programme. On constate qu'il arrive fréquemment d'avoir besoin plusieurs fois de la même partie (ou bloc), soit dans le même programme, soit même dans des programmes différents. On peut se servir de fonctions, et faire appel à ces fonctions : par exemple pour saisir un entier nécessairement positif.

Dans de nombreux cas cependant, il n'y a pas de retour, par exemple pour afficher un tableau. Pour éviter de dupliquer inutilement du code avec des parties de programme identiques, on va construire des "fonctions" qui n'ont pas de retour. Ce type de fonctions plus général est appelé *procédure*.

Exemple :

```
procedure AffTab(t[]: entier, n: entier)
début
        avec    i:              entier

        pour i de 1 à n faire
                afficher t[i]
        finfaire
fin AffTab
```

Ecrivons à l'aide de procédures un programme permettant de saisir, de trier puis d'afficher un tableau:

```
programme tri
```

```
début
        avec    t[100]:         entier
                n:              entier

        SaisirTab( ; t, n)
        TrierTab(t, n ; t)
        AffTab(t, n ;)
fin tri
```

Cette écriture est quasi identique à la décomposition descendante que nous avons déjà étudiée. Cependant, ici, les parties de programme n'ont pas besoin d'être réinsérées dans le programme. Egalement, contrairement aux parties, des paramètres sont définis.

En fait, une procédure est une partie de programme détachée du corps de ce programme. Elle attend en entrée des paramètres pour pouvoir faire l'exécution, comme si elle était intégrée dans le programme au même titre qu'une partie. On remarque que les paramètres peuvent être en entrée seulement ou bien en retour. Nous séparons les deux types de paramètres par un point-virgule. Les procédures sont définies dans des fichiers à part, et peuvent être utilisées par d'autres programmes:

```
procedure SaisirTab(; t[]: entier, n: entier)
début
        avec    i: entier
                v: entier

        SaisirEntierPos(; v)
        pour i de 1 à v faire
                saisir t[i]
        finfaire
        n := v
fin SaisirTab
```

Une procédure peut faire appel à une autre procédure, comme par exemple SaisirEntierPos, qui est une procédure permettant de saisir un nombre entier positif.

```
Procedure TrierTab(t[]: entier, n: entier ; t[]: entier)
début
        avec    i, imin: entier

        pour i de 1 à n-1 faire
                imin := RechIndiceMinTab(t, i, n)
                ExgTab(t, i, imin ; t)
        finfaire
fin TrierTab

procedure SaisirEntierPos(; x : entier)
début
        avec v: entier

        saisir v
        tantque v <=0 faire
```

```
            saisir v
        finfaire
        x := v
fin SaisirEntierPos

procedure ExgTab(tab[]: entier, a:entier, b: entier ; tab[]: entier)
début
        avec x: entier

        si a≠b alors
                x := tab[a]
                tab[a] := tab[b]
                tab[b] := x
        finsi
fin ExgTab
```

La procédure d'affichage et la fonction RechIndiceMinTab ont déjà été vues.

2.1 Remarques

• comme pour les fonctions, les variables sont locales. Toute variable utilisée dans une procédure doit être déclarée dans celle-ci ou doit être passée en paramètre.

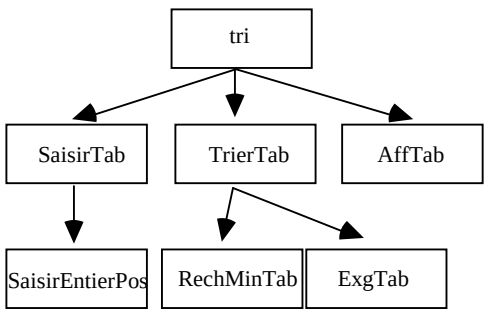
• Certains paramètres sont en entrée seulement (à gauche du ;), d'autres en sortie seulement (à droite) et certains en entrée et sortie.

Lorsqu'un paramètre est en sortie (ou en entrée et sortie):

```
toto(a: entier,...    ;  a: entier, ...)
```

On dit qu'il y a passage par **référence** contrairement au passage par **valeur**. Pour ce type de paramètre il n'est pas possible d'appeler la procédure avec des expressions par exemple: toto(5+x, ...). Avec un passage par référence la modification de la variable dans la procédure **se répercute au niveau supérieur**. De façon générale, lorsqu'il y a modification d'une variable par une procédure, on dit qu'il y a **effet de bord**. Les effets de bords peuvent être (dans certains cas) non désirés. En principe, ils sont à éviter. Soit les arguments sont en entrée, soit en sortie, rarement les deux, sauf pour les tableaux et structures.

2.2 Hiérarchie



Par rapport aux parties de programme, les procédures n'apportent rien de neuf, si ce n'est une écriture plus compréhensible et acceptée par les langages de programmation. Dorénavant, on adoptera une méthode de décomposition de problème descendante à l'aide de procédures et fonctions uniquement. On cherchera à éviter les parties de programme, pour éviter d'alourdir le code. Il est important de faire des procédures qui soient les plus générales possible, afin d'augmenter la "**réutilisabilité**".

3. Modularité

Un module est un ensemble de fonctions et procédures regroupées dans un même fichier, et ayant un rapport avec un même *sujet* ou *type* de données. Un module *liste d'entiers* peut comprendre les éléments suivants :

• La déclaration du type liste d'entier (c'est un couple tableau + entier)
• Le profil des opérations suivantes :
    • Saisir une liste de taille n
    • Afficher le tableau
    • Trier le tableau
    • Echanger deux cases
    • Ajouter une valeur à la fin
    • Ajouter une valeur en position donnée
    • Supprimer une valeur en position donnée
    • Rotation à gauche/ droite
    • etc.

En programmation, **un module** est caractérisé par deux composantes :

• **une spécification**: celle-ci permet de lister (énumérer) toutes les fonctions et procédures avec leur **profil**. Elle définit également les différentes structures de données. C'est la partie accessible ou vue par l'utilisateur du module (le **client**).

• **une implémentation**: c'est là que sont définies et programmées les procédures et fonctions. Cette partie est invisible au client du module.

En général, les langages de programmation proposent un certain nombre de modules, regroupés sous forme de **librairies** (ensembles de modules). En C/C++, Java et Python, il y en a de très nombreuses: pour les entrées / sorties, pour les chaînes de caractères, pour les opérations mathématiques, pour les affichages graphiques et l'interfaçage, pour la gestion du système d'exploitation (processus, mémoire, inter-process communication, etc.), pour la gestion du réseau, pour la gestion de dates et d'heures, etc.. On peut avoir accès en plus à des modules spécifiques, par exemple pour la gestion de bases de données relationnelles, pour les affichages 3D, etc.

3.1 Spécification d'un module

```
module ListeEnt = {

    type ListeEnt= {
        tabent[100] :       entier
        n_elem :            entier
```

```
        }

    procedure SaisirListe( ; li : ListeEnt)
    fonction TailleList( li : liste) retourne entier
    procedure AffListe( li : ListeEnt ; )
    procedure TrierListe( li :ListeEnt ; li : ListeEnt)
    procedure AjoutElem(li :ListeEnt, x: entier ; li : ListeEnt)
    etc...

        } // fin du module
```

La spécification indique le profil des opérations. En plus du profil, il faut toujours indiquer sous forme de commentaire **les pré- et post-conditions**.

- Les pré-conditions :

Elle rassemble les informations concernant les paramètres en entrée. On indique leur **signification** et surtout leur **domaine de définition**. On indique également ce qui se passe si le domaine de définition n'est pas respecté.

- Les post-conditions :

C'est la description de ce que fait la fonction ou procédure. On indique les valeurs de retour, les effets de bord et les affichages.

Exemple : une procédure permettant de supprimer une valeur dans une liste. Voici la spécification, accessible à l'utilisateur du module :

```
// Suppression d'une valeur à une position donnée dans une liste
// li : la liste dont on veut supprimer une valeur
// pos : la position de l'élément à supprimer. pos doit être compris entre 1 et
// TailleListe(li)
// Cette procédure modifie la liste li en décalant tous les éléments venant après pos de 1
// rang vers la gauche pour écraser la valeur en pos.
// La taille est réduite de 1. Si pos est en dehors du domaine de  définition rien n'est
effectué

procedure Supprimer(li : liste, pos: entier ; li : liste)
```

3.2 Implémentation

On détaille simplement pour chaque procédure et fonction, l'algorithme associé (tout est placé dans un même fichier):

```
procedure Supprimer(li : liste, pos: entier ; li : liste)
début
        avec i: entier

        si n_elem>0 et pos<=n_elem alors
                pour i de pos à n_elem-1 faire
                        tabent[i] := tabent[i+1]
```

```
            finfaire
        finsi
fin Supprimer
```

G. Structures

1. Le type structure ou enregistrement

Il arrive très fréquemment de travailler en informatique avec des données “complexes” ou “composées” telles que les images, les sons, les clients, les factures, les adresses, etc.

Toutes ces informations peuvent se composer à partir des types de base. Par exemple, un client est identifié par un nom, prénom, une date de naissance, une adresse et un numéro de téléphone. Le nom correspond à une chaîne de caractères, ainsi que le prénom et le numéro de téléphone. La date est composée de 3 entiers (le jour, le mois et l'année). L'adresse correspond à un numéro de rue (entier), le nom de la rue (chaîne), d'un code postal (entier ou chaîne) et d'une commune (chaîne).

Pour pouvoir faciliter la gestion de ces données complexes et, afin d’éviter une quantité trop importante de variables, il est possible de regrouper les informations dans un nouveau type. Ce type consiste en un assemblage de types de base. Par exemple, pour la gestion de clients, il est possible de se définir un nouveau type appelé *client*. Ce type client est un assemblage de : nom (chaîne), prénom (chaîne), etc.

Voici comment on déclare le nouveau type:

```
avec    type client =   {
                        nom :               chaîne[20],
                        prenom :            chaîne[25],
                        jour, mois,annee:   entier,             // date naissance
                        numrue :            entier              // adresse
                        rue :               chaîne[50],
                        code_postal :       chaîne[5],
                        commune :           chaîne[20]
                        }

        leclient :      client              // la variable leclient est du type client
```

L’assemblage de données porte le nom de **structure** ou **enregistrement**. Pour pouvoir travailler avec la variable de type structure leclient, il faut pouvoir accéder aux données qui la composent. Ces données sont appelées les **membres**. Pour accéder aux membres, on se sert d'un opérateur "." :

```
        saisir(leclient.nom)
        saisir(leclient.jour)
```

Pour accéder au deuxième caractère du nom du client leclient, on écrit :

```
leclient.nom[2] := 'a'
```

Notez qu'une structure peut contenir d'autres structures ainsi que des tableaux (en effet, la chaîne est un tableau) :

```
avec    type date =     {
                        jour, mois, annee :     entier
                        }
        type adresse =  {
                        nrue :                  entier,
                        rue :                   chaîne[50],
                        code :                  chaîne[50],
                        commune :               chaîne[25]
                        }
        type client =   {
                        nom :                   chaîne[20],
                        prenom :                chaîne[25],
                        naiss :                 date,
                        ad :                    adresse
                        }
```

Pour accéder aux membres d'une structure incluse dans une autre structure, on utilise une suite de "." :

```
afficher leclient.naiss.jour
```

Notons qu'il n'est pas possible de saisir ou d'afficher directement un type structure. Il faut le faire membre par membre. En effet, une structure n'est qu'une forme un peu plus pratique d'écriture permettant de rassembler un groupe de données. Par contre, il est possible de faire une affectation complète de deux variables de même structure :

```
avec    leclient1, leclient2 :          client
…
leclient2 := leclient1          // affecte membre à membre
```

2. Structure contenant un tableau : liste et ensemble

Nous avons vu qu'une liste pouvait être "créée" facilement grâce à un couple tableau et entier, le tableau contenant les éléments de la liste et l'entier indiquant le nombres de cases occupées dans le tableau. On peut donc rassembler ces deux données dans une structure pour créer un type *liste* (de même pour un type *ensemble*):

Exemple. Programme qui saisit et ajoute séquentiellement des entiers à une liste.

```
Programme liste
début
        avec type liste = {             // création du type liste
                tab[100] :      entier,
                nombre :        entier
                }
```

```
        li :    liste               // li est un couple tableau / entier
        rep :   caractère           // continuer la saisie

li.nombre := 0                      // initialement il n'y a aucune donnée dans la liste
répéter
        écrire "Encore une valeur (o/n)?"
        saisir rep
        si rep = 'o' alors
                lire li.tab[li.nombre+1]
                li.nombre := li.nombre +1
        finsi
tantque rep='o'
fin liste
```

L'utilisation d'une structure rend le programme plus compréhensible. Evidemment une structure peut contenir plusieurs tableaux.

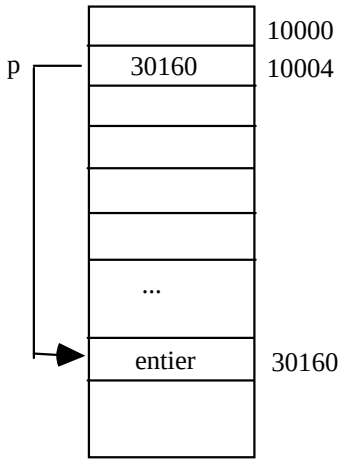
H. Pointeurs et référencement mémoire

1. Définition

Au cours des sections précédentes, nous avons étudié l'utilité des variables. Chaque variable est caractérisée par:

- un type (entier, réel, caractère, ...);
- une valeur;
- un nom;
- une zone mémoire.

Une variable est un tiroir dans lequel, il est possible de ranger une certaine valeur. Un **pointeur** ou **référence** est une variable un peu particulière car **son contenu est une adresse mémoire**:



Un pointeur est typé: c'est à dire qu'il pointe vers une zone contenant un type bien précis (entier, réel, booléen, etc.).

Attention: pas tous les langages ne permettent de définir directement des pointeurs. Dans le cas où les pointeurs n'existent pas directement en tant que type, on utilise simplement une référence: c'est un indice dans un tableau, le tableau étant une zone contiguë de la mémoire de l'ordinateur.

2. Opérations sur les pointeurs

Voici un exemple d'utilisation de pointeurs:

```
procedure pointeurs
début
        avec    a, b: entier
                p1, p2, p3 : pointeur entier

        a := 2
        b := 5
        p1 := adresse a
        p2 := nouveau entier
        contenu p1 := b
        contenu p2 := 3
        contenu p3 := contenu p2
        p3 := p1
        contenu p3 := 10

fin pointeurs
```

Le schéma correspondant à cet algorithme, nous permet de comprendre l'évolution des différentes variables ainsi que leur contenu.

2.1 L'opérateur adresse

Cet opérateur permet de récupérer l'adresse mémoire d'une variable, et donc de faire pointer un pointeur vers une variable donnée.

Le pointeur peut alors faire changer le contenu de cette variable.

L'opérateur d'adresse reste cependant très peu utilisé. En fait, uniquement dans des cas particulier, comme en C, pour simuler un passage de paramètres par référence (on effectue alors un passage par adresse).

2.2 L'opérateur nouveau

Au moment de l'initialisation le pointeur ne pointe vers "rien", ou dans le pire des cas vers une zone inconnue. Avant de pouvoir utiliser tout pointeur il faut l'initialiser en allouant une certaine zone de mémoire.

C'est l'opérateur nouveau qui permet d'allouer une zone:

p:= nouveau entier

Nouveau est une opération système complexe qui retourne une certaine adresse.

2.3 L'opérateur contenu

Cet opérateur permet de modifier le contenu de la zone mémoire pointée.

contenu p := 5

Si p est un pointeur et pointe vers une case contenant un entier, le contenu de cette case prend la valeur 5.

Cette opération est l'opération la plus fréquemment utilisée. Elle est primordiale, mais il faut toujours s'assurer que le pointeur a bien été initialisé.

3. Pointeur sur des structures

Un pointeur peut être d'un type de base, mais peut également être du type structure, c'est-à-dire pointer vers une zone mémoire correspondant à une structure de données.

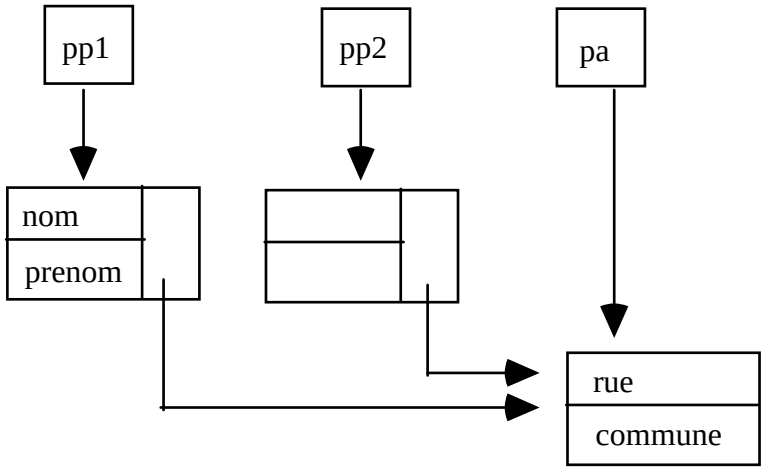
Exemple:

```
avec type pers = {
        nom:            chaine[30],
        prenom:         chaine[30] }

        pp: pointeur pers

pp := nouveau pers
(contenu pp).nom := "toto"
(contenu pp).prenom := "tata"
afficher (contenu pp).nom
```

Une structure de données peut elle-même contenir des pointeurs:

```
type adresse = {
                rue:            chaine[50],
                commune:        chaine[40] }

type pers = {
                nom:            chaine[30],
                prenom:         chaine[30],
                adr:            pointeur adresse  }

pa : pointeur adresse
pp1, pp2: pointeur pers

pp1 := nouveau pers
```

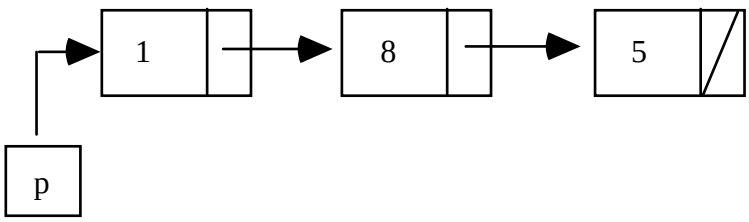
```
pp2 := nouveau pers
(contenu pp1).nom := "toto"
(contenu pp1).prenom := "titi"
(contenu pp2).nom := "tata"
(contenu pp2).prenom := "tutu"
pa := nouveau adresse
(contenu pp1).adr := pa
(contenu ((contenu pp1).adr)).rue := "rue truc muche"
(contenu ((contenu pp1).adr)).commune := "machin"
(contenu pp2).adr := pa
```



4. Chaînage en mémoire

Les tableaux ont un inconvénient majeur: il faut allouer un espace suffisamment grand dès le départ, ce qui fait que souvent la taille allouée est trop grande et donc en grande partie non utilisée.

Avec l'aide de pointeurs, il est possible d'effectuer des chainages de cellules en mémoire, chacune contenant une certaine information (comme une case du tableau). Cela permet de n'occuper que l'espace mémoire réellement utilisé. Exemple: une chaîne d'entiers



La structure de données associée est celle d'un maillon:

```
type maillon = {
```

```
        v:          entier
        suivant:    pointeur maillon  }
```

Voici le programme permettant de créer la chaine de l'exemple:

```
avec    p:      pointeur maillon
        s:      pointeur maillon

p:= nouveau maillon
s:= p
(contenu s).v := 1
(contenu s).suivant := nouveau maillon
s:= (contenu s).suivant
(contenu s).v := 8
(contenu s).suivant := nouveau maillon
s := (contenu s).suivant
(contenu s).v := 5
(contenu s).suivant := null
```

Au cours de traitements de chaines, on est fréquemment amené à utiliser les opérations suivantes:

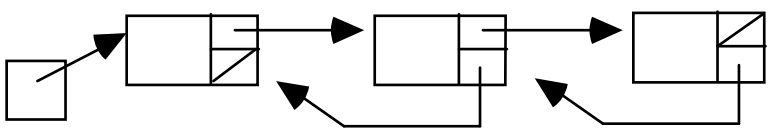
- adj/sup de la tête de chaîne
- adj/sup de la queue
- adj/sup à un certain rang dans la chaîne
- longueur de la chaîne
- élément en position
- concaténation de deux chaînes

etc.

Les chaînes peuvent servir à définir des types particuliers de listes, que l'on appelle listes chaînées.

5. Chaînage double

Dans le cas précédent, un seul pointeur était utilisé dans la structures de données. Il est possible de définir plusieurs pointeurs. Par exemple un pointeur vers le suivant et un vers le précédent. Dans ce cas on parlera de chaînage double:



La structure associée est la suivante:

```
type maillon = {
        v:                      entier
        suivant, precedent:     pointeur maillon }
```

De la même façon, il est possible de définir des opérations sur ces chaines, comme par exemple l'adjonction en tête, la suppression en tête, etc...

```
procedure adjtete(p: pointeur maillon, v: entier ; p: pointeur maillon)
début
        avec np: pointeur maillon

np := nouveau maillon
(contenu np).v := v
(contenu np).suivant := p
(contenu np).precedent := null
si p≠null alors
        (contenu p).precedent := np
finsi
p := np
fin adjtete
```

Exemple de suppression de la tête:

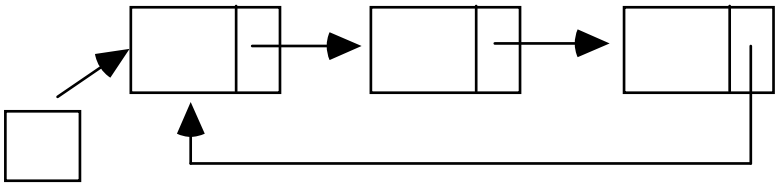
```
procedure suptete(p: pointeur maillon ; p: pointeur maillon)
début
        avec buff: pointeur maillon

si p≠null alors
        si (contenu p).suivant ≠ null alors
                buff := (contenu p).suivant
                (contenu buff).precedent := null
                detruire p
                p := buff
        sinon
                detruire p
                p := null
        finsi
finsi
fin suptete
```

Toutes les autres opérations peuvent également être modifiées, et adaptées au chaînage double.

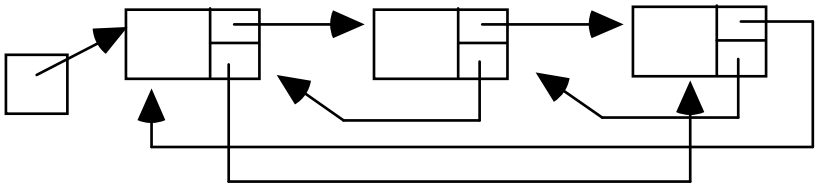
6. Autres types de chaînages

Il est possible d'imaginer encore beaucoup d'autres types de chaînages. Les plus courrant sont:

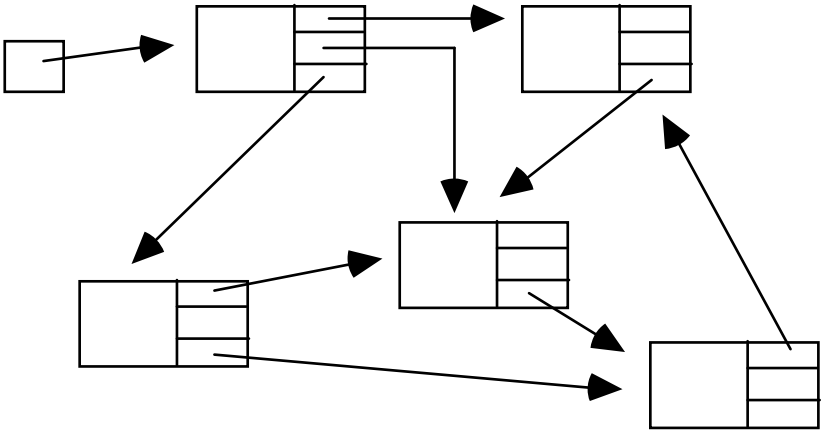
• le chaînage circulaire: la queue pointe vers la tête



• le chaînage circulaire double



• le chaînage quelconque



Ce dernier type de configuration forme un graphe.

I. La notion d'objets et de types abstraits de données

1. Pourquoi des objets?

En programmation avancée, on se rend compte que l'on est amené à travailler avec des environnements et univers de plus en plus complexes. La programmation modulaire descendante aide à subdiviser le problème, afin d'éviter, par exemple, de dupliquer du code et surtout afin de clarifier l'écriture.

En analysant les *modules* que nous avons introduit précédemment, on voit qu'ils ont souvent un rapport avec des *types* particuliers:

module ecran	--->	type ecran
module tableau d'entiers	--->	type liste
module fichier	--->	type fichier

Au delà d'un type, il s'agit souvent d'un **objet**, auquel sont affectées des opérations propres, permettant de manipuler (modifier) les variables de ce type.

2. Qu'est ce qu'un objet?

Tout objet est composé de deux éléments:

- un ensemble d'opérations qui permettent de gérer et de manipuler l'objet
- une structure de donnée, qui sert de tampon aux informations.

L'utilisateur n'a pas accès aux composantes de la structure. On parle d'encapsulation. Il ne voit l'objet qu'à travers les opérations. Exemple, objet fenêtre:

avec f: fenetre

```
f := OuvrirFenetre(320, 200, "ma fenetre")
DessineRectangle(f, 10, 10, 20, 20)
DessineTriangle(f, 10, 10, 100, 100, 200, 200)
FermerFenetre(f)
```

=> on ne "voit" l'objet fenêtre qu'à travers les opérations: OuvrirFenetre, Dessine et FermerFenetre, ainsi qu'à travers leur "effet". En fait, peu importe la façon dont les opérations sont concrètement réalisées. L'essentiel est de pouvoir créer des fenêtres et faire des affichages.

En réalité, le fichier est également un objet. Lorsque l'on ouvre un fichier et qu'on l'utilise, on se sert d'une variable de type "fichier". Mais le type fichier ?, c'est quelle structure exactement ?..., en fait... peu importe!

De même le type réel est un objet. Peu importe la façon dont sont codés les réels (norme ANSI sur n bits etc.), l'essentiel c'est de pouvoir utiliser les opérations +, -, * etc. et de connaître leurs domaines de validité.

En fait, tous les types de base sont également des objets: ils ont un codage binaire pour représenter l'information et un certain nombre d'opérations qui permettent de manipuler des variables de ce type.

Les avantages d'une vision objet sont: principalement la clarté des programmes. Ils deviennent faciles à lire, compréhensibles, intuitifs et faciles à maintenir.

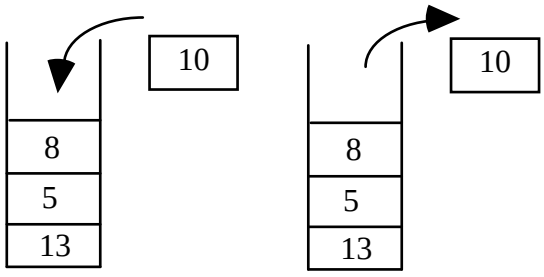
Chaque objet est une unité indépendante du reste du contexte (une boite noire) et n'est utilisé que comme "un service". Les programmes sont indépendants de l'implantation choisie, en raison de **l'encapsulation**. L'utilisateur de l'objet ne se préoccupe pas de son fonctionnement interne mais uniquement du résultat des opérations. Maintenir un logiciel c'est maintenir chaque objet individuellement. Faire la preuve de programme, c'est faire la preuve de chaque objet. Etendre un logiciel, c'est étendre les objets (les agrandir ou les remplacer) ou c'est ajouter des nouveaux objets, tout en préservant les parties "inchangées" du programme.

On dit qu'un langage est orientée objet, lorsqu'il est possible d'étendre les objets de base, pour en créer des nouveaux, qui feront alors partie intégrante du langage (comme s'ils existaient sous forme de types de base). Leur utilisation est transparente. Parmi les propriétés on retrouve souvent le **polymorphisme**, la **généricité** et **l'héritage**.

3. Spécification d'objets

En général, les objets sont spécifiés: c'est-à-dire que l'on indique l'ensemble des opérations, aussi appelées **méthodes**, et leur comportement (c'est-à-dire ce que font ces opérations).

Prenons l'exemple de l'objet **pile d'entiers**:



Une pile est, comme son nom l'indique, un empilement. Les opérations usuelles sont empiler, c'est-à-dire ajouter une valeur de plus sur le tas, depiler, c'est-à-dire retirer la valeur au sommet, rechercher le sommet et la hauteur de la pile.

Voici la spécification de l'objet pile:

SORTE pile
UTILISE entier, booleen
OPERATIONS

pilenouv:		--->	pile
empiler:	pile * entier	--->	pile
depiler:	pile	--->	pile * entier
sommet:	pile	--->	entier
hauteur:	pile	--->	entier

Dans la spécification on rajoute habituellement les preconditions des opérations:

PECONDITIONS

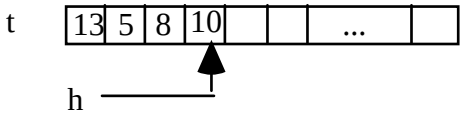
depiler(p) <=> hauteur(p)>0
sommet(p) <=> hauteur(p)>0

En informatique théorique il est d'usage de se donner une axiomatique sur l'ensemble des opérations. On est alors amené à travailler sur des prédicats du premier ordre typés.

Ici, nous nous contentons d'écrire les algorithmes, après avoir choisi une structure de donnée.

L'objet pile d'entier, tel qu'il est décrit précédemment est un type abstrait (TAD), il faut le rendre concret en choisissant une implantation.

Exemple: Utilisation d'un tableau avec un entier pour marquer le sommet.




```
type pile = {
        t[100] :        entier          -- les valeurs empilées
        h:              entier          -- la hauteur de la pile
        }

fonction pile::pile()  // le constructeur de base porte le nom de l'objet
début
        h := 0
fin pile

procedure pile::empiler(x: entier)
début
si h < 100 alors
        t[h+1] := x
        h := h+1
finsi
fin empiler

fonction pile::depiler() retourne entier
debut
si h > 0 alors
        h := h -1
        retourne t[h+1]
sinon
        -- générer une erreur
finsi
fin depiler

fonction pile::sommet() retourne entier
début
si h > 0 alors
```

```
        retourne t[h]
sinon
        -- générer une erreur
finsi
fin sommet

fonction pile::hauteur() retourne entier
début
        retourne h
fin hauteur
```

On peut rajouter à la spécification la précondition suivante:

PECONDITIONS

empiler (p,x) <=> hauteur(p)<100

Une fois l'objet spécifié et implanté, il peut être utilisé dans un programme, comme un service:

```
programme utilise_pile
début
        avec    p: pile
                v : entier

p := pile ()
p.empiler(10)
ecrire p.sommet()
ecrire p.hauteur()
p.empiler(3)
ecrire p.sommet()
v := p.depiler()
v := p.depiler()
ecrire p.hauteur()

fin utilise_pile
```

L'utilisation de la pile reste transparente. Il n'est pas nécessaire de se préoccuper de l'implantation choisie. On se fiche de savoir que derrière la structure pile se cache un tableau. D'ailleurs, il est possible à tout moment de choisir une implantation différente de l'objet pile.

Notons que les opérations font partie intégrante de l'objet (de la structure): l'appel de l'opération se fait donc en utilisant l'opérateur d'accès à un membre, cad le '.'

4. Constructeurs, destructeurs et observateurs

On retrouve en général pour tout objet trois classes d'opérations.

• Les constructeurs et destructeurs

Tout objet a toujours nécessairement un constructeur de base, permettant de créer l'objet ou de l'initialiser

```
p := pilenouv()
f := ouvrire("toto", lecture)
```

On parle aussi de constructeurs, pour les opérations permettant de faire "grandir" l'objet:

```
p.empiler( x)
```

En effet, l'opération d'empilement fait augmenter et évoluer l'objet dans un sens "positif".

Le destructeur de base correspond à l'inverse du constructeur de base. En générale, cela permet de libérer la mémoire éventuellement allouée par le constructeur de base

```
fermer(f)
```

On parle aussi de destructeur pour les opérations permettant de faire diminuer l'objet:

```
p.depiler()
```

• Les modificateurs

Ce sont les opérations qui permettent "de modifier" l'objet sans être forcément un constructeur ou un destructeur. Par exemple, mettre à jour la valeur d'un élément dans une liste.

• Les observateurs

Ce sont les opérations qui permettent "de voir" l'objet à travers d'autres objets, par exemple de base.

```
hauteur()       --> entier
sommet()        --> entier
estvide()       --> booleen
```

On renvoie toujours un objet de type autre que le type considéré.

5. Conception objet

Une comparaison entre une approche procédurale classique modulaire et objet permet d'illustrer l'intérêt de l'approche objet.

5.1 Approche procédurale classique

En programmation classique (procédurale), un programme se compose de deux éléments fondamentaux : **l'algorithmique** (ils s'agit des traitements) et les **structures de données** (il s'agit de l'information à traiter).

Le problème d'une telle approche réside principalement dans l'utilisation des données. En cas d'évolution du système et en particulier des données sous-jacentes, il faut rechercher, afin de les mettre à jour, **tous** les traitements ayant trait aux données concernées. L'exemple 1 qui suit illustre le problème.

Exemple 1. Calculer l'aire d'un triangle défini par trois points de l'espace 2D

Version 1 : utilisation de variables explicites pour les coordonnées (structure de donnée inexistante)

```
Programme aire
Avec Ax, Ay, Bx, By, Cx, Cy : réels
Début
     Saisir Ax, Ay, Bx, By, Cx, Cy
     Afficher "l'aire vaut : ", 0.5*((Bx-Ax)*(Cy-Ay)-(Cx-
Ax)*(By-Ay))
fin
```

Version 2 : utilisation de tableaux pour les coordonnées

```
Programme aire
Avec A[2], B[2], C[2] : réels
Début
     Saisir A[1], A[2], B[1], B[2], C[1], C[2]
     Afficher "l'aire vaut : ", 0.5*((B[1]-A[1])*(C[2]-A[2])-
(C[1]-A[1])*(B[2]-A[2]))
fin
```

Version 3 : utilisation d'une structure de type couple pour les coordonnées

```
Programme aire
Avec type couple = { x, y : réel }
     A, B, C : couple
Début
     Saisir A.x, A.y, B.x, B .y, C.x, C.y
     Afficher "l'aire vaut : ", 0.5*((B.x-A.x)*(C.y-A.y)-(C.x-
A.x)*(B.y-A.y))
fin
```

Le choix de la structure de données influe naturellement sur les traitements. Une modification de ce dernier induit logiquement une modification de tous les traitements lui ayant trait. Dans le cadre de grands systèmes informatiques, le nombre de traitements pour un choix de structure de données précis peut très vite devenir exubérant. Si bien que les critères de maintenance et de portabilité deviennent également très vite extrêmement déficients.

Par ailleurs, ce type d'approche engendre souvent des liens très forts entre les différents sous-systèmes le constituant. La modification d'un sous-système (ensemble de procédures et fonctions) engendre par-là, la nécessité de modifier un autre sous-système fortement connecté à ce dernier et ainsi de suite. L'effet boule de neige conduit à des systèmes qui dans leur ensemble sont difficiles à faire évoluer facilement et de manière très localisée.

Les deux problèmes précédents ont donc tout naturellement conduit les informaticiens à investiguer des alternatives plus efficaces et répondant mieux aux besoins évolutifs d'un système. Il s'agit de l'approche objet.

5.2 Approche objet

Avec une approche de type objet, un programme est essentiellement composé d'un ensemble d'objets communiquant entre eux. Les objets sont eux-mêmes composés de deux éléments : les **données** (partie statique de l'information) et les **méthodes** (partie dynamique de l'information). Contrairement à une approche procédurale, l'accent n'est plus mis sur le choix de la modélisation des données mais *sur leur comportement*. Un objet représente donc une forme d'abstraction de l'information à traiter.
L'exemple 1 du problème précédent (calcul de l'aire d'un triangle) s'écrit au travers d'une vision objet de la manière suivante :

Exemple 1 – revu en version objet avec deux classes : point2D et vecteur2D

```
Programme aire
Avec A, B,C :   point2D
     V1,V2 :          vecteur2D
Début
     Saisir A,B,C
     V1 := vecteur2D(A,B)
     V2 := vecteur2D(A,C)
     Afficher "l'aire vaut : ", 0.5*V1.Determinant2D(V2)
fin
```

L'exemple montre clairement que l'accent n'est plus mis sur la structure de données (ici, elle n'est pas même connue, d'ailleurs, elle n'a aucun lieu de l'être !), mais sur les traitements qu'il est possible d'effectuer, en l'occurrence : 1. saisir un point, 2. initialiser un vecteur à partir de deux points, 3. récupérer le déterminant entre deux vecteurs.

Les deux objets utilisés précédemment co-opèrent entre eux pour résoudre un problème général. Il eût évidemment été possible d'utiliser directement un objet triangle, auquel cas le programme aurait été le suivant :

Exemple 1 – revu en version objet avec une classe triangle

```
Programme aire
Avec tri :      Triangle
```

```
Début
      Saisir tri
      Afficher "l'aire vaut : ", tri.Aire()
fin
```

De façon générale, la structure de donnée utilisée devient secondaire et reste complètement cachée à ce que l'on appelle communément le **client** (ou l'utilisateur) de l'objet. Un objet n'est vu qu'à travers ses **méthodes** (celles-ci sont des sortes de fonctions et procédures, mais attention : il s'agit d'une chose très différente !). Une méthode est un service. Elle se rapporte toujours à la forme abstraite de l'objet qui est la **classe** – en fait, elle en fait partie intégrante, c'est ce qui explique l'écriture suivante : tri.Aire(). Ici, *tri* est une instance de la classe *Triangle*. Ce que l'on veut c'est appliquer la méthode *Aire* (qui calcule donc la superficie) définie au sein de la classe *Triangle*, à un objet bien particulier *tri*. Ceci ce fait au travers d'un mécanisme complexe qui peut être traduit par le biais de messages. Si un service "Aire" existe et est accessible pour l'objet particulier *tri*, alors cette demande sera honorée et un résultat fourni en retour. De manière un peu plus concrète, *tri.Aire()* doit être lu et compris comme : essayer d'appliquer la méthode (ou le service) *Aire* à l'objet particulier *tri* qui est une instance de la classe *Triangle*.

Evidemment deux cas de figure peuvent se présenter alors :

- La méthode requise existe et est accessible à l'objet en question, auquel cas elle fournit un résultat correspondant à sa spécification.
- La méthode n'existe pas ou n'est pas accessible à cet objet, auquel cas une exception est générée. Le résultat dépend alors du langage de programmation.

6. Caractéristiques d'une approche objet

6.1. Abstraction et classe

Une abstraction peut être vue comme une description simplifiée d'un système mettant l'accent sur certains détails ou propriétés de celui-ci. Ainsi, une bonne abstraction met-elle l'accent sur les éléments réellement les plus significatifs pour l'utilisateur en supprimant temporairement les éléments secondaires.

Une abstraction doit faire ressortir les caractéristiques essentielles d'un objet réel, en d'autres termes les caractéristiques qui permettent de le distinguer de tous les autres objets. Ceci permet de procurer des frontières conceptuelles bien définies.

Dans l'approche objet, une classe est un modèle pour plusieurs objets à particularités similaires. Les classes réunissent donc toutes les caractéristiques d'un ensemble d'objets. Une classe "arbre" peut décrire toutes les caractéristiques d'un arbre : feuilles, racines, branches, etc. Une **instance de classe** est un **objet réel**, alors que **la classe** reste une **représentation abstraite**.

L'abstraction d'un système permettre de faire ressortir les éléments les plus pertinents. Par exemple dans un système de contrôle de flux frontalier, une abstraction pertinente décrit le comportement des individus traversant la frontière : origine, type de

motorisation, fréquence de passage et non des détails complètement inutiles à ce système comme le nom ou la couleur de cheveux des individus.

Communément, on distingue les trois genres d'abstractions (il s'agit d'une liste non exhaustive) :

- **Abstraction d'entité :** représente un modèle d'une entité du monde du problème ou de la solution ;
- **Abstraction d'action :** procure un ensemble généralisé d'opérations réalisant toutes le même genre de fonctions (affichage, saisie, etc.) ;
- **Abstraction de coïncidence :** regroupe un certain nombre d'opérations qui n'ont pas de relation les unes avec les autres (librairie C).

6.2. Encapsulation

L'encapsulation est la mise en œuvre de l'abstraction. Le principe d'encapsulation consiste à cacher à l'utilisateur d'un objet certains détails de ce dernier. La structure interne de l'objet (les champs et leur valeur) ainsi que le codage des méthodes (type d'algorithme utilisé, etc.) n'a et ne doit avoir aucun intérêt pour l'utilisateur de l'objet, sauf dans des cas très particuliers. Le mécanisme est mis en œuvre de manière pratique par l'interdiction formelle d'accéder à des informations lorsque celles-ci sont déclarées **privées**. Inversement les informations ne sont qu'accessibles dans le cas d'une déclaration **publique**.

Il est fondamental que le concepteur et développeur du système fasse la séparation entre la **spécification de l'objet** (profile des méthodes permettant d'utiliser l'objet) et son **implantation physique** (données + algorithmes).

Dans l'exemple de l'aire du triangle du chapitre précédent, la conception interne de l'objet *tri* n'a pas à intéresser l'utilisateur de cet objet. Par contre, il est nécessaire qu'il connaisse l'ensemble des "services" accessibles et la manière dont il peut y accéder. Ceci se fait par le biais du profil des méthodes. Ces méthodes ont quant à elles évidemment le droit d'accéder à toutes les informations privées pour pouvoir réaliser la spécification.

6.3. Hiérarchisation et héritage

Lorsque l'on travaille avec des ensembles de classes plus ou moins complexes, on est très vite amener à créer des hiérarchisations sémantiques. Deux cas peuvent se produire :

- Soit une classe est **spécialisée** : par exemple une classe véhicule se spécialise en voiture, camion, moto, etc.
- Soit plusieurs classes ayant trait à une même catégorie sont regroupées dans une classe plus générale comme les classes : sapin, épicéa, cèdre, etc. en une classe conifères, elle-même pouvant être **généralisée** en une classe arbre etc.

Ces généralisations et spécialisations se font lors de la phase d'analyse et conception du système. Elles permettent de construire une description hiérarchisée de classes unifiées

par la notion **d'héritage**. Cela signifie qu'une classe peut hériter des propriétés de sa classe mère (aussi appelée **super-classe**), comme les données, les méthodes, etc. Celles-ci peuvent alors être modifiées (surcharge, filtrage, restriction, etc.) ou de nouvelles ajoutées pour préciser une spécificité de comportement. Ce mécanisme permet entre autres, de forcer une structuration plus méthodique, tout en évitant des redondances inutiles de code. La relation classe et sous-classe est souvent aussi appelée une relation de **dérivation**. On dit que la sous-classe dérive de sa super-classe.

Dans certains langages, comme Java ou Smalltalk toutes les classes dérivent d'une super-classe qui se trouve toujours tout au sommet de la hiérarchie. Notons que dans la hiérarchie de classes, plus on avance vers le sommet plus les concepts sont abstraits et plus on descend, plus ils sont concrets.

Remarque : il est important de ne **pas confondre dérivation avec composition**. En effet, une classe donnée peut se composer d'autres classes sans pour autant en dériver (c'est-à-dire sans qu'il n'y ait aucune relation hiérarchique). Par exemple une voiture dérive d'une classe véhicule. Elle comporte quatre roues, un moteur, etc. Roue et moteur représentent des classes à part entière. Cependant ces classes n'entrent pas dans une relation hiérarchique avec voiture, elles la composent simplement.

6.4. Polymorphisme

Le polymorphisme consiste en la concrétisation de l'abstraction d'action. Un même nom de méthode peut être attribué à des classes différentes qu'elles soient liées par une relation d'héritage ou non.

Dans le cas de classes non liées par héritage, l'intérêt du polymorphisme est essentiellement celui de pouvoir abstraire une méthode. Prenons par exemple deux classes A et B, chacune possédant une méthode d'affichage de même nom *affiche()*. Il est à présent possible de créer des listes d'éléments d'instances de ces deux classes : une liste comprenant à la fois des objets A et B. Une méthode, permettant l'affichage de la liste, consistera simplement en un appel pour chacun de ses éléments de la méthode *affiche()*. Comme chaque objet "porte" non seulement ses propres données mais également l'ensemble de ses méthodes (en fait une table d'adresse de méthodes sans évidemment dupliquer chacune d'elles), le système pourra sans aucune difficulté retrouver la méthode *affiche()* correspondant à l'objet en question (soit la méthode affiche pour A soit celle pour B). Le système permet de différencier les méthodes à la fois grâce aux classes auxquelles elles se rapportent et à la fois, au sein d'une même classe, grâce au profil. En effet, le polymorphisme est également valable au sein d'une même classe : on parle de **surcharge** au sein d'une même classe. Cependant dans ce cas, il faut nécessairement que les profils des méthodes soient différents.

Dans le cas de classes liées par relation d'héritage, le polymorphisme permet d'effectuer des spécialisations ou au contraire des restrictions. Si la classe B dérive de A, et que A possède une méthode *f()*, alors B peut :

- soit également posséder sa propre méthode *f()* (on parle aussi d'une surcharge, mais à travers l'héritage) ; il s'agira d'une spécialisation ou d'une restriction. La méthode *f()* propre à B peut : soit compléter celle de A en ajoutant du code

supplémentaire tout en conservant le code hérité de A, soit au contraire complètement remplacer la méthode de A en proposant un tout nouveau code restreint à B.

- soit ne pas posséder de méthode *f()*, auquel cas celle de A est héritée telle quelle.

Dans tous les cas la méthode *f()* est accessible à un objet B. Lorsque la méthode *f()* est appelée par une instance de B, le système regarde donc d'abord s'il existe un *f()* propre à B, sinon il prendra le *f()* du niveau supérieur de la hiérarchie, etc. jusqu'à arriver au sommet.

6.5. Généricité

La généricité n'est pas un élément fondamental à l'approche objet, contrairement aux points présentés précédemment. Il s'agit d'un élément complémentaire qui n'existe pas dans tous les langages (il existe en C++, pas en Java). Le principe de la généricité c'est de créer des modèles (ou des patrons) soit de classes soit d'algorithmes. La généricité permet en l'occurrence de paramétrer des classes ou des algorithmes par un ou plusieurs types non encore existant (qui seront préciser ultérieurement). Le but ultime c'est bien sûr d'abstraire la conception au plus possible, pour pouvoir éviter de réécrire par exemple du code inutilement.

Une classe générique ne peut pas être utilisée directement puisqu'il faut d'abord l'instancier à des types concrets. Le paramétrage peut être de deux types : soit par des classes non encore définies, soit par des constantes de types de bases (entiers, réels, booléens, etc.). On peut par exemple définir un type pile de "n'importe quoi" où la taille est limitée à une constante donnée. Voici la spécification formelle paramétrée (sans axiomatique) :

SORTE pile [TYPE, N : entier] // pile de n'importe quoi de taille *N* limitée
UTILISE entier, booléen
OPERATION

pilenouv :		→	pile
empiler :	TYPE x pile	→	pile
depiler :	pile	→	pile
sommet :	pile	→	TYPE
hauteur :	pile	→	entier
vide :	pile	→	booléen

PRECONDITIONS
empiler(p,x) ⇔ hauteur(p)<N
depiler(p) ⇔ non vide(p)
sommet(p) ⇔ non vide(p)

Dans un langage comme C++, il est possible de définir des classes paramétrées par des types non encore connus et par des constantes. Il suffit pour cela de faire précéder la classe par le mot *template* suivi de la liste des paramètres.