Algorithmique et programmation : les bases

Résumé

Ce document décrit les éléments de base de notre langage algorithmique (la structure d'un algorithmique, les variables, les types, les constantes, les expressions et les instructions) et la manière de les décrire dans le langage C qui sera utilisé pour la partie programmation.

Table des matières

Liste des exercices

1 Pourquoi définir notre langage algorithmique ?

Plusieurs raisons justifient l'utilisation d'un langage algorithmique spécifique :
- bien insister sur la différence entre programmation et construction d'une solution algorithmique.
- être indépendant d'un langage de programmation particulier : même si ce langage est proche de Pascal dans sa structure, il n'est pas identique, et peut être très facilement traduit dans d'autres langages de programmation tels que C, Ada, Modula-2, etc.
- utiliser des concepts intéressants issus de plusieurs langages comme par exemple :
  - les structures de contrôle de Pascal avec les mots-clés de fin de structure de Modula-2 ;
  - les modes de passages des paramètres aux sous-programmes d'Ada ;
  - la variable prédéfinie **`Résultat`** d'Eiffel pour les fonctions, etc.
- favoriser la créativité en ayant un langage souple (permettant par exemple des instructions abstraites) mais suffisamment rigoureux pour que tout le monde puisse comprendre un algorithme écrit dans ce langage.

2 Structure d'un algorithme

Un algorithme (comme un programme) est composé de trois parties principales :

1. La partie **définitions** permet de définir les « entités » qui pourront être manipulées dans l'algorithme. En particulier, on définit des constantes, des types et des sous-programmes.
2. La partie **déclarations** permet de déclarer les données qui sont utilisées par le programme. Les données sont représentées par des variables.
3. La partie **instructions** constitue le **programme principal**. Les instructions sont exécutées par l'ordinateur pour transformer les données.

2.1 Exemple d'algorithme : calculer le périmètre d'un cercle

Un exemple d'algorithme est donné dans le listing 1. Il décrit comment obtenir le périmètre d'un cercle à partir de son diamètre. Cet exemple est volontairement très simple.

Listing 1 – Algorithme pour calculer le périmètre d'un cercle

```
Algorithme périmètre_cercle

    -- Déterminer le périmètre d'un cercle à partir de son rayon

    -- Remarque : aucun contrôle n'est fait lors de la saisie du rayon.
    -- Cet algorithme/programme n'est donc pas robuste.

Constante
    PI = 3.1415

Variable
    rayon: Réel          -- le rayon du cercle lu au clavier
```

```
    périmètre: Réel     -- le perimètre du cercle

Début
    -- Saisir le rayon
    Écrire("Rayon = ")
    Lire(rayon)

    -- Calculer le périmètre
    périmètre ← 2 * PI * rayon          -- par définition
    { périmètre = 2 * PI * rayon }

    -- Afficher le périmètre
    Écrire("Le périmètre est : ", périmètre)
Fin
```

2.1.1 Structure de l'algorithme

L'algorithme est composé d'un nom. Ensuite, un commentaire général explique l'objectif de l'algorithme.

Dans le partie « définition », nous avons défini la constante PI.

Dans la partie « déclaration », deux variables sont déclarées pour représenter respectivement le rayon du cercle et son périmètre.

Dans la partie « instruction », les instructions permettent d'afficher (**Écrire**) des informations à l'utilisateur du programme (« Rayon = ») et de lui demander de saisir la valeur du rayon (**Lire**) et d'initialiser la variable périmètre. L'algorithme se termine avec l'affichage du périmètre pour que l'utilisateur puisse le voir sur l'écran.

Exercice 1 : Lien entre raffinage et algorithme
Donner le raffinage qui a conduit à l'algorithme décrit dans le listing 1.

2.1.2 Identificateurs

Les entités qui apparaissent (le programme, les variables, les constantes, les types, les sous-programmes, etc.) doivent avoir un nom. Ce nom permet à l'ordinateur de les distinguer et aux hommes de les comprendre et de les désigner. Les noms ainsi donnés sont appelés **identificateurs**. Un identificateur commence par une lettre ou un souligné (_) et se continue par un nombre quelconque de lettres, chiffres ou soulignés.

2.1.3 Commentaires

Notre langage algorithmique propose deux types de commentaires, l'un introduit par deux tirets et qui se termine à la fin de la ligne et l'autre délimité par des accolades. Nous donnerons une signification particulière à ces deux types de commentaires :

– Les commentaires introduits par deux tirets seront utilisés pour faire apparaître les raffinages, donc la structure de l'algorithme, dans l'algorithme lui-même. Les commentaires précèdent et sont alignés avec les instructions qu'ils décrivent.

Ils peuvent être également utilisés pour expliquer ce qui a été fait. Dans ce cas, ils sont placés à la fin de la ligne et ont un rôle de justification. Ils sont également utilisés pour expliquer le rôle d'une variable ou d'une constante, etc.
- Les commentaires entre accolades seront utilisés pour mettre en évidence une propriété sur l'état du programme. Ces commentaires contiennent en générale une expression booléenne qui doit être vraie à cet endroit du programme.

Dans l'exemple du calcul du périmètre d'un cercle (listing 1), nous trouvons ces différents types de commentaires :
- le premier commentaire explique l'objectif du programme (R0) ;
- les commentaires « saisir le rayon », « calculer le périmètre » et « afficher le périmètre » correspondent aux étapes identifiées lors du raffinage ;
- les commentaires « le rayon du cercle lu au clavier » et « le périmètre du cercle » sont des commentaires expliquent le rôle de la variable ;
- le commentaire *{ périmètre = 2 * PI * rayon }* indique qu'à ce moment du programme, la valeur de la variable périmètre est la même que celle de l'expression `2 * PI * rayon`.

2.2 Exemple de programme en C : calculer le périmètre d'un cercle

L'algorithme précédent peut être traduit dans le langage C. On obtient alors le programme suivant.

Listing 2 – Programme C pour calculer le périmètre d'un cercle

```c
/*****************************************************************
 * Auteur  : Xavier Crégut <cregut@enseeiht.fr>
 * Version : 1.1
 * Titre   : Déterminer le périmètre d'un cercle à partir de son rayon.
 *****************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define PI 3.1415

int main()
{
    double rayon;       /* le rayon du cercle lu au clavier */
    double perimetre;   /* le perimètre du cercle */

    /* Saisir le rayon */
    printf("Rayon = ");
    scanf("%lf", &rayon);

    /* Calculer le périmètre */
    perimetre = 2 * PI * rayon;         /* par définition */
    /*{ perimetre == 2 * PI * rayon }*/

    /* Afficher le périmètre */
```

```
    printf("Le périmètre est : %4.2f\n", perimetre);

    return EXIT_SUCCESS;
}
```

La structure d'un programme C est proche de celle d'un algorithme. Le fichier, qui doit avoir l'extension `.c`, commence par un cartouche faisant apparaître le nom des auteurs du programme, la version ou la date de réalisation et l'objectif du programme. Ces éléments sont mis dans des commentaires et sont donc ignorés par le compilateur.

Les `#include` correspondent à des directives qui indiquent au compilateur (en fait au pré-processeur) d'inclure les fichiers nommés `stdio.h` et `stdlib.h`. Ces fichiers font parties de la bibliothèque standard du C et donne accès à des fonctions déjà définies. Par exemple les fonctions d'affichage (`printf`) et de lecture (`scanf`) sont définies dans `stdio.h`. La constante `EXIT_SUCCESS` est définie dans `stdlib.h`. Si on ne met pas ces `#include`, on ne peut pas utiliser ces fonctions ou constantes.

Le `#define` suivant permet de définir la constante PI. Il correspond donc à une définition.

Finalement, les déclarations et instructions sont regroupées entre les accolades qui suivent `int main()`, d'abord les déclarations, puis les instructions. `main` est la fonction principale, c'est-à-dire que c'est elle qui est exécutée quand le programme sera lancé. Les instructions sont les mêmes que cette présentées dans l'algorithme même si elles ont une forme un peu différente.

Notons que les identificateurs respectent la même règle qu'en algorithmique mais qu'il n'est pas possible d'utiliser les lettres accentuées en C.

En revanche les commentaire ce note de manière différente. Ils sont entre `/* ... */` et ne peuvent pas être imbriqués. Pour représenter une propriété du programme, nous utiliserons `/*{... }*/`.

En C++ : Le langage C++ ajoute les commentaires qui commencent par `//` et se termine avec la fin de la ligne (comme `--`). Ils peuvent être utilisés la où on met `--` en algorithmique.

3 Variables

3.1 Qu'est ce qu'une variable ?

Un algorithme est fait pour résoudre un ensemble de problèmes semblables (cf définition du terme « algorithmique » dans le petit Robert). Par exemple, un logiciel qui gère la facturation d'une entreprise doit connaître les noms et adresses des clients, les produits commandés, etc. Ces informations ne peuvent pas être devinées par le programme et doivent donc être saisies par les utilisateurs. Ces informations sont appelées *données en entrée*. Elles proviennent de l'extérieur du programme et sont utilisées dans le programme.

Inversement, le programme effectue des opérations, des calculs dont les résultats devront être transmis aux utilisateurs. Par exemple, le total d'une facture est calculée comme la somme de tous les produits commandés en multipliant le nombre d'articles par leur prix unitaire ; le total des ventes pour une période donnée est obtenu en faisant la somme des montants des factures,

etc. Ces informations seront affichées à l'écran, imprimées, stockées dans des bases de données, etc. Ce sont des informations qui sortent du programme. On les appelle *données en sortie*.

Le programmeur, pour décrire son algorithme utilise des variables pour représenter les données manipulées par un programme. Ce peut être les données en entrée, les données en sortie mais également les données résultant de calculs intermédiaires. Ainsi, pour calculer le montant d'une ligne d'une facture, le programmeur expliquera que le montant est obtenu en multipliant le prix unitaire par la quantité d'articles commandés. Il utilisera donc trois variables :

- `prix_unitaire`, le prix unitaire de l'article ;
- `quantité`, la quantité d'articles commandé ;
- et `montant`, le montant de la ligne du bon de commande.

3.2 Définition d'une variable

Une variable peut être vue comme une zone dans la mémoire (vive) de l'ordinateur qui est utilisée pour conserver les données qui seront manipulées par l'ordinateur.

Une variable est caractérisée par quatre informations :

- son **rôle** : il indique à quoi va servir la variable dans le programme. Par exemple, on peut utiliser une variable pour conserver le prix unitaire d'un article, prix exprimé en francs. Cette information doit apparaître dans l'algorithme sous la forme d'un commentaire informel (texte libre) associé à la variable ;
- son **nom** : composé uniquement de lettres minuscules, majuscules, de chiffres et du caractère souligné, il permet d'identifier la variable. On parle d'« identificateur ». Ce nom doit être significatif (refléter le rôle de la variable). Un compromis doit bien sûr être trouvé entre expressivité et longueur. On peut par exemple appeler `prix_unitaire` une variable qui représente le prix unitaire d'un article. Le nom `prix_unitaire_d_un_article` est un nom beaucoup trop long ;
- son **type** : une variable est utilisée pour représenter des données qui sont manipulées par le programme. Un type est utilisé pour caractériser l'ensemble des valeurs qu'une variable peut prendre. Par exemple le prix unitaire représente le prix d'un article exprimé en francs. On peut considèrer qu'il est représenté par un réel. Il ne peut alors prendre que ce type de valeurs. De la même manière la quantité, ne peut prendre que des valeurs entières et le nom est une chaîne de caractères. On dira respectivement que le type de `prix_unitaire` est **`Réel`**, le type de `quantité` est **`Entier`** et le type de `nom` est **`Chaîne`**.

```
prix_unitaire: Réel      -- prix unitaire d'un article (en euros)
quantité: Entier         -- quantité d'articles commandés
nom: Chaîne              -- nom de l'article
```

En C, on commence par mettre le type suivi du nom de la variable et un point-virgule.

```
double prix_unitaire;   /* prix unitaire d'un article (en euros) */
int quantite;           /* quantité d'articles commandés */
char nom[20];           /* nom de l'article */
```

Les types et leur significations seront présentés dans la suite du cours.

Il est possible de déclarer plusieurs variables du même type en les séparant par des virgules mais ceci est déconseillé sauf si le même commentaire s'applique à toutes les variables.

```
int a, b, c;    /* trois entiers */
```

– sa **valeur** : La variable contient une information qui peut varier au cours de l'exécution d'un programme. C'est cette information que l'on appelle valeur de la variable. La valeur d'une variable doit correspondre au type de la variable. Ainsi, une variable `quantité` de type entier pourra prendre successivement les valeurs de 10, 25 et 3.

La valeur d'une variable n'existe que lorsque le programme est exécuté. Les autres informations (nom, rôle et type) sont définies lors de la conception du programme, pendant la construction de l'algorithme. Le rôle, le nom et le type sont des informations statiques qui doivent être précisées lors de la déclaration de la variable. En revanche, la valeur est une information dynamique qui changera au cours de l'exécution du programme.

Règle : Une variable doit toujours être initialisée avant d'être utilisée.

Définition : Une variable est un nom désignant symboliquement un emplacement mémoire typé auquel est associé une valeur courante. Cette valeur peut être accédée[1] (expressions, section 6) ou modifiée (affectation, section 8).

Attention : Le nom d'une variable doit être significatif : il doit suggérer, si possible sans ambiguïté, la donnée représentée par cette variable.

Règle : En général, dès qu'on identifie une variable, il faut mettre un commentaire qui explique ce qu'elle représente et le rôle qu'elle joue dans l'algorithme.

L'ensemble de variables et leur description constitue le **dictionnaire des données**.

4 Types fondamentaux

Définition : Un type caractérise les valeurs que peut prendre une variable. Il définit également les opérations, généralement appelées opérateurs, qui pourront être appliquées sur les données de ce type.

On appelle types fondamentaux les types qui sont définis dans notre langage algorithme par opposition aux types structurés (tableaux, enregistrements et types énumérés) qui doivent être définis par le programmeur.

Intérêts : Les types ont deux intérêts principaux :

– Permettre de vérifier automatiquement (par le compilateur) la cohérence de certaines opérations. Par exemple, une valeur définie comme entière ne pourra par recevoir une valeur chaîne de caractères.

– Connaître la place nécessaire pour stocker la valeur de la variable. Ceci est géré par le compilateur du langage de programmation considéré et est en général transparent pour le programmeur.

[1] Le verbe « lire » est parfois utilisé en informatique pour indiquer que l'on accède à la valeur de la variable. Il ne faut pas confondre ce « lire » avec l'instruction d'entrée/sortie de même nom et qui permet d'initialiser une variable à partir d'une valeur lue sur périphérique d'entrée tel que le clavier. Il ne faut donc pas confondre les deux sens de « lire » : « accèder à la valeur d'une variable » et « saisir la valeur d'une variable ».

Opérateur : À chaque type est associé un ensemble d'opérations (ou opérateurs).
Opérateurs de comparaison : Tous les types présentés dans cette partie sont munis des opérations de comparaison suivantes : <, >, <=, >=, = et <>.

Ils se notent respectivement en C : <, >, <=, >=, == et !=. Notez bien que l'égalité est notée avec deux fois le caractère =.

Les opérateurs de comparaison sont des opérateurs à valeur booléenne (cf type booléen, section 4.3).
Attention : Tous les types manipulés par une machine sont discrets et bornés. Ces limites dépendent du langage de programmation et/ou de la machine cible considérés. Aussi, nous n'en tiendrons pas compte ici. Il faut cependant garder à l'esprit que toute entité manipulée est nécessairement bornée.

4.1 Les entiers

Le type **Entier** caractérise les entiers relatifs.

```
-- Exemple de représentants des entiers
10
0
-10
```

Les opérations sont : +, -, *, **Div** (division entière), **Mod** (reste de la division entière) et **Abs** (valeur absolue)

- et + peuvent être unaires ou binaires.

```
10 Mod 3  -- 1 (le reste de la division entière de 10 par 3)
10 Div 3  -- 3 (le quotient de la division entière de 10 par 3)
1 Div 2   -- 0 (le quotient de la division entière de 1 par 2)
Abs(-5)   -- 5 (l'entier est mis entre parenthèses (cf sous-programmes))
```

En C, le type entier se note **int**. Cependant, des qualificatifs peuvent venir préciser :

– sa taille, c'est-à-dire le nombre d'octets sur lequel il est représenté (2 octets pour **short**, 4 pour **long**). La taille d'un **int** est comprise entre celle d'un **short int** d'un **long int**. Notons que **int** est optionnel quand on utilise **short** et **long**.

```
short int a;    /* un entier court */
short a;        /* également un entier court (int est implicite) */
long l;         /* un entier long (int est aussi implicite) */
```

– s'ils sont signés ou non. Par défaut, les entiers sont signés (positifs ou négatif). Si l'on précise **unsigned** devant le type, ils ne peuvent pas être négatifs.

```
unsigned int n;         /* un entier non signé */
unsigned short s;       /* un entier court non signé */
```

Le reste de la division entière se note % et la division entière se note tous simplement /. Il faut faire attention à ne pas la confondre avec la division sur les réels.

```
10 % 3  /* 1 (le reste de la division entière de 10 par 3) */
10 / 3  /* 3 (le quotient de la division entière de 10 par 3) */
1 / 2   /* 0 (le quotient de la division entière de 1 par 2) */
```

```
abs(-5) /* 5 (l'entier est mis entre parenthèses (cf sous-programmes)) */
```

Notons que les débordement de capacité sur les opérations entières ne provoquent aucune erreur à l'exécution... mais le résultat calculé est bien sûr faux par rapport au résultat attendu !.
Remarque : Le type **char** (caractère, section 4.4) fait partie des entiers.

4.2 Les réels

Le type **Réel** caractérise les réels.

```
-- Exemple de représentants des réels
10.0
0.0
-10e-4
```

Les opérations sont : +, -, *, /, **Abs** (valeur absolue), Trunc (partie entière).
- et + peuvent être unaires ou binaires.
En C, il existe deux types réels, les réels simple précision appelés **float** et les réels double précision appelés **double**.

La valeur absolue se note fabs. Elle prend en paramètre un **double** et retourne un **double**. Pour pouvoir l'utiliser, il faut ajouter en début de fichier **#include** <math.h>. Dans ce même module sont définies la racine carrée (sqrt), les fonctions trigonométriques (sin, cos, etc.)...

La partir entière d'un réel s'obtient en faisant un cast : (**int**) 3.14 correspond à 3. Ceci correspond à convertir en entier le réel 3.14.

4.3 Les booléens

Le type **Booléen** caractérise les valeurs booléennes qui ne correspondent qu'à deux valeurs **VRAI** ou **FAUX**.

Les opérations sont **Et**, **Ou** et **Non** qui sont définies par la table de vérité suivante :

A	B	A **Et** B	A **Ou** B	**Non** A
VRAI	**VRAI**	**VRAI**	**VRAI**	**FAUX**
VRAI	**FAUX**	**FAUX**	**VRAI**	**FAUX**
FAUX	**VRAI**	**FAUX**	**VRAI**	**VRAI**
FAUX	**FAUX**	**FAUX**	**FAUX**	**VRAI**

Lois de De Morgan : Les lois de De Morgan sont particulièrement importantes à connaître (cf structures de contrôle, section 9).

```
Non (A Et B) = (Non A) Ou (Non B)
Non (A Ou B) = (Non A) Et (Non B)
Non (Non A)  = A
```

Exercice 2 : Vérifier les formules de De Morgan
En utilisant des tables de vérité, vérifier que les formules de De Morgan sont correctes.

Remarque :

```
A: Booléen      -- A est une variable de type Booléen

A     est équivalent à A = VRAI
Non A est équivalent à A = FAUX
```

Il est préférable d'utiliser `A` et **`Non`** `A`. Les deux autres formulations, si elles ne sont pas fausses, sont des pléonasmes !

En C, le type booléen n'existe pas. C'est le type entier qui remplace les booléens avec la convention suivante : 0 correspond à FAUX, tous le reste à VRAI. Il faut donc comparer par rapport à 0 et non par rapport à 1 ou un autre entier non nul !

Cependant, il existe un module standard (`<stdbool.h>`) qui définit un type booléen `bool` avec les deux valeurs `true` et `false`.

Les opérateurs logiques se notent `&&` pour **`Et`**, `||` pour **`Ou`** et `!` pour **`Non`**.

```
&&      /* ET logique          expr1 && expr2  */
||      /* OU logique          expr1 || expr2  */
!       /* NON logique         ! expr1         */
```

Les expressions booléennes sont évaluées en court-circuit (on parle d'évaluation partielle), c'est-à-dire que dès que le résultat d'une expression est connu, l'évaluation s'arrête. Par exemple, `true || expression` sera évaluée à `true` sans calculer la valeur de expression.

4.4 Les caractères

Le type **`Caractère`** caractérise les caractères.

```
-- Exemple de représentants des caractères
'a'         -- le caractère  a
'\''        -- le caractère  '
'\\'        -- le caractère  \
'\n'        -- retour à la ligne
'\t'        -- Caractère tabulation
```

Remarque : On considère que les lettres majuscules, les lettres minuscules et les chiffres se suivent. Ainsi, pour savoir si un variable `c` de type caractère correspond à un chiffre, il suffit d'évaluer l'expression `(c >= '0')` **`Et`** `(c <= '9')`.

Opérations :

- `Ord` : Permet de convertir un caractère en entier.
- `Chr` : Permet de convertir un entier en caractère.

Pour tout `c:` **`Caractère`** on a `Chr(Ord(c)) = c`.

En C, le type caractère se note **`char`**. Les constantes caractères se notent comme en algorithmique. Cependant, le type **`char`** est un fait un type entier et sa valeur est le code ASCII du caractère. Il n'y a donc pas de fonctions `Chr` et `Ord`.

```
char c;
int i;
c = 'A';        /* la valeur de c est 'A' */
i = c;          /* la valeur de i est 65, code ASCII de 'A' */
```

Enfin, si c est un caractère correspondant à un chiffre (`c >= '0'&& c <= '9'`), `c - '0'` est la valeur entière de ce chiffre (entre 0 et 9).

4.5 Les chaînes de caractères

Le type **Chaîne** caractérise les chaînes de caractères.

```
"Une␣chaîne␣de␣caractères"          -- un exemple de Chaîne
"Une␣chaîne␣avec␣guillement␣(\")"   -- un exemple de Chaîne
```

Remarque : Nous verrons plus en détail les chaînes de caractères lorsque nous traiterons de la structure de données « tableau ».
Attention : Il ne faut pas confondre le nom d'une variable et une constante chaîne de caractères !

5 Constantes

Certaines informations manipulées par un programme ne changent jamais. C'est par exemple la cas de la valeur de π, du nombre maximum d'étudiants dans une promotion, etc.

Ces données ne sont donc pas variables mais constantes. Plutôt que de mettre explicitement leur valeur dans le texte du programme (constantes littérales), il est préférable de leur donner un nom symbolique (et significatif). On parle alors de constantes (symboliques).

```
PI = 3.1415      -- Valeur de PI
MAJORITÉ = 18    -- Âge correspondant à la majorité
TVA = 19.6       -- Taux de TVA en vigueur au 15/09/2000 (en %)
CAPACITÉ = 120   -- Nombre maximum d'étudiants dans une promotion
INTITULÉ = "Algorithmique␣et␣programmation"     -- par exemple
```

π peut être considérée comme une constante absolue. Son utilisation permet essentiellement de gagner en clarté et lisibilité. Les constantes MAJORITÉ, TVA, CAPACITÉ, etc. sont utilisées pour paramètrer le programme. Ces quantités ne peuvent pas changer au cours de l'exécution d'un programme. Toutefois, si un changement doit être réalisé (par exemple, la précision de PI utilisée n'est pas suffisante), il suffit de changer la valeur de la constante symbolique dans sa définition (et de recompiler le programme) pour que la modification soit prise en compte dans le reste de l'algorithme.[2]

En C, les constantes sont définies en utilisant **#define** :

```
#define PI 3.1415       /* Valeur de PI */
#define MAJORITÉ 18     /* Âge correspondant à la majorité */
#define TVA 19.6        /* Taux de TVA en vigueur au 15/09/2000 (en %) */
#define CAPACITÉ 120    /* Nombre maximum d'étudiants dans une promotion */
#define INTITULÉ "Algorithmique␣et␣programmation"       /* par exemple */
```

[2] Ceci suppose d'arrêter le programme, de le recompiler et de l'exécuter à nouveau.

Attention : Ne surtout pas mettre de point-virgule (« ; ») après la déclaration d'une constante avec **`#define`**. **`#define`** n'est pas traitée par le compilateur mais par le préprocesseur qui fait bêtement du remplacement de texte. Le point-virgule provoquera donc des erreurs là où est utilisée la macro et non où elle est définie !
Question : Quel est l'intérêt d'une constante symbolique ?

6 Expressions

Définition : Une expression est « quelque chose » qui a une valeur. Ainsi, en fonction de ce qu'on a vu jusqu'à présent, une expression est une constante, une variable ou toute combinaison d'expressions en utilisant les opérateurs arithmétiques, les opérateurs de comparaison ou les opérateurs logiques.
Exemple : Voici quelques exemples de valeurs réelles :

```
10.0            -- une constante litérale
PI              -- une constante symbolique
rayon           -- une variable de type réel
2*rayon         -- l'opérateur * appliqué sur 2 et rayon
2*PI*rayon      -- une expression mêlant opérateurs, constantes et variables
rayon >= 0      -- expression avec un opérateur de comparaison
```

Attention : Pour qu'une expression soit acceptable, il est nécessaire que les types des opérandes d'un opérateur soient compatibles. Par exemple, faire l'addition d'un entier et d'un booléen n'a pas de sens. De même, on ne peut appliquer les opérateurs logiques que sur des expressions booléennes.

Quelques règles sur la compatibilité :

- Deux types égaux sont compatibles.
- On peut ensuite prendre comme règle : le type A est compatible avec le type B si et seulement si le passage d'un type A à un type B se fait sans perte d'information. En d'autres termes, tout élément du type A a un correspondant dans le type B. Par exemple, **`Entier`** est compatible avec **`Réel`** mais l'inverse est faux.
  Si l'on écrit `n + x` avec `n` entier et `x` réel, alors il y a « coertion » : l'entier `n` est converti en un réel puis l'addition est réalisée sur les réels, le résultat étant bien sûr réel.

Règle d'évaluation d'une expression : Une expression est évaluée en fonction de la priorité des opérateurs qui la composent, en commençant par ceux de priorité plus forte. À priorité égale, les opérateurs sont évalués de gauche à droite.

Voici les opérateurs rangés par priorité décroissante :

`.` (accès à un champ) la priorité la plus forte
`+`, `-`, **`Non`** (unaires)
`*`, `/`, **`Div`**, **`Mod`**, **`Et`**
`+`, `-`, **`Ou`**
`<`, `>`, `<=`, `>=`, `=` et `<>` priorité la plus faible

Il est toujours possible de mettre des parenthèses pour modifier les priorité.

```
prix_ht * (1 + tva)          -- prix_ttc
```

En cas d'ambiguïté (ou de doute), il est préférable de mettre des parenthèses.

Exercice 3 : Parenthèser
Parenthéser complètement les expressions suivantes. Peuvent-elles être considérées comme correctes et si oui, à quelles conditions, si non pourquoi ?

```
2 + x * 3
- x + 3 * y <= 10 + 3
x = 0 Ou y = 0
```

Exercice 4 : Validité d'instructions
À quelles conditions les instructions suivantes sont-elles cohérentes ?

```
(a Mod b)
(rep = 'o') Ou (rep = 'n')
(R <> 'O') Et (R <> 'o')
(C >= 'A') Et (C <= 'Z') Ou (C >= 'a') Et (C <= 'z')
termine Ou (nb > 10)
Non trouve Ou (x = y)
```

En C, la priorité des opérateurs est différentes. Voici la table des priorités de C.

```
16G   -> .
15D   (unaires) sizeof ++ -- ~ ! + - * & (cast)
13G   * / %
12G   + -
11G   <<   >>
10G   <   <=   >   >=
 9G   ==   !=
 8G   &
 7G   ^
 6G   |
 5G   &&
 4G   ||
 3G   ?: (si arithmétique)
 2D   = *= /= %=  +=  -=  <<=  >>=  &= |= ^=
 1G   ,

a + b + c + d   // G : associativité à gauche ((a + b) + c) + d
x = y = z = t   // D : associativité à droite x = (y = (z = t))
```

Le dernier exemple correspond à l'affectation.
Remarque : Les fonctions que nous verrons par la suite sont également des expressions. Elles sont assimilables à des opérateurs définis par le programmeur.
Exemple : Les expressions booléennes.

Une expression booléenne peut être constituée par :
- une constante booléenne (**VRAI** ou **FAUX**) ;
- une variable booléenne ;
- une comparaison entre deux expressions de même type ($=, \neq, <, >, \leq, \geq$) ;
- la composition de une ou deux expressions booléennes reliées par un opérateur booléen (**Et**, **Ou**, **Non**).

7 Instructions d'entrée/sorties

Le point de référence est le programme. Ainsi les informations d'entrée sont les informations produites à l'extérieur du programme et qui rentrent dans le programme (saisie clavier par exemple), les informations de sortie sont élaborées par le programme et transmises à l'extérieur (l'écran par exemple).

7.1 Opération d'entrée : `Lire`

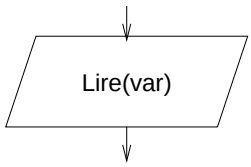
`Lire``(var)` : lire sur le périphérique d'entrée une valeur et la ranger dans la variable `var`.
Autre formulation : affecter la variable `var` avec une valeur lue au clavier.

Règle : Le paramètre « var » est nécessairement une variable. En effet, la valeur lu sur le périphérique d'entrée doit être rangée dans « var ».

Évaluation : L'évaluation se fait en deux temps :

- récupérer l'information sur le périphérique d'entrée (elle est convertie dans le type de la variable var) ;
- ranger cette information dans la variable var.

Organigramme :



En C, on utilise `scanf` qui est une fonction de saisie qui fonctionne avec un format décrivant la nature de l'information à lire et donc la conversion à effectuer.

```
char un_caractere;
int un_entier;
float un_reel;
double un_double;

scanf("%c", &un_caractere);
scanf("%d", &un_entier);
scanf("%f", &un_reel);
scanf("%lf", &un_double);
scanf("%c%d%lf", &un_caractere, &un_entier, &un_double);
```

Chaque `%` rencontré dans le format (la chaîne de caractères) est suivi d'un caractère indiquant la nature de l'information à lire (c pour caractère, d pour entier, etc.). À chaque `%` doit correspondre une variable donnée après le format. Les variables sont séparées par des virgules et sont précédées du signe `&` (voir sous-programmes). Le `&` indique que l'on donne l'adresse de la variable de qui permet à `scanf` de changer la valeur de la variable.

En C++ : En faisant **`#include`** `<iostream>`, on peut utiliser l'opérateur `>>` :

```
char un_caractere;
int un_entier;
float un_reel;
```

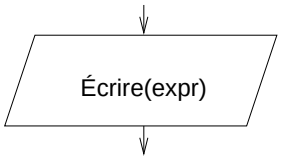
```
double un_double;

std::cin >> un_caractere;
std::cin >> un_entier;
std::cin >> un_reel;
std::cin >> un_double;
std::cin >> un_caractere >> un_entier >> un_double;
```

Notons que `std::cin` désigne l'entrée standard, le clavier par défaut.
Il n'y a plus à faire attention ni au format utilisé, ni au nombre de variables fournies.

7.2 Opération de sortie : `Ecrire`

`Écrire``(expr)` : transférer (afficher, imprimer...) la valeur de l'expression `expr` vers le périphérique de sortie.
Règle : « expr » est une expression quelconque d'un type fondamental.
Évaluation : L'évaluation se fait en deux temps :
- évaluer l'expression (c'est-à-dire calculer sa valeur) ;
- afficher (ajouter) sur le périphérique de sortie la valeur de l'expression.

Organigramme :



Variantes : Nous utiliserons une variante `EcrireLn(expr)` qui ajoute un saut de ligne sur le périphérique de sortie après avoir affiché la valeur de l'expression.

En C, on utilise `printf` qui est une fonction de saisie qui, comme `scanf`, fonctionne avec un format décrivant la nature de l'information à écrire et donc la conversion à effectuer.

```
char un_caractere = 'A';
int un_entier = 10;
float un_reel = 3.14;
double un_double = 1.10e-2;

printf("%c\n", un_caractere);
printf("%d\n", un_entier);
printf("%f\n", un_reel);
printf("%f\n", un_double);
printf("2 * %d = %d\n", un_entier, un_entier * 2);
printf("c = %c et nb = %f\n", un_caractere, un_double);
```

Notons que `std::cout` désigne la sortie standard, l'écran par défaut. `std::cerr` désigne la sortie en erreur (également reliée à l'écran par défaut).

Le programme affiche alors :

```
A
10
3.140000
```

```
0.011000
2 * 10 = 20
c = A et nb = 0.011000
```

Notons que l'on ne met pas de & devant l'expression.

En C++ : En faisant **#include** <iostream>, on peut utiliser l'opérateur << :

```
char un_caractere = 'A';
int un_entier = 10;
float un_reel = 3.14;
double un_double = 1.10e-2;

std::cout << un_caractere << std::endl;
std::cout << un_entier << std::endl;
std::cout << un_reel << std::endl;
std::cout << un_double << std::endl;
std::cout << "2 * " << un_entier << " = " << un_entier * 2 << std::endl ;
std::cout << "c = " << un_caractere
          << " et nb = " << un_double << std::endl;
```

Le programme affiche alors :

```
A
10
3.14
0.011
2 * 10 = 20
c = A et nb = 0.011
```

Remarque : **Lire** et **Écrire** sont deux « instructions » qui peuvent prendre en paramètre n'importe quel type fondamental. Si la variable est entière, **Lire** lit un entier, si c'est une chaîne de caractères, alors c'est une chaîne de caractères qui est saisie. On dit que **Lire** et **Écrire** sont polymorphes.

Malheureusement, le `scanf` et le `printf` de C ne sont pas polymorphes et c'est pour cette raison que le programmeur doit préciser le format. Les opérateurs << et >> de C++ sont, quant à eux, polymorphes.

Exercice 5 : Cube d'un réel
Écrire un programme qui affiche le cube d'un nombre réel saisi au clavier.

8 Affectation

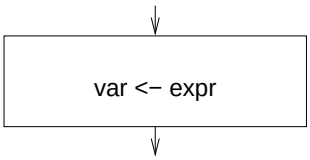
Définition : L'affectation permet de modifier la valeur associée à une variable.

`var` ← `expr`

Règle : Le type de `expr` doit être compatible avec le type de `var`.
Évaluation : L'évaluation de l'affectation se fait en deux temps :

1. calculer la valeur de l'expression `expr` ;
2. ranger cette valeur dans la variable `var`.

Organigramme :



Exemple : Voici quelques exemples d'affectation :

```
rayon ← 10
diamètre ← 2 * rayon
périmètre ← PI * rayon
```

Exercice 6 : Comprendre l'affectation
Quelle est la valeur de n après l'exécution de chacune des instructions suivantes ?

```
n ← 5
c ← 'c'
Lire (n)
n ← 10
n ← n + 1
```

En C, l'affectation se note avec un signe =.
Remarque : L'affectation peut être enchaînée : `a = b = c = 0;` consiste à initialiser c, b puis a avec la valeur 0. C'est équivalent à `a = (b = (c = 0));`.
Attention : Il ne faut pas confondre = (affectation) et == (égalité).

Il exite des formes condensées de l'affectation. Par exemple, `x = x + y` peut se noter `x += y`. Ces formes condensées fonctionnent avec la plupart des opérateurs mais elles sont à éviter dans le cas général car elles peuvent nuire à la lisibilité.

```
x += y  /* x = x + y */
x -= y  /* x = x - y */
x %= y  /* x = x % y */
x |= y  /* x = x | y */
...
```

Enfin, il existe les opérateurs de pré-incrémentation et post-incrémentation (idem avec décrémentation).

```
int i = 10;
i++;    /* postincrémentation de i */
++i;    /* préincrémentation de i */
i--;    /* postdécrémentation de i */
--i;    /* prédécrémentation de i */
```

Ces opérateurs peuvent être utilisés dans des instructions (ce qui n'est généralement pas recommandé). On parle de post ou de pré car il sont respectivement exécutés avant et avant l'instruction elle-même.

Ainsi `x = ++y;` est équivalent à :

```
++y;
x = y;
```

Prenons un exemple plus compliqué :

```
int x, y, z, t, u;
x = 3;
y = 7;
z = ++x;        /* x == 4, y == 7, z == 11, t ==  ?, u == ? */
t = y--;        /* x == 4, y == 6, z == 11, t == 11, u == ? */
u = x++ + --y;  /* x == 5, y == 5, z == 11, t == 11, u == 9 */
```

La dernière instruction est équivalente à :

```
--y;    /* décrémentation de y ==> y == 5 */
u = x + y;      /* y = 4 + 5   ==> y == 9 */
x++;    /* incrémentation de x ==> x == 5 */
```

Exercice 7 : Permuter deux caractères
Écrire un programme qui permute la valeur de deux variables `c1` et `c2` de type caractère.

Exercice 8 : Cube d'un réel (avec une variable)
Reprenons l'exercice 5.

8.1 Utiliser une variable intermédiaire pour le résoudre.

8.2 Quel est l'intérêt d'utiliser une telle variable ?

8.3 Éxecuter à la main l'algorithme ainsi écrit.

9 Structures de contrôle

Dans un langage impératif, on peut définir l'état d'un programme en cours d'exécution par deux choses :

– l'ensemble des valeurs des variables du programme ;
– l'instruction qui doit être exécutée.

L'exécution d'un programme est alors une séquence d'affectations qui font passer d'un état initial (les valeurs des variables sont indéterminées) à un état final considéré comme le résultat.

Les structures de contrôle décrivent comment les affectations s'enchaînent séquentiellement. Elles définissent dont le transfert du contrôle après la fin de l'exécution d'une instruction.
Remarque : Ceci était avant réalisé en utilisant des « goto », instruction qui a été banni en 1970 avec la naissance de la programmation structurée.

En programmation structurée, le transfert de contrôle s'exprime par :

1. enchaînement séquentiel (une instruction puis la suivante) ;
2. traitements conditionnels ;
3. traitements répétitifs (itératifs) ;
4. appel d'un sous-programme (un autre programme qui réalise un traitement particulier).

Pour chacune des structures de contrôle présentées ci-après, je donnerai la syntaxe de la structure dans notre langage algorithmique et dans le langage C, les règles à respecter (en particulier pour le typage), la sémantique (c'est-à-dire la manière dont elle doit être comprise et donc interprétée), des exemples ou exercices à but illustratifs.

9.1 Enchaînement séquentiel

Les instructions sont exécutées dans l'ordre où elles apparaissent.

```
opération_1
...
opération_n
```

Exemple :

```
tmp ← a          -- première instruction
a ← b            -- deuxième instruction
b ← tmp          -- troisième instruction
```

Question : Quand utilisera-t-on cette séquence de trois instructions ? Que permet-elle de faire ?
Remarque : On utilise parfois le terme d'instruction composée.

En C, la séquence s'exprime comme en algorithmique. Pour bien mettre en évidence une séquence d'instructions, on peut la mettre entre accolades. On parle alors de bloc d'instructions. L'intérêt des accolades est alors double :

- il permet de considérer l'ensemble des instructions dans les accolades comme une seule instruction ;
- il permet de déclarer des variables (locales à ce bloc).

9.2 Instructions conditionnelles

9.2.1 Conditionnelle `Si ... Alors ... FinSi`

```
Si condition Alors
    séquence    -- une séquence d'instructions
FinSi
```

En C,

```
if (condition)
    une_seule_instruction;

if (condition) {
    instruction_1;
    ...
    instruction_n;
}
```

Cette deuxième forme est largement préférable car dans la première on ne peut mettre qu'une seule instruction contrôlée par le **`if`** alors que dans la seconde, on peut en mettre autant qu'on veut, grâce aux accolades.

Dans la suite, nous utiliserons pour toutes les structures de contrôle la forme avec les accolades mais il existe la forme sans accolades (et donc avec une seule instruction) que nous déconseillons fortement d'utiliser !

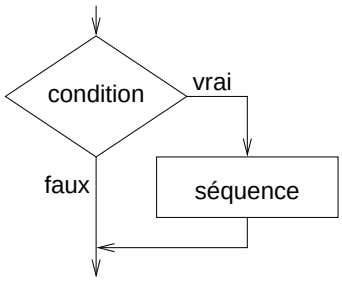
Règle : La condition est nécessairement une expression booléenne.
Évaluation :

- la condition est évaluée ;

– si la condition est vraie, la séquence est exécutée puis le contrôle passe à l'instruction qui suit le **FinSi** ;
– si la condition est fausse, le contrôle passe à l'instuction qui suit le **FinSi**.

En d'autres termes, la séquence est exécutée si et seulement si la condition est **VRAI**.

Organigramme :



Exercice 9 : Une valeur entière est-elle paire ?

Écrire un algorithme qui lit une valeur entière au clavier et affiche « paire » si elle est paire.

9.2.2 Conditionnelle `Si ... Alors ... Sinon ... FinSi`

```
Si condition Alors
    séquence1        -- séquence exécutée ssi condition est VRAI
Sinon        { Non condition }
    séquence2        -- séquence exécutée ssi condition est FAUX
FinSi
```

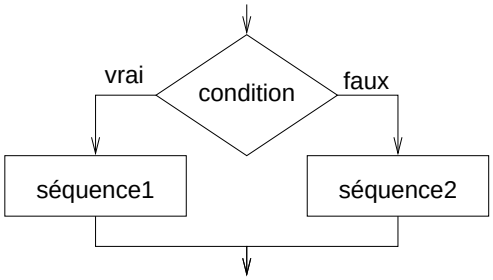
En C,

```
if (condition) {                /* séquence1 */
    instruction1,1;
    ...
}
else { /* ! condition */        /* séquence2 */
    instruction2,1;
    ...
}
```

Évaluation : Si la condition est vraie, c'est séquence_1 qui est exécutée, sinon c'est séquence_2. Dans les deux cas, après l'exécution de la séquence, l'instruction suivante à exécuter est celle qui suit le **FinSi**.

Remarque : Il est recommandé de faire apparaître sous forme de commentaire la condition associée à la partie **Sinon**.

Organigramme :



Exercice 10 : Maximum de deux valeurs réelles
Étant données deux valeurs réelles lues au clavier, afficher à l'écran la plus grande des deux.

Exercice 11 : Sinon et Si
Réécrire une instruction **`Si ... Alors ... Sinon ... FinSi`** en utilisant seulement la structure **`Si ... Alors ... FinSi`** (sans utiliser le **`Sinon`**).

Quel est l'intérêt du **`Sinon`** ?

9.2.3 La clause `SinonSi`

La conditionnelle **`Si ... Alors ... Sinon ... FinSi`** peut être complétée avec des clauses **`SinonSi`** suivant le schéma suivant :

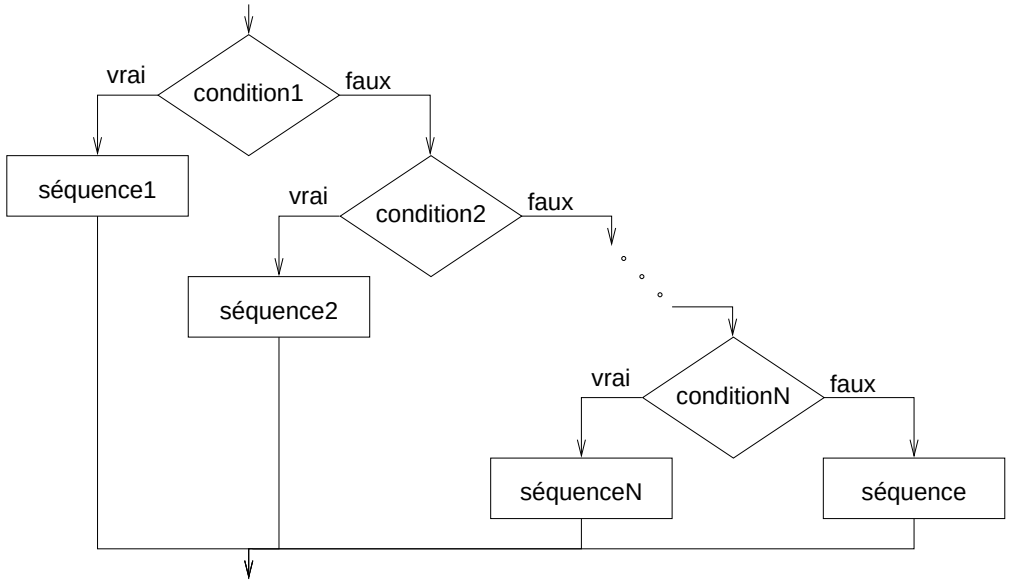
```
Si condition_1 Alors
    séquence_1
SinonSi condition_2 Alors
    séquence_2
...
SinonSi condition_N Alors
    séquence_N
Sinon       { Expliciter la condition ! }
    séquence
FinSi
```

En C,

```
if (condition_1) {       /* séquence_1 */
    instruction_{1,1};
    ...
}
else if (condition_2) {  /* séquence_2 */
    instruction_{2,1};
    ...
}
...
else if (condition_{n,1}) { /* séquence_n */
    instruction;
}
```

Évaluation : Les conditions sont évaluées dans l'ordre d'apparition. Dès qu'une condition est vraie, la séquence associée est exécutée. L'instruction suivante à exécuter sera alors celle qui suit le **`FinSi`**. Si aucune condition n'est vérifiée, alors la séquence associée au **`Sinon`**, si elle existe, est exécutée.

Organigramme :



Exercice 12 : Signe d'un entier
Étant donné un entier lu au clavier, indiquer s'il est nul, positif ou négatif.

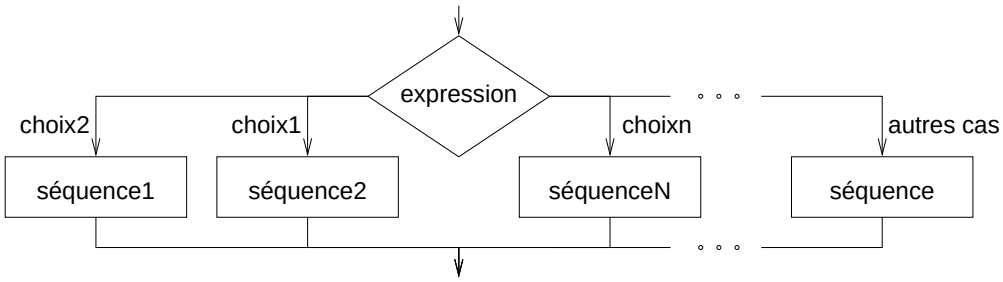
9.2.4 Conditionnelle `Selon`

```
Selon expression Dans
    choix_1 :
        séquence_1
    choix_2 :
        séquence_2
    ...
    choix_N :
        séquence_N
Sinon
    séquence
FinSelon
```

Règles :
- expression est nécessairement une expression de type scalaire.
- choix$_i$ est une liste de choix séparés par des virgules. Chaque choix est soit une constante, soit un intervalle (`10..20`, par exemple).
  L'instruction **`Selon`** peut donc être considérée comme un cas particulier de la conditionnelle
  `Si ... SinonSi ... FinSi`.

Évaluation : L'`expression` est évaluée, puis sa valeur est successivement comparée à chacun des ensembles choix$_i$. Dès qu'il y a correspondance, les comparaisons sont arrêtées et la séquence associée est exécutée. Les différents choix sont donc *exclusifs*. Si aucun choix ne correspondant, alors la séquence associée au **`Sinon`**, si elle existe, est exécutée.

Remarque : La clause **Sinon** est optionnelle... mais il faut être sûr de traiter tous les cas (toutes les valeurs possibles de l'expression).
Organigramme :



En C, Le `Selon` est la structure de contrôle dont la transcription en C est la plus éloignée de la version algorithmique.

```
switch (expression) {
    case expr_cste1:
        instructions1;
    case expr_cste2:
        instructions2;
    ...
    case expr_csten:
        instructionsn;
    default:
        instruction;
}
```

Principe : L'exécution commence par les instructions de la 1re expression constante qui correspond à l'expression du **switch** et continue jusqu'à un **break** ou la fin du **switch**. Ainsi, si on ne met pas de **break**, les instructions du cas suivant seront également exécutées.
Conséquence : Si le même traitement doit être fait pour plusieurs cas, il suffit de lister les différents **case** correspondants consécutivement.
Conseil : Mettre un **break** après chaque groupe d'instructions d'un **case**.

Exercice 13 : Réponse
Écrire un programme qui demande à l'utilisateur de saisir un caractère et qui affiche « affirmatif » si le caractère est un « o » (minuscule ou majuscule), « négatif » si c'est un « n » (minuscule ou majuscule) et « ? ! ? ! ? ! ? » dans les autres cas.

9.3 Instructions de répétitions

9.3.1 Répétition `TantQue`

```
TantQue condition Faire
    séquence
FinTQ
```

En C,

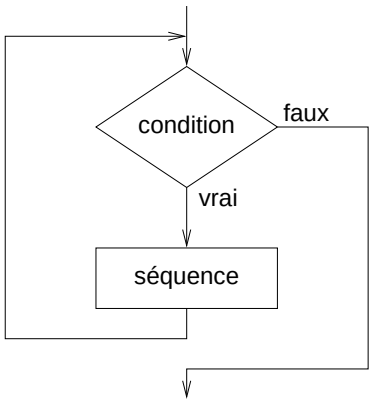
```
while (condition) {
    instruction;
    ...
}
```

Règles :

- La condition doit être une expression booléenne.
- Pour que la boucle se termine, il est nécessaire que la séquence modifie la condition.

Évaluation : La condition est évaluée. Si elle vaut **`FAUX`** alors la boucle se termine et l'exécution se poursuit avec l'instruction qui suit **`FinTQ`**. Si elle vaut **`VRAI`** alors la séquence d'instructions est exécutée et la condition est de nouveau évaluée.

Organigramme :



Exercice 14 : Condition après un `FinTQ`

Que sait-on sur l'état du programme lorsque l'instruction suivante à exécuter est celle qui suit le **`FinTQ`** ?

Remarque : On fera apparaître explicitement cette condition sous forme de commentaire après le **`FinTQ`**.

Remarque : Comme le test de la condition est fait en premier, la séquence peut ne pas être exécutée. Il suffit que la condition soit fausse dès le début.

Problème de la terminaison. Nous avons vu qu'une propriété importante d'un algorithme/programme est qu'il doit se terminer. Jusqu'à présent la terminaison était assurée car avec seulement la séquence et les conditionnelles, l'exécution du programme se fait toujours vers l'avant. On doit donc nécessairement atteindre la fin.

Avec les répétitions, on peut revenir en arrière dans le programme. Il est alors important de s'assurer que l'on finira toujours par sortir des répétitions d'un programme. Sinon, le programme risque de s'exécuter indéfiniment. On parle alors de boucle sans fin.

Pour garantir qu'une boucle (une répétition) se termine, on peut mettre en évidence un variant. C'est une expression entière qui doit toujours être positive et décroître strictement à chaque passage dans la boucle. Si on arrive à exhiber ce variant et prouver les deux propriétés, alors on est assuré que la répétition se termine.

Un programme qui se termine n'est pas forcément un programme valide. Pour vérifier la validité d'une boucle, on peut mettre en évidence un invariant. C'est une expression booléenne qui

est vraie avant et après chaque passage dans la boucle. Elle aide à prouver la validité d'une boucle et d'un programme. L'invariant est cependant généralement difficile à trouver et la preuve difficile à faire. Cependant, dans les exercices nous essaierons de mettre en évidence des invariants, même s'ils ne sont qu'intuitifs et incomplets.

Exercice 15 : Somme des premiers entiers (TantQue)
Calculer la somme des n premiers entiers.

Exercice 16 : Saisie contrôlée d'un numéro de mois
On souhaite réaliser la saisie du numéro d'un mois (compris entre 1 et 12) avec vérification. Le principe est que si la saisie est incorrecte, le programme affiche un message expliquant l'erreur de saisie et demande à l'utilisateur de resaisir la donnée.

On utilisera un **`TantQue`** pour réaliser la saisie contrôlée.

Généraliser l'algorithme au cas d'une saisie quelconque.

Théorème : Tout algorithme (calculable) peut être exprimé à l'aide de l'affectation et des trois structures **`Si Alors FinSi`**, **`TantQue`** et enchaînement séquentiel.

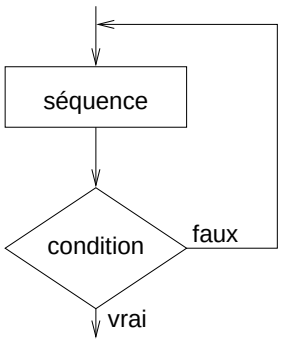
... Mais ce n'est pas forcément pratique, aussi des structures supplémentaires ont été introduites.

9.3.2 Répétition Répéter ... JusquÀ

```
Répéter
    séquence
JusquÀ condition
```

Remarques :
- la condition n'est évaluée qu'après l'exécution de la séquence ;
- la séquence est exécutée au moins une fois ;
- la condition doit être modifiée par la séquence ;

Organigramme :



En C,

```
do {
    instruction;
    ...
} while (condition_continuation);
```

En C, il ne s'agit pas d'un **Répéter ... JusquÀ** mais d'un **Répéter ... TantQue** !

Exercice 17 : Plusieurs sommes des n premiers entiers
Écrire un programme qui affiche la somme des n premiers entiers naturels, n étant un entier saisi au clavier. Le programme devra proposer la possibilité à l'utilisateur de recommencer le calcul pour un autre entier.

Exercice 18 : Saisie contrôlée d'un numéro de mois
On souhaite réaliser la saisie du numéro d'un mois (compris entre 1 et 12) avec vérification. Le principe est que si la saisie est incorrecte, le programme affiche un message expliquant l'erreur de saisie et demande à l'utilisateur de resaisir la donnée.

On utilisera un répéter pour réaliser la saisie contrôlée.

Généraliser l'algorithme au cas d'une saisie quelconque.

Exercice 19 : TantQue et Répéter
Écrire la répétition **Répéter** à partir du **TantQue** et reciproquement.

9.3.3 Répétition Pour

```
Pour var ← val_min [ Décrémenter ] JusquÀ var = val_max Faire
    sequence
FinPour

-- Une variante
Pour Chaque var Dans val_min..val_max [ Renversé ] Faire
    sequence
FinPour
```

Règle :
- La variable var est une variable d'un type scalaire. Elle est dite *variable de contrôle*.
- Les expressions val_min et val_max sont d'un type compatible avec celui de var.
- La séquence d'instructions ne doit pas modifier la valeur de la variable var.

Évaluation : Les expressions val_min et val_max sont évaluées. La variable var prend alors successivement chacune des valeurs de l'intervalle [val_min..val_max] dans l'ordre indiqué et pour chaque valeur, la séquence est exécutée.

Remarques :
- Cette structure est utilisée lorsqu'on connaît à l'avance le nombre d'itérations à faire.
- Les expressions val_min et val_max ne sont évaluées qu'une seule fois.
- La séquence peut ne pas être exécutée (intervalle vide : val_min > val_max).
- La séquence termine nécessairement (le variant est val_max - var + 1).

Attention : Il est interdit de modifier la valeur de la variable de contrôle var dans la boucle.

Remarque : Il n'y a pas d'équivalent du **Pour** dans les organigrammes. On peut utiliser la traduction du **Pour** sous forme de **TantQue** ou de **Répéter**.

En C,

```
for (initialisation; condition; incrémentation) {
```

```
    instruction;
    ...
}
```

Le **for** de C est plus général (et plus dangereux) que le **Pour** algorithmique. En effet, la condition (de continuation) est évaluée à chaque étape et l'incrémentation est une instruction quelconque. En particulier, on perd la garantie de terminaison du **Pour** algorithmique.

Le **for** de C n'est en fait qu'un **while** qui regroupe sur une même ligne (dans les parenthèses), l'initialisation, la condition de continuation et le passage au suivant.

```
{   /* réécriture du for à l'aide du while */
    initialisation;     /* initialisation */
    while (condition) { /* condition de continuation */
        instruction;    /* traitement */
        ...
        incrémentation; /* incrémentation */
    }
}
```

Conseil : Conserver la sémantique du **Pour** algorithmique : on sait à l'avance combien de fois la boucle doit être exécutée.

Exercice 20 : Somme des premiers entiers
Calculer la somme des n premiers entiers.

Exercice 21 : Alphabet
Écrire les lettres de l'alphabet.

9.3.4 Quelle répétition choisir ?

La première question à se poser est : « Est-ce que je connais a priori le nombre d'itérations à effectuer ? ». Dans l'affirmative, on choisit la boucle **Pour**.

Dans la négative, on peut généralement employer soit un **TantQue**, soit un **Répéter**. En général, utiliser un **Répéter** implique de dupliquer une condition et utiliser un **TantQue** implique de dupliquer une instruction.

Dans le cas, où on choisit d'utiliser un **Répéter**, il faut faire attention que les instructions qu'il contrôle seront exécutées au moins une fois. S'il existe des cas où la séquence d'instructions ne doit pas être exécutée, alors il faut utiliser un **TantQue** (ou protéger le **Répéter** par un **Si**).

Une heuristique pour choisir entre **TantQue** et **Répéter** est de se demander combien de fois on fait l'itération. Si c'est au moins une fois alors on peut utiliser un **Répéter** sinon on préfèrera un **TantQue**.

Remarque : Pour aider au choix, il est parfois judicieux de se poser les questions suivantes :

- Qu'est ce qui est répété (quelle est la séquence) ?
- Quand est-ce qu'on arrête (ou continue) ?

```
R1 : Comment { Quelle répétition choisir ? }
        Si nombre d'itérations connu Alors
            Résultat ← Pour
```

```
Sinon
    Si itération exécutée au moins une fois Alors
        Résultat ← Répéter
    Sinon
        Résultat ← TantQue
    FinSi
FinSi
```