Algorithmique & Langage C

Professeur : M. EL HAJJI

TABLE DES MATIERES

Algorithmique

Avez-vous déjà ouvert un livre de recettes de cuisine ? Avez vous déjà déchiffré un mode d'emploi traduit directement du coréen pour faire fonctionner un magnétoscope ou un répondeur téléphonique réticent ? Si oui, sans le savoir, vous avez déjà exécuté des **algorithmes**.

Plus fort : avez-vous déjà indiqué un chemin à un touriste égaré ? Avez vous fait chercher un objet à quelqu'un par téléphone ? Si oui, vous avez déjà fabriqué – et fait exécuter – des algorithmes.

Comme quoi, l'algorithmique n'est pas un savoir ésotérique réservé à quelques rares initiés touchés par la grâce divine, mais une aptitude partagée par la totalité de l'humanité. Donc, pas d'excuses…

L'art de programmer, c'est l'art - au sens d'artisan - de faire résoudre des problèmes par des machines (ordinateur).

1 NOTIONS ET VOVABULAIRES

1.1 ALGORITHME

Un algorithme, c'est une suite d'instructions(أوامر) , qui une fois exécutée correctement, conduit à un résultat donné.

Exemple :

Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller, si l'algorithme est faux, le résultat est, disons, aléatoire.

Pour fonctionner, **un algorithme doit donc contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter**.

En informatique, les choses auxquelles ont doit donner des instructions sont les ordinateurs, et ceux-ci ont le bon goût d'être tous strictement aussi idiots les uns que les autres.

Remarque :

La maîtrise de l'algorithmique requiert deux qualités, très complémentaires d'ailleurs :

- il faut avoir une certaine intuition, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, si l'on y tient, qu'intervient la forme « d'intelligence » requise pour l'algorithmique. Alors, c'est certain, il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, et j'insiste sur ce point, les réflexes, cela s'acquiert. Et ce qu'on appelle l'intuition n'est finalement que de l'expérience tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».
- il faut être **méthodique** et **rigoureux**. En effet, chaque fois qu'on écrit une série d'instructions qu'on croit justes, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter, armé d'un papier et d'un crayon, afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas la moindre once d'intelligence. Mais elle reste néanmoins indispensable, si l'on ne veut pas écrire à l'aveuglette.

Pourquoi apprendre l'algorithmique pour apprendre à programmer ? En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ?

Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique. Cette dimension est présente quelle que soit le langage de programmation ; mais lorsqu'on programme dans un langage (en C, en Visual Basic, etc.) on doit en plus se colleter les problèmes de

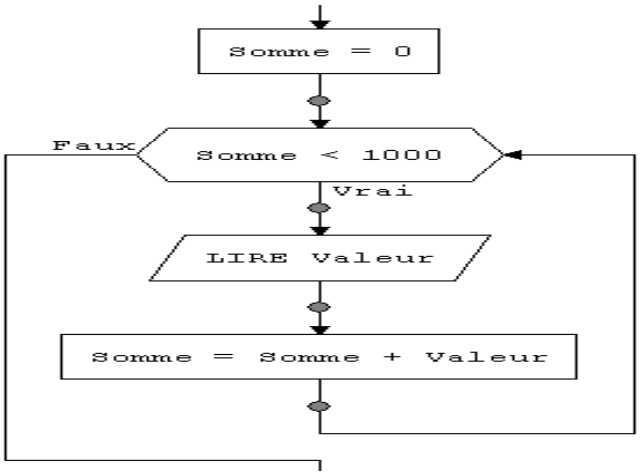
syntaxe, ou de types d'instructions, propres à ce langage. Apprendre l'algorithmique de manière séparée, c'est donc sérier les difficultés pour mieux les vaincre.

1.2 AVEC QUELLES CONVENTIONS ECRIT-ON UN ALGORITHME ?

Historiquement, plusieurs types de notations ont représenté des algorithmes.

Il y a eu notamment une représentation graphique, avec des carrés, des losanges, etc. qu'on appelait des organigrammes.

Exemple :



Cette représentation est quasiment abandonnée, parce que dès que l'algorithme commence à grossir un peu, ce n'est plus pratique du tout du tout.

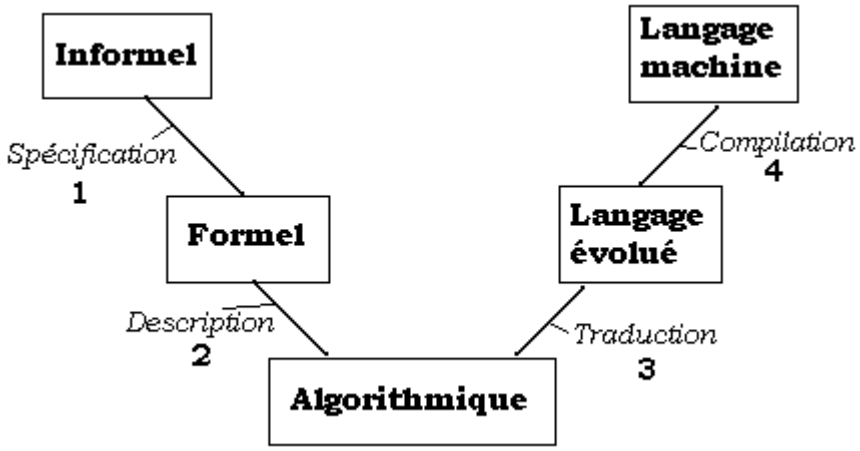
C'est pourquoi on utilise généralement une série de conventions appelée « pseudo-code », qui ressemble à un langage de programmation authentique dont on aurait évacué la plupart des problèmes de syntaxe. Ce pseudo-code est susceptible de varier légèrement d'un livre (ou d'un enseignant) à un autre.

Exemple :

```
Algorithme
   Variable  moy : entier
Début
        Ecrire « saisir le nom de l'étudiant : »
        Lire (nom)
        Ecrire « saisir la moyenne de l'étudiant : »
        Lire (moy)
        Si moy>12 Alors Ecrire « Admis »
        Sinon Si moy>10 Alors Ecrire « Redouble »
                Sinon Ecrire « Exclus »
              Finsi
        Finsi
Fin
```

1.3 DEMARCHE

Le schéma de la programmation d'un problème se réduit à 4 phases :



La phase 1 de spécification utilisera les types abstraits de données, la phase 2 (correspondant aux phases 3 et 4 du cycle de vie) utilisera la méthode de programmation algorithmique, la phase 3 (correspondant à la phases 5 du cycle de vie) utilisera un traducteur manuel pascal, la phase 4 (correspondant à la phases 6 du cycle de vie) correspondra au passage sur la machine avec vérification et jeux de tests.

Nous utiliserons un " langage algorithmique " pour la description d'un algorithme résolvant un problème. Il s'agit d'un outil textuel permettant de passer de la conception humaine à la conception machine d'une manière souple pour le programmeur.

Nous pouvons résumer dans le tableau ci-dessous les étapes de travail et les outils conceptuels à utiliser lors d'une telle démarche.

TAPES PRATIQUES	Matériel et moyens techniques à disposition
Analyse	Papier, Crayon, Intelligence, Habitude.
Mise en forme de l'algorithme	C'est l'aboutissement de l'analyse, esprit logique et rationnel.
Description	Utilisation pratique des outils d'une méthode de programmation, ici les technique de l'algorithmique.
Traduction	Transfert des écritures algorithmiques en langage de programmation (C, visual basic, pascal…).
Tests et mise au point	Mise au point du programme sur des valeurs tests ou à partir de programmes spécialisés.
Exécution	Phase finale : le programme s'exécute sans erreur.

Langage de programmation

L'apprentissage d'un langage de programmation ne sert qu'aux phases 3 et 4 (traduction et exécution) et ne doit pas être confondu avec l'utilisation d'un langage algorithmique qui prépare le travail et n'est utilisé que comme plan de travail pour la phase de traduction. En utilisant la construction d'une maison comme analogie, il suffit de bien comprendre qu'avant de construire la maison, le chef de chantier a besoin du plan d'architecte de cette maison pour passer à la phase d'assemblage des matériaux ; il en est de même en programmation.

2 LES VARIABLES

Une variable est un nom qui sert à réserver un emplacement de la mémoire destinée à recevoir une valeur (donnée).

Une variable peut être vue comme une boîte dans laquelle sont rangées des informations qui peuvent être récupérées en tout temps. À la différence de la boîte qui redevient vide lorsqu'on en retire son contenu, la variable stocke une « copie » des données que l'algorithme y range (Affectation).

Une variable conserve ses données jusqu'à ce que d'autres données y soient stockées, remplaçant les données antérieures, ou jusqu'à ce que la variable soit détruite par l'algorithme.

Une variable doit avoir un **nom** et un **type** qui détermine la manière de traiter cette variable par la l'ordinateur.

2.1 DECLARATION DES VARIABLES

La première chose à faire avant de pouvoir utiliser une variable est de créer la **boîte** et de lui coller une **étiquette**. Ceci se fait tout au **début de l'algorithme**, avant même les instructions proprement dites. C'est ce qu'on appelle **la déclaration des variables**.

Le nom de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue :

- Un nom de variable doit débuter par une lettre (A à Z, a à z) ou le caractère de soulignement (_).
- Un nom de variable peut être constitué de lettres minuscules (a à z), de lettres majuscules (A à Z), de chiffres (0 à 9) et du caractère de soulignement (_).
- Un nom de variable ne doit pas correspondre à un mot réservé, tels que ÉCRIRE, FIN, SI et ce sans égard aux accents (par exemple ECRIRE est aussi un mot réservé).

Pour déclarer une variable il faut préciser un nom et un type nous écrivons pour cela:

Variable nom_variable : type

Où **nom_variable** est le nom de la variable et type est un nom de type.

2.2 LES TYPES

Donner un type à une variable consiste à définir l'ensemble des valeurs que peut prendre cette variable ainsi que les opérations qui peuvent lui être appliquées.

On distingue plusieurs types de données.

2.2.1 Types numériques:

Commençons par le cas très fréquent, celui d'une variable destinée à recevoir des nombres.

Type Numérique	intervalle
Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647

Réel simple	$-3{,}40\text{x}10^{38}$ à $3{,}40\text{x}10^{38}$
Réel double	$1{,}79\text{x}10^{308}$ à $1{,}79\text{x}10^{308}$

En algorithmique, on ne se tracassera pas trop avec les sous-types de variables numériques (sachant qu'on aura toujours assez de soucis comme ça, allez). On se contentera donc de préciser qu'il s'agit d'un nombre, en gardant en tête que dans un vrai langage, il faudra être plus précis.

En pseudo-code, une déclaration de variables aura ainsi cette tête :

Variable** g **: Numérique
ou encore
Variables** PrixHT, TauxTVA, PrixTTC **: Numérique

2.2.2 Type caractère:

Un caractère peut être une lettre (a à z ou A à Z) ou bien un symbole (+, &, $, # ...) ou encore un chiffre (0 à 9).

On désigne le type caractère par le mot clé caractère.

2.2.3 Type chaîne de caractères:

Une chaîne de caractères est une suite finie de caractère (a à z ou A à Z) ou bien des symbole !?@_ ou core des chiffres de 0 jusqu'à 9.

Exemple:

Variable i: entier

Variable Nombre, note, surface : réel

C : caractère

Nom : chaine_caracteres

Cela signifie que i est de type entier. Nombre, note et surface sont de type réel. C est de type caractère alors que Nom est de type chaîne de caractères

2.2.4 Les constantes:

Une constante est une donnée qui ne varie pas tout le long d'exécution d'un algorithme.

Exemple:

PI= 3.14, v=3, g= 9.8, TAV =20

Déclaration d'une constante:

Pour déclarer une constante on doit préciser un nom et une valeur pour cette constante, on écrit ainsi:

Constante nom_constante=valeur

Exemple:

Constante p=3.14, g=9.8

Cela signifie que g et p sont des constantes de valeurs 9.8 et 3.14 respectivement.

2.2.5 Type booléen

Le dernier type de variables est le type booléen: on y stocke uniquement les valeurs logiques VRAI et FAUX.

On peut représenter ces notions abstraites de VRAI et de FAUX par tout ce qu'on veut : de l'anglais (TRUE et FALSE) ou des nombres (0 et 1). Peu importe. Ce qui compte, c'est de comprendre que le type booléen est très économique en termes de place mémoire occupée, puisque pour stocker une telle information binaire, un seul bit suffit.

Remarques

Certains langages autorisent d'autres types numériques, notamment :

- le type monétaire (avec strictement deux chiffres après la virgule)
- le type date (jour/mois/année).

3 STRUCTURE D'UN ALGORITHME:

Un algorithme a la structure suivante :

```
Algorithme nom_algorithme
    Variable   /*déclaration des variables qui seront utilisées par l'algorithme */
    Constante /* déclaration des constantes qui seront utilisées par l'algorithme
Début
    Action1 (instruction أمر)
    Action2
    ……
    Action_n
Fin
```

Où **nom_algorithme** désigne le nom de l'algorithme et Le mot clé Variable signifie que les noms qui le suivent sont des variables.

Le mot clé **Constante** signifie que les noms qui le suivent sont des constantes.

Le mot clé **Début** désigne le début de l'algorithme.

Action1,…. Action_n désignent les actions à effectuer lors de l'exécution de l'algorithme.

Le mot clé Fin désigne où se termine l'algorithme.

Exemples:

Soit à écrire l'algorithme qui permet de calculer la surface d'un cercle.

Algorithme surface_cercle

Variable R, S : reel

Constante P=3.14

Début

Fin

4 LES INSTRUCTIONS DE BASE

Les actions (opérations) élémentaires qui composent un algorithme sont appelées **instructions.**

4.1 L'INSTRUCTION DE LECTURE ET D'ECRITURE

4.1.1 L'instruction de lecture (Entrée)

C'est l'action qui permet à l'utilisateur de fournir à l'algorithme les valeurs de données variables. Elle permet d'attribuer une valeur à un objet en allant lire sur un périphérique d'entrée et elle range cette valeur dans l'objet.

Syntaxe :

Lire (V) Où V est une variable

Lors de l'exécution de l'action lire, l'algorithme attend que l'utilisateur fournisse, à partir de clavier, la valeur de la variable V.

4.1.2 L'instruction d'écriture (sortie)

C'est l'action qui permet à l'algorithme d'afficher pour l'utilisateur des messages ou des résultats à l'écran.

Syntaxe :

Ecrire (Val)

Où Val ça peut être une variable, une constante, ou une chaîne de caractère.

Exemple

Algorithme surface_cercle

Variable R : reel

Constante P=3.14

Debut

Ecrire(' Donner le rayon du cercle')

Lire (R)

Ecrire('la surface du cercle est :', R*R*P)

Fin

4.2 L'INSTRUCTION D'AFFECTATION

Une **instruction d'affectation** consiste à mettre dans une variable la valeur d'une expression. Autrement dit, mettre la valeur dans la zone mémoire qui représente la variable.

Syntaxe :

nom_variable ← Expression ;

Exemple :

A ⟵ 100;

B ⟵ 5*A+1 ;

La première instruction demande de placer la valeur 100 dans la variable A.

La seconde demande de calculer l'expression 5*A+1 et de placer le résultat dans la variable B.

Exemple : l'algorithme qui calcule la surface d'un cercle

```
Algorithme surface_cercle
        Variable R, S : reel
        Constante P=3.14
Debut
        Ecrire(' Donner le rayon du cercle')
        Lire (R)
        S ⟵ R*R*P
        Ecrire ('la surface du cercle est :', R*R*P)
Fin
```

Exercice : écrire un algorithme qui permet de la somme et le produit de deux données numérique.

Solution :

```
Algorithme Somme_Produit ;
        Var A , B : REEL ;
        S, P : REEL ;                    } Déclaration des données
Début
        Ecrire (' Donner deux nombres :') ;
        Lire (A, B)
        S ⟵ A+B ;                        } Bloc d'instructions
        S ⟵ A+B ;
        Ecrire ('la somme des deux nombres est :', S) ;
        Ecrire ('Le produit des deux nombres est :', P) ;
Fin
```

4.3 EXPRESSION ET OPERATEURS

- Une expression est un ensemble de valeurs, reliées par des opérateurs, et équivalent à une seule valeur
- Un opérateur est un signe qui relie deux valeurs, pour produire un résultat. Afin de permettre les calculs mathématiques.

Les opérateurs possibles dépendent du type des valeurs qui sont en jeu.

Opérateurs numériques :

Ce sont les quatre opérations arithmétiques tout ce qu'il y a de classique.
+ : addition
- : soustraction
* : multiplication
/ : division
Mentionnons également le ^ qui signifie « puissance ». 45 au carré s'écrira donc 45 ^ 2.

Opérateur alphanumérique : &

Cet opérateur permet de concaténer, autrement dit d'agglomérer, deux chaînes de caractères. Par exemple :

Variables A, B, C en Caractère
Début
A ← "Gloubi"
B ← "Boulga"
C ← A & B
Fin

La valeur de C à la fin de l'algorithme est "GloubiBoulga"

Opérateurs logiques (ou booléens) :

Il s'agit du ET, du OU, du NON et du mystérieux (mais rarissime XOR).(cous de notions mathèmatiques)

Exercices

Exercice 1.1

Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?

Variables A, B **en Entier**
Début
A ← 1
B ← A + 3
A ← 3
Fin

Exercice 1.2

Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

Variables A, B, C **en Entier**
Début
A ← 5
B ← 3
C ← A + B
A ← 2
C ← B – A
Fin

Exercice 1.3

Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?

Variables A, B **en Entier**
Début
A ← 5
B ← A + 4
A ← A + 1
B ← A – 4
Fin

Exercice 1.4

Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

Variables A, B, C **en Entier**
Début
A ← 3
B ← 10
C ← A + B
B ← A + B

A ← C
Fin

Exercice 1.5

Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?
Variables A, B **en Entier**
Début
A ← 5
B ← 2
A ← B
B ← A
Fin
Moralité : les deux dernières instructions permettent-elles d'échanger les deux valeurs de B et A ? Si l'on inverse les deux dernières instructions, cela change-t-il quelque chose ?

Exercice 1.6

Plus difficile, mais c'est un classique absolu, qu'il faut absolument maîtriser : écrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.

Exercice 1.7

Une variante du précédent : on dispose de trois variables A, B et C. Ecrivez un algorithme transférant à B la valeur de A, à C la valeur de B et à A la valeur de C (toujours quels que soient les contenus préalables de ces variables).

Exercice 1.8

Que produit l'algorithme suivant ?
Variables A, B, C **en Caractères**
Début
A ← "423"
B ← "12"
C ← A + B
Fin

Exercice 1.9

Que produit l'algorithme suivant ?
Variables A, B, C **en Caractères**
Début
A ← "423"
B ← "12"
C ← A & B
Fin

Exercice 2.1

Quel résultat produit le programme suivant ?
Variables val, double numériques
Début
Val ← 231
Double ← Val * 2
Ecrire Val
Ecrire Double
Fin

Exercice 2.2

Ecrire un programme qui demande un nombre à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.

Exercice 2.3

Ecrire un programme qui lit le prix HT d'un article, le nombre d'articles et le taux de TVA, et qui fournit le prix total TTC correspondant. Faire en sorte que des libellés apparaissent clairement.

Exercice 2.4

Ecrire un algorithme utilisant des variables de type chaîne de caractères, et affichant quatre variantes possibles de la célèbre « belle marquise, vos beaux yeux me font mourir d'amour ». On ne se soucie pas de la ponctuation, ni des majuscules.

5 INSTRUCTION DE CONTROLE

On appelle *instruction de contrôle* toute instruction qui permet de contrôler la succession des actions d'un programme. Parmi les instructions de contrôle, on distingue les instructions de test (*branchement)* et les instructions de répétition (*boucle).*

5.1 STRUCTURE D'UN TEST

Reprenons le cas de notre « programmation algorithmique du touriste égaré ». Normalement, l'algorithme ressemblera à quelque chose comme : « Allez tout droit jusqu'au prochain carrefour, puis prenez à droite et ensuite la deuxième à gauche, et vous y êtes ».

Mais en cas de doute légitime de votre part, cela pourrait devenir : « Allez tout droit jusqu'au prochain carrefour et là regardez à droite. Si la rue est autorisée à la circulation, alors prenez la et ensuite c'est la deuxième à gauche. Mais si en revanche elle est en sens interdit, alors continuez jusqu'à la prochaine à droite, prenez celle-là, et ensuite la première à droite ».

Ce deuxième algorithme a ceci de supérieur au premier qu'il prévoit, en fonction d'une situation pouvant se présenter de deux façons différentes, deux façons différentes d'agir. Cela suppose que l'interlocuteur (le touriste) sache analyser la condition que nous avons fixée à son comportement (« la rue est-elle en sens interdit ? ») pour effectuer la série d'actions correspondante.

Eh bien, croyez le ou non, mais les ordinateurs possèdent cette aptitude, sans laquelle d'ailleurs nous aurions bien du mal à les programmer. Nous allons donc pouvoir parler à notre ordinateur comme à notre touriste, et lui donner des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre.

Cette structure logique répond au doux nom de **test**. Toutefois, ceux qui tiennent absolument à briller en société parleront également de structure alternative.

Il n'y a que deux formes possibles pour un test ; la première est la plus simple, la seconde la plus complexe.

```
Si   booléen      Alors
          Instructions
Finsi
```

```
Si    booléen     Alors
        Instructions 1
Sinon
         Instructions 2
Finsi
```

Ceci appelle quelques explications.

Un booléen est une expression dont la valeur est VRAI ou FAUX. Cela peut donc être (il n'y a que deux possibilités) :

- une variable (ou une expression) de type booléen
- une condition

Exemple

Un algorithmique pour un touriste égaré

```
Allez tout droit jusqu'au prochain carrefour
Si la rue à droite est autorisée à la circulation Alors
  Tournez à droite
  Avancez
  Prenez la deuxième à gauche
Sinon
  Continuez jusqu'à la prochaine rue à droite
  Prenez cette rue
  Prenez la première à droite
Finsi
```

5.1.1 Condition

une condition est composée de trois éléments :

- une valeur
- un opérateur de comparaison
- une autre valeur

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères…). Mais si l'on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type.

Les opérateurs de comparaison sont :

- égal à…
- différent de…
- strictement plus petit que…
- strictement plus grand que…
- plus petit ou égal à…
- plus grand ou égal à…

5.2 Faut-il mettre un ET ? Faut-il mettre un OU ?

Une remarque pour commencer : dans le cas de conditions composées, les parenthèses jouent un rôle fondamental.

```
Variables A, B, C, D, E en Booléen
Variable X en Entier
Début
Lire X
A ← X > 12
B ← X > 2
C ← X < 6
D ← (A ET B) OU C
E ← A ET (B OU C)
Ecrire D, E
Fin
```

Si X = 3, alors on remarque que D sera VRAI alors que E sera FAUX.
S'il n'y a dans une condition que des ET, ou que des OU, en revanche, les parenthèses ne changent strictement rien.
Quand faut-il ouvrir la fenêtre de la salle ? Uniquement si les conditions l'imposent, à savoir :

Si** il fait trop chaud **ET** il ne pleut pas **Alors
Ouvrir la fenêtre
Sinon
Fermer la fenêtre
Finsi

Cette petite règle pourrait tout aussi bien être formulée comme suit :

Si** il ne fait pas trop chaud OU il pleut **Alors
Fermer la fenêtre
Sinon
Ouvrir la fenêtre
Finsi

Ces deux formulations sont strictement équivalentes. Ce qui nous amène à la conclusion suivante :
Toute structure de test requérant une condition composée faisant intervenir l'opérateur ET peut être exprimée de manière équivalente avec un opérateur OU, et réciproquement.

Exemple : écrire un algorithme qui affiche le maximum de deux nombre A et B

```
Début
Déclaration   Entier : A ,B , Max
Ecrire « saisir une valeur pour A et B : »
Lire( A , B)
    Si A>B Alors
        Max ← A
        Ecrire (max)
    Sinon

        Max ← B
        Ecrire (max)
    Finsi
Fin
```

Exercice 1 : connaissant deux valeurs A et B, on veut écrire un algorithme qui affiche la plus grande des deux.

```
Algorithme : Déclaration   Entier : A ,B , Max
Début
        Ecrire « saisir une valeur pour A et B : »
        Lire( A , B)
        Si A>B Alors Max ← A
        Sinon Max ← B
        Finsi
        Ecrire « le plus grand des deux est : », Max
        Fin
```

Exercice 2 : afficher le résultat de fin d'année pour un étudiant connaissant sa moyenne générale.

```
Algorithme : Déclaration   Entier : moy
Début
        Ecrire « saisir le nom de l'étudiant : »
        Lire( nom)
        Ecrire « saisir la moyenne de l'étudiant : »
        Lire( moy)
        Si moy>12 Alors Ecrire « Admis »
        Sinon Si moy>10 Alors Ecrire « Redouble »
                Sinon Ecrire « Exclus »
              Finsi
        Finsi
Fin
```

Exercice 3 : Ecrire un algorithme qui calcule le salaire brute d'un ouvrier sachant que les heures supplémentaire (plus de 172 heures) sont payées 50% en plus.

```
Algorithme : Déclaration Réel    : TH ,NH, SB
Début
        Ecrire « entrez le nombre d'heure »
        Lire NH
        Ecrire « entrez le tarif horaire »
        Lire TH
        SB ← TH*NH
        Si NH <= 172 Alors SB ← SB
        Sinon SB ← SB +(NH-172)*TH/2
        finsi
        Ecrire « le salaire brute est : » SB
        Fin
```

Exercice 4 :

Calculer le montant de la facture d'un client ayant commandé une quantité d'un produit avec un prix unitaire Hors Taxe.
Le taux de TVA est 20%
Les frais de transport sont 0.8 dh de Km,
le client est dispensé du frais de transport si le montant est supérieur à 4500 dhs.

```
Algorithme : Déclaration Réel : Q , PU , K,M
Début
        Ecrire « donner la quantité : »
        Lire Q
        Ecrire « donner le prix unitaire : »
        Lire PU
        Ecrire « le nombre de Km :»
        Lire K
        M ← (Q*PU)+(Q*PU*0.2)
        Si M>4500 alors écrire « M=M »
        Sinon écrire M ← M+(0.2*K)
        Fin si
        Ecrire « le montant de la facture est : »,M
Fin
```

Exercice 5 :

Une bibliothèque fait une réduction sur l'achat des livres :
- 25% pour les étudiants
- 15% pour les enseignants

Ecrire un algorithme qui, ayant un montant d'achat et un code client, calcule et affiche le prix à payer.

```
Algorithme : Déclaration Char : CC
                         Réel : MA,P,PP,
Début
  Ecrire « entrer le code client : »
  Lire CC
  Ecrire « le montant d'achat : »
  Lire MA
  Si CC = « E » alors PP ← MA-(MA*0.25)
  Sinon Si Cc = « P » alors PP ← MA-(MA*0.15)
              Sinon PP ← MA
      Fin si
  Ecrire « le prix à payer est : »,PP
  Fin si
Fin
```

Exercice 6 :

Ecrire un algorithme qui calcul et affiche le maximum de trois nombre A, B et C.

```
Algorithme : Déclaration  Réel : MA,P,PP,
Début
        Ecrire « donner les trois chiffres : »

        Lire (A, B,C)

        Si A>B alors
           Si A>C écrire « le plus grand est : »,A
           Sinon écrire « le plus grand est : »,C
           Fin si
        Sinon A<B alors Si B<C alors
           écrire « le plus grand est: » ,C
          Sinon écrire « le plus grand est : »,B
                        Fin si
```

Exercice 7 :

Ecrire un algorithme qui permet de résoudre l'équation du premier degré x+B=0

```
Algorithme : Déclaration  Réel : A,B,C,X
Début                                                    A
        Ecrire « la valeur de A »
        Lire A
        Ecrire « la valeur de B »
        Lire B
        Si A=0 alors
          Si B=0
              écrire (" l'ensemble de solution est : R ")
          Sinon
              écrire (" l'ensemble vide : ")
          Fin si
        Sinon
            x ← -B/A
        Ecrire (" la valeur de x est : ",x)
        Fin si
Fin
```

Exercices

Exercice 4.1

Formulez un algorithme équivalent à l'algorithme suivant :

Si Tutu > Toto + 4 OU Tata = "OK" **Alors**
  Tutu ← Tutu + 1
Sinon
  Tutu ← Tutu – 1
Finsi

Exercice 4.2

Cet algorithme est destiné à prédire l'avenir, et il doit être infaillible !

Il lira au clavier l'heure et les minutes, et il affichera l'heure qu'il sera une minute plus tard. Par exemple, si l'utilisateur tape 21 puis 32, l'algorithme doit répondre :

"Dans une minute, il sera 21 heure(s) 33".

NB : on suppose que l'utilisateur entre une heure valide. Pas besoin donc de la vérifier.

Exercice 4.3

De même que le précédent, cet algorithme doit demander une heure et en afficher une autre. Mais cette fois, il doit gérer également les secondes, et afficher l'heure qu'il sera une seconde plus tard.

Par exemple, si l'utilisateur tape 21, puis 32, puis 8, l'algorithme doit répondre : "Dans une seconde, il sera 21 heure(s), 32 minute(s) et 9 seconde(s)".
NB : là encore, on suppose que l'utilisateur entre une date valide.

Exercice 4.4
Un magasin de reprographie facture 0,10 E les dix premières photocopies, 0,09 E les vingt suivantes et 0,08 E au-delà. Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.

Exercice 4.5
Les habitants de Zorglub paient l'impôt selon les règles suivantes :

- les hommes de plus de 20 ans paient l'impôt
- les femmes paient l'impôt si elles ont entre 18 et 35 ans
- les autres ne paient pas d'impôt

Le programme demandera donc l'âge et le sexe du Zorglubien, et se prononcera donc ensuite sur le fait que l'habitant est imposable.

Exercice 4.6
Les élections législatives, en Guignolerie Septentrionale, obéissent à la règle suivante :

- lorsque l'un des candidats obtient plus de 50% des suffrages, il est élu dès le premier tour.
- en cas de deuxième tour, peuvent participer uniquement les candidats ayant obtenu au moins 12,5% des voix au premier tour.

Vous devez écrire un algorithme qui permette la saisie des scores de quatre candidats au premier tour. Cet algorithme traitera ensuite le candidat numéro 1 (et **uniquement** lui) : il dira s'il est élu, battu, s'il se trouve en ballottage favorable (il participe au second tour en étant arrivé en tête à l'issue du premier tour) ou défavorable (il participe au second tour sans avoir été en tête au premier tour).

Exercice 4.7
Une compagnie d'assurance automobile propose à ses clients quatre familles de tarifs identifiables par une couleur, du moins au plus onéreux : tarifs bleu, vert, orange et rouge. Le tarif dépend de la situation du conducteur :

- un conducteur de moins de 25 ans et titulaire du permis depuis moins de deux ans, se voit attribuer le tarif rouge, si toutefois il n'a jamais été responsable d'accident. Sinon, la compagnie refuse de l'assurer.
- un conducteur de moins de 25 ans et titulaire du permis depuis plus de deux ans, ou de plus de 25 ans mais titulaire du permis depuis moins de deux ans a le droit au tarif orange s'il n'a jamais provoqué d'accident, au tarif rouge pour un accident, sinon il est refusé.
- un conducteur de plus de 25 ans titulaire du permis depuis plus de deux ans bénéficie du tarif vert s'il n'est à l'origine d'aucun accident et du tarif orange pour un accident, du tarif rouge pour deux accidents, et refusé au-delà
- De plus, pour encourager la fidélité des clients acceptés, la compagnie propose un contrat de la couleur immédiatement la plus avantageuse s'il est entré dans la maison depuis plus d'un an.

Ecrire l'algorithme permettant de saisir les données nécessaires (sans contrôle de saisie) et de traiter ce problème. Avant de se lancer à corps perdu dans cet exercice, on pourra réfléchir un peu et s'apercevoir qu'il est plus simple qu'il n'en a l'air (cela s'appelle faire une analyse !)

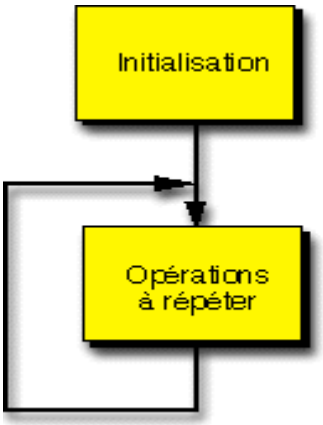
Exercice 4.8

Ecrivez un algorithme qui a près avoir demandé un numéro de jour, de mois et d'année à l'utilisateur, renvoie s'il s'agit ou non d'une date valide.
Cet exercice est certes d'un manque d'originalité affligeant, mais après tout, en algorithmique comme ailleurs, il faut connaître ses classiques ! Et quand on a fait cela une fois dans sa vie, on apprécie pleinement l'existence d'un type numérique « date » dans certains langages…).
Il n'est sans doute pas inutile de rappeler rapidement que le mois de février compte 28 jours, sauf si l'année est bissextile, auquel cas il en compte 29. L'année est bissextile si elle est divisible par quatre. Toutefois, les années divisibles par 100 ne sont pas bissextiles, mais les années divisibles par 400 le sont. Ouf !
Un dernier petit détail : vous ne savez pas, pour l'instant, exprimer correctement en pseudo-code l'idée qu'un nombre A est divisible par un nombre B. Aussi, vous vous contenterez d'écrire en bons télégraphistes que A divisible par B se dit « A dp B ».

5.3 LES STRUCTURES REPETITIVES

Des opérations comme le calcul itératif, la sommation, le décompte d'entités, la recherche d'entités implique la notion de répétition.

À partir du moment où il faut répéter, il faut choisir un point de départ pour démarrer le processus de répétition. Cette étape s'appelle l'initialisation et comme nous le verrons dans ce qui suit, c'est une étape essentielle. Le cas le plus simple de répétition peut être représenté comme suit :



Prenons un programme utilisé par surveillant général qui calcule la moyenne des notes. L'exécution de ce programme fournit la moyenne des notes uniquement pour un seul élève. S'il l'on veut les moyennes de 200 élèves, il faut ré-exécuter ce programme 200 fois. Afin d'éviter cette tâche fastidieux d'avoir ré-exécuter le programme 200 fois, on peut faire recourt à ce qu'on appelle **les structures répétitives**.

On dit aussi les structures **itératives ou boucles**.

Une structure répétitive sert à répéter un ensemble d'instructions. Il existe trois formes de structures **répétitives : POUR, TANT QUE, REPETER**.

- Les répétitions où la condition d'arrêt est placée au début (**TANT QUE**),
- Les répétitions où la condition d'arrêt est placée à la fin (**REPETER**),
- Les répétitions où le nombre d'itérations est fixé une fois pour toute (**POUR**).

5.3.1 La structure POUR

Cette structure permet de répéter des instructions un nombre connu de fois. Sa syntaxe est :

POUR compteur = val_initial **A** val_final **PAS DE** incrément
Instructions à répéter
FIN POUR

compteur c'est ce qu'on appelle compteur. C'est une variable de type entier.

val_*initial* et *val_final* sont respectivement les valeur initiale et final prise par le compteur. Ce sont des valeurs entières.

Incrément est la valeur d'augmentation progressive du compteur. La valeur par défaut du pas est de 1.

Dans de telle on peut ne pas le préciser.

Pour un pas positif, la valeur négative doit être inférieure à la valeur finale. Pour un pas négatif, valeur négative doit être supérieure à la valeur finale.

Si la valeur initiale est égale à la valeur finale, la boucle sera exécutée une seule fois.

Exemple

Ecrivons un algorithme de façon qu'il puisse calculer les moyennes de 200 élèves.

```
Algorithme moyenne
VARIABLES mat, phy, ang, fra, hg, moyenne : REELS
VARIABLE i : ENTIER
Début
    POUR i = 1  A 200
        ECRIRE "Entrez la note de math :"
        LIRE mat
        ECRIRE "Entrez la note de physique :"
        LIRE phy
        ECRIRE "Entrez la note de français :"
        LIRE fra
        ECRIRE "Entrez la note 'anglais :"
        LIRE ang
        ECRIRE "Entrez la note d'histoire-Géo :"
        LIRE hg
        moyenne ← ((mat + phy) * 5 + fra * 4 + (ang+ hg) * 2) / 18
        ECRIRE "La moyenne est : ", moyenne
    FIN POUR
Fin
```

5.3.2 La structure TANT QUE

Cette structure permet de répéter les instructions

```
TANT QUE condition
    Instructions à répéter
FIN TANT QUE
```

condition c'est une condition qu'on appelle parfois condition d'arrêt. Cette condition est testée avant la première exécution.

Cette structure diffère de la première par le fait qu'on va répéter des instructions pour un nombre de fois inconnu au préalable.

Exemple : Reprenant toujours le programme de notre surveillant. S'il ne sait pas combien de moyennes à calculer on ne pourra pas utiliser la structure POUR. Dans ce cas on est obligé d'utiliser la structure TANT QUE. Le programme sera alors :

```
Variables mat, phy, ang, fra, hg, moyenne : Réels
Variable reponse : Chaîne
DEBUT
        reponse ← "o"
        TANT QUE reponse = "o"
        Ecrire "Entrez la note de math :"
        Lire mat
        Ecrire "Entrez la note de physique :"
        Lire phy
        Ecrire "Entrez la note de français :"
        Lire fra
        Ecrire "Entrez la note d'anglais :"
        Lire ang
        Ecrire "Entrez la note d'histoire-Géo :"
        Lire hg
        moyenne ← ((mat + phy) * 5 + fra * 4 + (ang + hg) * 2) / 18
        Ecrire "La moyenne est : ", moyenne
        Ecrire "Voulez-vous continuer oui (o) /non (n) ?"
        Lire reponse
        FIN TANT QUE
FIN
```

5.3.3 La structure REPETER

Cette structure sert à répéter des instructions **jusqu'à** ce qu'une condition soit réalisée. Sa syntaxe est :

```
REPETER
        Instructions à répéter
JUSQU'A condition
```

Considérons le programme suivant :

```
Variables a , c : Entiers
DEBUT
        REPETER
                Lire a
                c ← c * c
                Ecrire c
        JUSQU'A a = 0
        Ecrire « Fin »
FIN
```

Les mots REPETER et JUSQU'A encadrent les instructions à répéter. Cela signifie que ces instructions doivent être répéter autant de fois jusqu'à ce que la variable **a** prennent la valeur 0.

Notez bien que le nombre de répétition dans cette structure n'est indiqué explicitement comme c'est la cas de la structure TANT QUE. Il dépend des données que l'on fournit au programme.

Pour bien expliciter cela, voyons ce que produira ce programme si l'on lui fournit successivement les valeurs 2, 4, 0.

Le résultat se présentera ainsi :

```
4
16
0
Fin
```

Exercices

Exercice 5.1

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 5.2
Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : « Plus petit ! », et inversement, « Plus grand ! » si le nombre est inférieur à 10.

Exercice 5.3
Ecrire un algorithme qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

Exercice 5.4
Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) :
Table de 7 :
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
…
7 x 10 = 70

Exercice 5.5
Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer :
1 + 2 + 3 + 4 + 5 = 15
NB : on souhaite afficher uniquement le résultat, pas la décomposition du calcul.

Exercice 5.6
Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle.
NB : la factorielle de 8, notée 8 !, vaut
1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

Exercice 5.7
Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dise ensuite quel était le plus grand parmi ces 20 nombres :
Entrez le nombre numéro 1 : 12
Entrez le nombre numéro 2 : 14
etc.
Entrez le nombre numéro 20 : 6
Le plus grand de ces nombres est : 14
Modifiez ensuite l'algorithme pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre :
C'était le nombre numéro 2

Exercice 5.8
Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro.
Exercice 5.9
Lire la suite des prix (en euros entiers et terminée par zéro) des achats d'un client. Calculer la somme qu'il doit, lire la somme qu'il paye, et simuler la remise de la monnaie en affichant les textes "10 Euros", "5 Euros" et "1 Euro" autant de fois qu'il y a de coupures de chaque sorte à rendre.

Exercice 5.10
Écrire un algorithme qui permette de connaître ses chances de gagner au tiercé, quarté, quinté et autres impôts volontaires.
On demande à l'utilisateur le nombre de chevaux partants, et le nombre de chevaux joués. Les deux messages affichés devront être :
Dans l'ordre : une chance sur X de gagner
Dans le désordre : une chance sur Y de gagner
X et Y nous sont donnés par la formule suivante, si n est le nombre de chevaux partants et p le nombre de chevaux joués (on rappelle que le signe ! signifie "factorielle", comme dans l'exercice 5.6 ci-dessus) :
X = n ! / (n - p) !
Y = n ! / (p ! * (n – p) !)
NB : cet algorithme peut être écrit d'une manière simple, mais relativement peu performante. Ses performances peuvent être singulièrement augmentées par une petite astuce. Vous commencerez par écrire la manière la plus simple, puis vous identifierez le problème, et écrirez une deuxième version permettant de le résoudre.

6 LES TABLEAUX

6.1 LES TABLEAUX A UNE SEUL DIMENSION

Imaginez que l'on veuille calculer la moyenne des notes d'une classe d'élèves. Pour l'instant on pourrait l'algorithme suivant :

```
Variables somme, nbEleves, Note, i : Réels
DEBUT
        somme ← 0
        Ecrire " Nombre d'eleves :"
        Lire nbEleves
        POUR i = 1 A nbEleves
        Ecrire " Note de l'eleve numero ", i , " : "
        Lire Note
        somme ← somme + Note
        FIN POUR
        Ecrire " La moyenne est de :", somme / nbEleves
FIN
```

Si l'on veut toujours calculer la moyenne des notes d'une classe mais en gardant en mémoire toutes les notes des élèves pour d'éventuels calculs (par exemple calculer le nombre d'élèves qui ont des notes supérieurs à la moyenne). Dans ce cas il faudrait alors déclarer autant de variables qu'il y a d'étudiants. Donc, si l'on a 10 élèves il faut déclarer 10 variables et si l'on a N il faut déclarer N variables et c'est pas pratique. Ce qu'il faudrait c'est pouvoir par l'intermédiaire d'une seule variable stocker plusieurs valeurs de même type et c'est le rôle des **tableaux**.

Un **tableau** est un ensemble de valeurs de même type portant le même nom de variable. Chaque valeur du tableau est repérée par un nombre appelé **indice**.

Les tableaux c'est ce que l'on nomme un **type complexe** en opposition aux types de données simples vus précédemment. La déclaration d'un tableau sera via la syntaxe suivante dans la partie des déclarations :

```
Tableau nom_tableau (nombre) : Type
```

- *nom_tableau* : désigne le nom du tableau
- nombre : désigne le nombre d'éléments du tableau. On dit aussi sa taille
- Type : c'est le type du tableau autrement dit le type de tous ces éléments

Exemples :

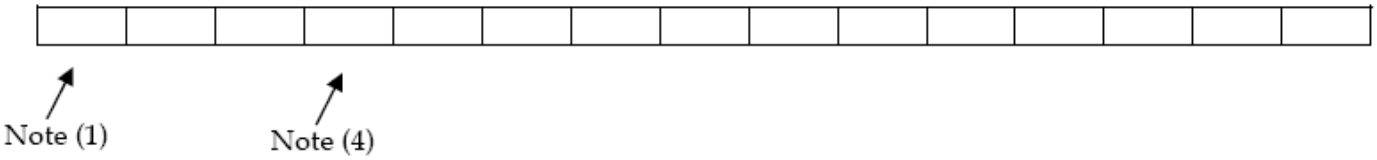
- **Tableau** Note (20) : **Réel**

Note (20) est un tableau qui contient vingt valeurs réelles.

- Tableau nom (10) , prenom (10) : Chaîne

Nom (10) et prenom (10) sont deux tableaux de 10 éléments de type chaîne.

Un tableau peut être représenté graphiquement par (exemple **Note (15)**) :



Si l'on veut accéder (en lecture ou en écriture) à la i ème valeur d'un tableau en utilisant la syntaxe suivante :

nom_tableau (indice)

Par exemple si X est un tableau de 10 entiers :

- X (2) ← - 5

met la valeur -5 dans la 2 ème case du tableau

- En considérant le cas où a est une variable de type Entier, a ← X (2)

met la valeur de la 2 ème case du tableau tab dans a, c'est- à- dire 5

- **Lire** X (1)

met l'entier saisi par l'utilisateur dans la première case du tableau

- Ecrire X (1)

Affiche la valeur de la première case du tableau

Remarques :

- **Un tableau possède un nombre maximal d'éléments défini lors de l'écriture de l'algorithme (les bornes sont des constantes explicites, par exemple 10, ou implicites, par exemple MAX). Ce nombre d'éléments ne peut être fonction d'une variable.**
- **La valeur d'un indice doit toujours :**
  - être un nombre entier
  - être inférieure ou égale au nombre d'éléments du tableau

Exemples

1. Considérons les programmes suivants:

```
Tableau X (4) : Entier
DEBUT
        X (1) ← 12
        X (2) ← 5
        X (3) ← 8
        X (4) ← 20
FIN
Tableau voyelle (6) : Chaîne
DEBUT
        voyelle (1) ← « a »
        voyelle (2) ← « e »
        voyelle (3) ← « i »
```

voyelle (4) ← « o »
voyelle (5) ← « u »
voyelle (6) ← « y »

FIN

Donner les représentations graphiques des tableaux X (4) et voyelle (6) après exécution de ces programmes.

2. Quel résultat fournira l'exécution de ce programme :

```
Variable i : Entier
Tableau C (6) : Entier
DEBUT
    POUR i = 1 A 6
        Lire C (i)
    FIN POUR
    POUR i = 1 A 6
        C (i) ← C (i) * C (i)
    FIN POUR
    POUR i = 1 A 6
        Ecrire C (i)
    FIN POUR
FIN
```

Si on saisit successivement les valeurs : 2 , 5 , 3 , 10 , 4 , 2.

Solution

1. La représentation graphique du tableau X (4) après exécution du premier programme est :

12	5	8	20

La représentation graphique du tableau voyelle (4) après exécution du deuxième programme est :

a	e	i	o	u	y

2. L'exécution du programme nous affichera successivement à l'écran :

4
25
9
100
16
4

Exercices

Exercice 6.1
Ecrire un algorithme qui déclare et remplisse un tableau de 7 valeurs numériques en les mettant toutes à zéro.

Exercice 6.2
Ecrire un algorithme qui déclare et remplisse un tableau contenant les six voyelles de l'alphabet latin.

Exercice 6.3
Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.

Exercice 6.4
Que produit l'algorithme suivant ?

```
Tableau Nb(5) en Entier
Variable i en Entier
Début
Pour i ← 0 à 5
  Nb(i) ← i * i
i suivant
Pour i ← 0 à 5
  Ecrire Nb(i)
i suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

Exercice 6.5

Que produit l'algorithme suivant ?

```
Tableau N(6) en Entier
Variables i, k en Entier
Début
N(0) ← 1
Pour k ← 1 à 6
  N(k) ← N(k-1) + 2
k Suivant
Pour i ← 0 à 6
  Ecrire N(i)
i suivant
Fin
```

Peut-on simplifier cet algorithme avec le même résultat ?

Exercice 6.6

Que produit l'algorithme suivant ?

```
Tableau Suite(7) en Entier
Variable i en Entier
Début
Suite(0) ← 1
Suite(1) ← 1
Pour i ← 2 à 7
  Suite(i) ← Suite(i-1) + Suite(i-2)
i suivant
Pour i ← 0 à 7
  Ecrire Suite(i)
i suivant
Fin
```

Exercice 6.7

Ecrivez la fin de l'algorithme 6.3 afin que le calcul de la moyenne des notes soit effectué et affiché à l'écran.

Exercice 6.8

Ecrivez un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.

Exercice 6.9

Ecrivez un algorithme calculant la somme des valeurs d'un tableau (on suppose que le tableau a été préalablement saisi).

Exercice 6.10
Ecrivez un algorithme constituant un tableau, à partir de deux tableaux de même longueur préalablement saisis. Le nouveau tableau sera la somme des éléments des deux tableaux de départ.
Tableau 1 :

4	8	7	9	1	5	4	6

Tableau 2 :

7	6	5	2	1	3	7	4

Tableau à constituer :

11	14	12	11	2	8	11	10

Exercice 6.11
Toujours à partir de deux tableaux précédemment saisis, écrivez un algorithme qui calcule le schtroumpf des deux tableaux. Pour calculer le schtroumpf, il faut multiplier chaque élément du tableau 1 par chaque élément du tableau 2, et additionner le tout. Par exemple si l'on a :
Tableau 1 :

4	8	7	12

Tableau 2 :

3	6

Le Schtroumpf sera :
3 * 4 + 3 * 8 + 3 * 7 + 3 * 12 + 6 * 4 + 6 * 8 + 6 * 7 + 6 * 12 = 279

Exercice 6.12
Ecrivez un algorithme qui permette la saisie d'un nombre quelconque de valeurs, sur le principe de l'ex 6.8. Toutes les valeurs doivent être ensuite augmentées de 1, et le nouveau tableau sera affiché à l'écran.

Exercice 6.13
Ecrivez un algorithme permettant, toujours sur le même principe, à l'utilisateur de saisir un nombre déterminé de valeurs. Le programme, une fois la saisie terminée, renvoie la plus grande valeur en précisant quelle position elle occupe dans le tableau. On prendra soin d'effectuer la saisie dans un premier temps, et la recherche de la plus grande valeur du tableau dans un second temps.

Exercice 6.14
Toujours et encore sur le même principe, écrivez un algorithme permettant, à l'utilisateur de saisir les notes d'une classe. Le programme, une fois la saisie terminée, renvoie le nombre de ces notes supérieures à la moyenne **de la classe**.

.

6.2 LES TABLEAUX MULTIDIMENSIONNELS

Nous avons vu qu'un tableau à une dimension correspond à une liste ordonnée de valeurs, repérée chacune par un **indice**.

Dans tous les langages de programmation, il est possible de définir des tableaux à deux dimensions (permettant par exemple de représenter des matrices). Ainsi, on pourra placer des valeurs dans un tableau à deux dimensions et cela consiste comme dans le cas des tableaux à une dimension à donner un nom à l'ensemble de ces valeurs. Chaque valeur est alors repérée par deux indices qui précise la position.

On déclare un tableau à deux dimensions de la façon suivante :

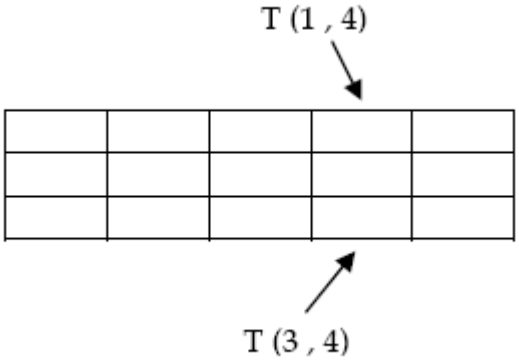
```
Tableau nom_tableau (i , j ) : Type
```

Avec

- *nom_tableau* : désigne le nom du tableau
- *i* : désigne le nombre de lignes du tableau
- *j* : désigne le nombre de colonnes du tableau
- Type : représente le type des éléments du tableau

Exemple :

Soit T (3 , 5) un tableau d'entiers. On peut représenter graphiquement par :



T (1 , 4) et **T(3 , 4)** sont deux éléments du tableau. Entre parenthèse on trouve les valeurs des indices séparées par une virgule. Le premier sert à repérer le numéro de la ligne, le second le numéro de la colonne.

On accède en lecture ou en écriture à la valeur d'un élément d'un tableau à deux dimensions en utilisant la syntaxe suivante :

```
Nom_tableau (i , j)
```

Par exemple si T est défini par : **Tableau** T (3 , 2) : Réel

- **T (2 , 1) ← -1.2**

met la valeur -1.2 dans la case 2,1 du tableau

- **En considérant le cas où a est une variable de type Réel**

a ← T (2 , 1)

met -1.2 dans a

Exemple

Considérons le programme suivant :

```
Tableau X (2 , 3) : Entier
Variables i , j , val : Entiers
DEBUT
        val ← 1
        POUR i = 1 A 2
                POUR j = 1 A 3
                        X (i , j) ← val
                        val ← val + 1
                FIN POUR
        FIN POUR
        POUR i = 1 A 2
                POUR j = 1 A 3
                        Ecrire X (i , j)
                FIN POUR
        FIN POUR
```

a. Que produit l'exécution de ce programme.

b. que produira ce programme si l'on remplace les derniers lignes par :

```
POUR j = 1 A 3
        POUR i = 1 A 2
                Ecrire X (i , j)
        FIN POUR
FIN POUR
```

Solution

L'exécution du programme donnera :

Les deux premières boucles du programme permettent de remplir le tableau. Ainsi la représentation graphique sera :

1	2	3
4	5	6

Les deux dernières boucles permettent d'afficher ces six éléments. Le résultat sera donc :

Question a

1
2
3
4
5
6

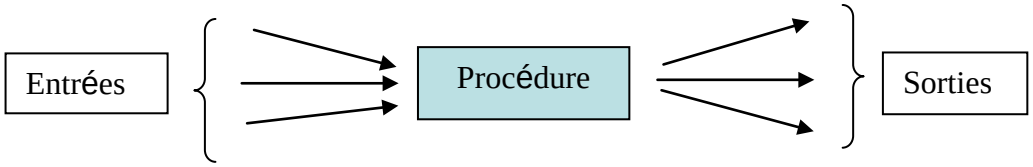
Question b

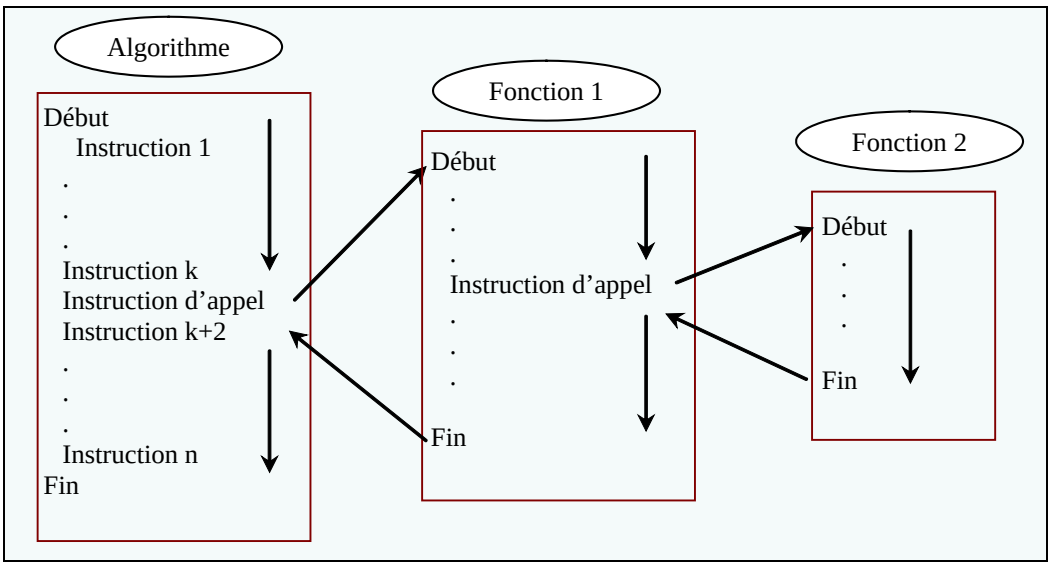
1
4
2
5
3
6

7 LES FONCTIONS ET LES PORICEDURES

Une fonction (ou procédure) est un algorithme destiné à être utilisé par d'autres algorithmes ou fonctions.

Une fonction (ou procédure) peut être assimilée à une boîte noire (un composant) qui a des entrées et des sorties.





7.1 LES FONCTIONS

Une fonction est un sous-programme qui retourne un **seul** résultat. Pour définir une fonction on utilise la syntaxe suivante :

```
FONCTION nom_fonction (Argument1 : Type, Argument2 : Type,….) : Type
Déclarations
DEBUT
    Instructions de la fonction
    nom_fonction ← Valeur renvoyée
FIN
```

On constate que la déclaration d'une fonction revient à lui préciser un nom, un type ainsi qu'une liste d'arguments.

- **Une fonction peut avoir des paramètres qui sont :**
  - soit des **données** nécessaires aux traitements réalisés par la fonction (Entrées),
  - soit des zones mémoires dans lesquelles seront stockés les **résultats** de l'exécution de la fonction (Sorties).
- **Les paramètres sont communiqués à la fonction lors de son appel.**

Par exemple, le sous-programme *sqr* permet de calculer la racine carrée d'un réel. Ce sous- programme admet un seul paramètre de type réel positif.

Le programme qui utilise *sqr* doit donner le réel positif dont il veut calculer la racine carrée, cela peut être :

- une variable, par exemple *a*
- une constante, par exemple 5.25

Les arguments d'une fonction sont en nombre fixe (≥ 0).

Une fonction possède un seul type, qui est le type de la valeur retournée qui est affecté au nom de la fonction.

Une fois la fonction définie, il est possible (en fonction des besoins) à tout endroit du programme appelant de faire appel à elle en utilisant simplement son nom suivi des arguments entre parenthèses.

Les parenthèses sont toujours présentes même lorsqu'il n'y a pas de paramètre.

Les arguments spécifiés lors de l'appel de la fonction doivent être en même nombre que dans la déclaration de la fonction et des types prévus. Dans le programme appelant ces arguments sont appelés **paramètres effectifs**.

La valeur ainsi renvoyée par la fonction peut être utilisée dans n'importante quelle expression compatible avec son type.

Exemple

```
FONCTION Calcul (x : Réel , y : Réel , z : Réel) : Réel
        Variable a : Entier
DEBUT
        a ← 3
        Calcul ← (x + y + z)  * a
FIN
```

Ça c'est un exemple de déclaration de fonction. Cette fonction appelée « Calcul » est de type réel et elle admet trois arguments de type réel.

Maintenant voici un exemple de programme complet :

```
FONCTION Calcul (x : Réel , y : Réel , z : Réel) : Réel
        Variable a : Entier
DEBUT
        a ← 3
        Calcul ← (x + y + z)  * a
FIN
Variables i , j , k , b : Réels
DEBUT
        Lire i
        Lire j
        Lire k
        b ← Calcul (i , j , k) + 2
        Ecrire b
FIN
```

Dans ce programme on fait appel a une fonction. Sa valeur est utilisée dans une expression.

7.1.1 Portée des Objets : locaux et globaux

Parmi les objets (variables, types, constantes, …) utilisés par une fonction on distingue :

- **Les objets locaux :** sont définis dans la fonction. Ces objets ont une portée limitée à la fonction.
- **Les objets globaux :** ils sont utilisés par la fonction mais sont déclarés en dehors de la fonction à un niveau supérieur; Attention aux problèmes de visibilité.
- **Les paramètres.**

Exemple

Soit le programme suivant :

```
Fonction Surface (a : Réel) : Réel
Variables valeur , resultat : Réels
DEBUT
        valeur ← 3.14
        resulat ← valeur * a
        Surface ← resultat
FIN
Variable rayon : Réel
DEBUT
        Ecrire « Entrez le rayon du cercle : »
        Lire rayon
        Ecrire « La surface de cette cercle est : » , Surface (rayon)
FIN
```

Les variables ***valeur*** et ***resultat*** déclarées dans la fonction ***Surface*** sont locales.

Considérons presque le même programme sauf que la variable ***valeur*** est déclarée maintenant dans le programme appelant.

```
Fonction Surface (a : Réel) : Réel
Variables resultat : Réels
DEBUT
        resulat ← valeur * a
        Surface ← resultat
FIN
Variable valeur , rayon : Réel
DEBUT
        valeur ← 3.14
        Ecrire « Entrez le rayon du cercle : »
        Lire rayon
        Ecrire « La surface de cette cercle est : » , Surface (rayon)
FIN
```

Dans ce deuxième programme seul la variable ***resultat*** est **locale**. Tandis que la variable ***valeur*** est devenue **globale**. On peut par conséquent accéder à sa valeur dans la fonction ***Surface***.

7.1.2 Modes de passage des paramètres

Les modes de passages des paramètres précisent comment les paramètres d′une fonction doivent être transmis au moment de l′appel.

On distingue principalement deux modes de passages :

- Passage par valeur
- Passage par adresse

a. Passage par valeur :

Le paramètre formel est traité comme une variable locale de la fonction

A chaque appel, la valeur du paramètre réel est copiée dans le paramètre formel.

Si vous transmettez une variable comme paramètre par valeur, la fonction en crée une copie; les modifications apportées à la copie sont sans effet sur la variable d′origine.

Le paramètre réel peut être une constante, une expression ou une variable de type simple ou structuré.

b. Passage par adresse :

A chaque appel, l′adresse du paramètre réel est transmise à la fonction.

Le paramètre formel est traité comme une variable dont l′adresse est celle du paramètre réel correspondant.

Toutes les modifications apportées au paramètre formel dans la fonction affecteront le paramètre réel.

Le paramètre réel doit être une variable de type simple ou structurée.

Exemple

Considérons les deux programmes suivants :

Programme 1

Fonction Calcul (a : **Réel**) : **Réel**

DEBUT

Calcul ← a * 2

a ← a – 1

FIN

Variable x : **Réel**

DEBUT

x ← 3

Ecrire Calcul (x)

Ecrire x

FIN

Programme 2

Fonction Calcul (a : **Réel**) : **Réel**

DEBUT

Calcul ← a* 2

a← a – 1

FIN

Variable x : Réel

DEBUT

x ← 3

Ecrire Calcul (x)

Ecrire x

FIN

Dans le premier programme on a un passage de paramètre par valeur et dans le deuxième on a un passage de paramètres par adresse. Le premier programme affichera le résultat suivant :

6

3

car même si la valeur de ***a*** change celle de *x* non.

Tandis que le deuxième programme il affichera :

6

2

La valeur de ***x*** changera car celle de ***a*** a changée.

7.2 LES PROCEDURES

Les procédures sont des sous- programmes qui ne retournent aucun résultat. Elles admettent comme les fonctions des paramètres.

On déclare une procédure de la façon suivante :

PROCEDURE nom_procedure (Argument1 : Type, Argument2 : Type,….)

Déclarations

DEBUT

Instructions de la procédure

FIN

Et on appelle une procédure comme une fonction, en indiquant son nom suivi des paramètres entre parenthèses.

Exemple 2

Ecrivons une procédure permettant d'échanger le contenu de deux variables de même type, cette fonction sera nommée **Echange.**

```
PROCEDURE Echanger (x : Réel , y : Réel)
Variable z : Réel
DEBUT
        z ← x
        x ← y
        y ← z
FIN
```

Exemple 3

On dispose d'une phrase dont les mots sont séparer par des point virgules. Ecrivez une procédure qui permet de remplacer les points virgules par des espaces. On suppose qu'on dispose des fonctions suivantes :

- Longueur : permet de calculer la longueur d'une chaîne de caractères.

Utilisation : Longueur (chaîne)

- Extraire : permet d'extraire une partie (ou la totalité) d'une chaîne.

Utilisation : Extraire (chaîne , position_debut, longueur)

Paramètre : chaîne de laquelle on fait l'extraction

position_debut la position à partir de laquelle va commencer l'extraction

longueur désigne la longueur de la chaîne qui va être extraite.

```
PROCEDURE Changer (chaine : Chaîne)
Variables i , l : Entier
Variables caract , schaine : Chaîne
DEBUT
        l ← Longueur (chaine)
        schaine = « «  »
        POUR i = 1 A l
                caract ← Extraire (chaine , i , 1)
                SI caract = « ; » ALORS
                        caract = «  »
                FIN SI
                schaine ← schaine & caract
        FIN POUR
        chaine ← schaine
FIN
Variable chaine : Chaîne
Variable i : Entier
DEBUT
        chaine ← « bonjour;tout;le;monde »
        changer (chaine)
        Ecrire chaine
FIN
```

8 LES STRUCTURES

Imaginons que l'on veuille afficher les notes d'une classe d'élèves par ordre croissant avec les noms et prénoms de chaque élève. On va donc utiliser trois tableaux (pour stocker les noms, les prénoms et les notes). Lorsque l'on va trier le tableau des notes il faut aussi modifier l'ordre les tableaux qui contiennent les noms et prénoms. Mais cela multiplie le risque d'erreur. Il serait donc intéressant d'utiliser ce qu'on appelle **les structures**.

Les structures contrairement aux tableaux servent à rassembler au sein d'une seule entité un ensemble fini d'éléments de type éventuellement différents. C'est le deuxième type complexe disponible en algorithmique.

A la différence des tableaux, il n'existe pas par défaut de type structure c'est-à-dire qu'on ne peut pas déclarer une variable de type structure. Ce qu'on peut faire c'est de construire toujours un nouveau type basé sur une structure et après on déclare des variables sur ce nouveau type.

La syntaxe de construction d'un type basé sur une structure est :

```
TYPE NomDuType = STRUCTURE
    attribut1 : Type
    attribut2 : Type
    . . .
    attributn : Type
FIN STRUCTURE
```

Le type d'un attribut peut être :

- **Un type simple**
- **Un type complexe**
  - Un tableau
  - Un type basé sur une structure

Exemples

```
TYPE Etudiant = STRUCTURE
        nom : chaîne
        prenom : chaîne
        note : Réel
FIN STRUCTURE
```

Etudiant
nom
prenom
note

Dans cet exemple on a construit un type *Etudiant* basé sur une structure. Cette structure a trois attributs (on dit aussi champ) : nom, prenom et note.

```
TYPE Date = STRUCTURE
        jour : Entier
        mois : Entier
        annee : Entier
FIN STRUCTURE
```

Dans ce deuxième exemple, on a construit un type *Date* basé sur une structure. Cette structure a aussi trois attributs : jour, mois et annee.

Après on peut déclarer des variables basés sur ce type. Par exemple :

```
Variable Etud : Etudiant
```

Donc Etud est une variable de type Etudiant.

Il est possible de déclarer un tableau d'éléments de ce type **Etudiant** par exemple. On pourra écrire donc :

```
Tableau Etud (20) : Etudiant
```

Etud (1) représente le premier étudiant.

Maintenant, pour accéder aux attributs d'une variable dont le type est basé sur une structure on suffixe le nom de la variable d'un point « . » suivi du nom de l'attribut. Par exemple, dans notre cas pour affecter le nom "Dinar" à notre premier étudiant, on utilisera le code suivant :

```
Etud (1).nom = '' Dianr ''
```

Exercice

On souhaite gérer les notes d'un étudiant.

1. Définir la structure «Etudiant » dont les champs sont :

Champ	Type
Nom	Chaîne
Prénom	Chaîne
Note	Tableau de 3 éléments
Moyenne	Réel

2. Ecrire l'algorithme qui permet de lire les informations d'un étudiant (nom, prénom et notes), de calculer sa moyenne et d'afficher à la fin un message sous la forme suivante :

« La moyenne de l'étudiant Dinar Youssef est : 12.45 »

où « Dinar » et « Youssef » sont les noms et prénoms lus et 12.45 est la moyenne calculée.

3. Modifier l'algorithme de l'exercice précédent de façon que l'on puisse gérer les notes de 50 étudiants.

Partie II
Le langage C

1 DEFINITION

D'après wikipédia : « Le C est un langage de programmation impératif, généraliste, traditionnellement utilisé en programmation système. Inventé au début des années 1970 avec UNIX, C est devenu un des langages les plus utilisés. De nombreux langages plus modernes comme C++, Java et PHP reprennent des aspects de C »

1.1 DEFINITION D'UN MODULE EN C

Les instructions écrites en pseudo-code vont être traduites sous forme de fonctions

Un module est défini par deux fichiers :

- Le fichier d'entête ou d'interface (module.h) contient la liste des opérations implantées par le module
- Le fichier source (module.c) contient la définition de ces opérations

Intérêt de cette approche : l'encapsulation

Encapsulation

- Fait de « cacher » au client qui utilise un module la définition exacte des opérations :
  - Pour utiliser le module, le client n'a besoin de connaître que la liste des opérations (fichier d'entête)
  - Il n'a pas connaissance des détails de l'implantation contenue dans le fichier source
- Exemple : bibliothèque graphique
- Avantages :
  - Conceptuel : réduit le risque d'erreurs
  - Économique : permet de protéger le code source

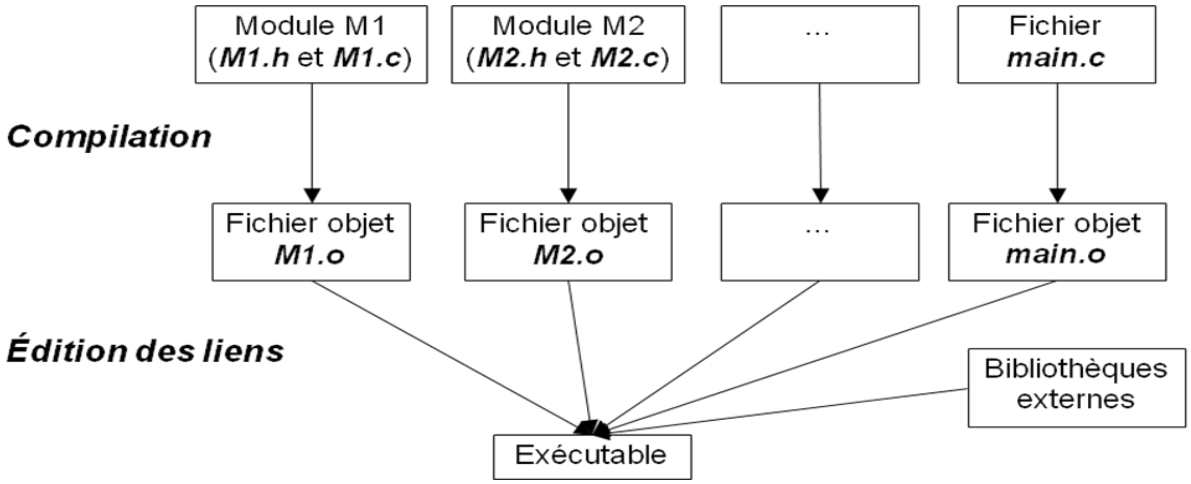
1.2 PROCESSUS DE MISE AU POINT D'UNE APPLICATION

- Mise au point de chaque module :
  - Écriture du code
  - Compilation
  - Test
- Si un module M1 utilise un module M2 : (niveaux d'interdépendance croissants)
  - Mise au point de M2
  - Inclusion de M2 dans M1
  - Mise au point de M1
- Mise au point de l'application principale (fichier main.c)

2 PROGRAMMATION MODULAIRE ET COMPILATION SEPAREE COMPILATION

- Le programme C décrit par un fichier texte appelé fichier *source* (*.c) traduit en langage machine par un compilateur
- La compilation se décompose en 4 phases:
  1. traitement par le préprocesseur
  2. la compilation
  3. l 'assemblage
  4. l 'édition de liens

- Traitement par le préprocesseur : réalise des transformations d 'ordre purement textuel (inclusion d 'autres fichiers sources, etc.) (*.i)
- Compilation : traduit le fichier généré par le préprocesseur en assembleur (*.s)
- Assemblage : transforme le code assembleur en fichier binaire appelé fichier *objet (*.o)*
- Edition de liens : lie entre différent objet (sources de fichiers séparés) pour produire un fichier *exécutable*.



Avantages

- Compilation indépendante de chaque module
- Si le code source d'un module est modifié, seuls ce module et ceux qui l'utilisent doivent être recompilés

3 LES COMPOSANTES ELEMENTAIRES DU C

Un programme en C est constitué de 6 composantes élémentaires :

- identificateurs
- mots-clefs
- constantes
- chaînes de caractères
- opérateurs
- signes de ponctuation
- + les commentaires

Identificateurs

- Un identificateur peut désigner :
  - Nom de variable ou fonction
  - type défini par typedef, struct, union ou enum,
  - étiquette
- un identificateur est une suite de caractères :
  - lettres, chiffres, « blanc souligné » (_)
- Premier caractère n'est jamais un chiffre
- minuscules et majuscules sont différenciées
- longueur>=31

Mots-clefs

Réservés pour le langage lui-même et ne peuvent être utilisés comme identificateur, 32 mots-clefs :

- const, double, int, float, else, if, etc.

Commentaires

- Débute par /* et se termine par */

/* Ceci est un commentaire */

4 STRUCTURE D'UN PROGRAMME C

- Une expression est une suite de composants élémentaires syntaxiquement correcte, par exemple :
  - x = 0
  - (i>=0) && (i<10) && (p[i] !=0)
- Une instruction est une expression suivie d 'un point-virgule (fin de l'instruction)
- Plusieurs instructions peuvent être rassemblées par des accolades { } et forme une instruction composée ou bloc, par exemple :

```
if (x !=0)
{
  z = y / x;
  t = y % x;
}
```

- Une instruction composée d 'un spécificateur de type et d 'une liste d 'identificateurs séparés par une virgule est une ***déclaration***, par exemple :

```
int a;
int b = 1, c;
char message[80];
```

- Toute variable doit faire une déclaration avant d 'être utilisée en C.
- Un programme C se présente de la façon suivante :

```
[ directives au préprocesseur ]
[ déclarations de variables externes ]
[ fonctions secondaires ]
main ()
{
déclarations de variables internes
instructions
}
```

- La fonction *main* peut avoir des paramètres formels
- Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale :

```
type ma_fonction ( arguments )
{
déclarations de variables internes
instructions
}
```

- cette fonction retourne un objet de type *type*, les arguments ont une syntaxe voisine des déclarations, par exemple :

```
int produit (int a, int b)
{
int resultat;
resultat = a * b;
return(resultat);
}
```

5 LES TYPES PREDEFINIS

C est un langage typé : toute variable, constante ou fonction est d 'un type précis.

Les types de bases de C :

- char
- int
- float double
- short long unsigned

Type caractère codé sur 7 ou 8 bits:

- particularité du type caractère en C est qu 'il est assimilé à un entier Þ on peut utiliser des expressions tel que c +1 qui signifie le caractère suivant dans le code ASCII (voir table)

```
main()
{
        char c =  'A';
        printf("%c", c+1);
}
```

5.1 LES TYPES ENTIERS

- *int* : mot naturel de la machine. Pour PC Intel = 32bits.
- int peut être précédé par
  - un attribut de précision : *short* ou *long*
  - un attribut de représentation : *unsigned*

Caractère	char	8 bits
Entier court	short	16 bits
Entier	int	32 bits
Entier long	long	32 bits

5.2 LES TYPES FOLTTANTS

- Les types float, double et long double représentent les nombres en virgule flottante :

Flottant	float	32 bits
Flottant double précision	double	64 bits
Flottant quadruple précision	long double	128 bits

- Représentation normalisée : signe,mantisse 2^{exposant}

5.3 LES CONSTANTS

- Valeur qui apparaît littéralement dans le code source, le type de constante étant déterminé par la façon dont la constante est écrite.
- 4 types : entier, réel, caractère, chaîne de caractère
- Caractère imprimable mis entre apostrophes : 'A' ou '$'
- Exception des caractères imprimables \, *'*, *? et* " sont désignés par \\, \', \? *et* \".

- Caractères non-imprimables peuvent être désignés par '*code-octal*' ou '*xcode-hexa*', par exmple : '*33*' ou '*x1b*' désigne le caractère '*espace*'.
  - \n *nouvelle ligne* \r *retour chariot*
  - \t *tabulation horizontale* \f *saut de page*
  - \v *tabulation verticale* \a *signal d'alerte*
  - \b *retour arrière*

6 LES OPERATEURS

- Affectation :
  - ***variable = expression***
  - *expression* est évalué et est affectée à *variable*.
  - L'affectation une conversion de type implicite : la valeur de l'expression est convertit dans le type du terme gauche.

```
main()
{
int i, j = 2;
float x = 2.5;
i = j + x;
x = x + i;
printf("\n %f \n",x)
}
```

6.1 LES OPERATIONS ARITHMETIQUES

- Opérateur unaire – et opérateurs binaires : +, -, *, /, % (reste de la division = modulo)
- Contrairement au langage pascal : la division entière est désignée par /. Si les 2 opérandes sont de types entier, / produira une division entière. Par exemple :
  - float x;
  - x = 3 / 2; affecte à x la valeur 1.
  - x = 3 / 2.; affect à x la valeur 1.5
- ***pow(x,y)*** : fonction de la librairie math.h pour calculer x^y.

6.2 LES OPERATEURS RELATIONNELS

- La syntaxe est ***expression1 op expression2***
  - > strictement supérieur,
  - >= supérieur ou égal,
  - < strictement inférieur,
  - <= inférieur ou égal,
  - == égal,
  - != différent
- La valeur booléenne rendu est de type int
  - 1 pour vraie
  - 0 sinon

6.3 LES OPERATEURS LOGIQUES

- Booléens :
  - ***&&*** : et logique
  - || : ou logique
  - ***!*** : négation logique

- Bit à bit :
  - **&** : et logique
  - | : ou inclusif
  - ^ : ou exclusif
  - ~ : complément à 1
  - <<(>>) : décalage à gauche (à droite)

6.4 OPERATEURS D'AFFECTATION COMPOSE

- += -= *= /= &= ^= |= <<= <<=
- Pour tout opérateur *op* :

  expression1 op= expression2

  équivaut à

  expression1 = expression1 op expression2
- *expression1* n'est évalué qu'une seule fois

6.5 OPERATEURS INCREMENTATION OU DECREMENTATION

- Incrémentation : ++
- Décrémentation : --
- En suffixe *i++* ou en préfixe *++i* : dans les deux cas la valeur de i sera incrémentée, sauf pour le premier cas la valeur affectée est l'ancienne, exemple :

```
int a = 3, b, c;
b = ++a  /* a et b valent 4*/
c = b++          /* c vaut 4 et b vaut 5 */
```

6.6 OPERATEUR VIRGULE

- Suite d'expressions séparées par une virgule
  - expression1 , expression2 , expression 3
- Expression évaluée de gauche à droite, sa valeur sera la valeur de l'expression de droite

```
main()
{
int a , b
b = ((a = 3) , (a + 2));
printf("\n b= %d\n",b);
}
```

imprime b = 5

6.7 OPERATEUR CONDITIONNEL TERNAIRE

- Opérateur conditionnel : *?*

condition ? expression1 : expression2

- Cette expression est égale à expression1 si la condition est vraie et à expression2 sinon
  - *x>=0 ? x : -x; correspond à la valeur absolue*
  - *m = ((a>b) ? a : b); affecte à m le maximum de a et b*

6.8 OPERATEUR DE CONVERSION DE TYPE

Opérateur de conversion de type, appelé cast, permet de modifier explicitement le type d'un objet :

(type) objet

```
main()
{
int i = 3 , j = 2;
printf("%f \n", (float) i/j);
}
```

Renvoie la valeur 1.5

6.9 OPERATEUR ADRESSE

L'opérateur d'adresse & appliqué à une variable retourne l'adresse-mémoire de cette valeur

&objet

7 INSTRUCTIONS DE BRANCHEMENT CONDITIONNEL

- if ----- else

```
if (expression1)
Instruction1
else if (expression2)
Instruction2
...
```

- Le else est facultatif

```
If (expression)
instruction
```

Instructions de branchement conditionnel

- Switch

```
switch (expression)
{ case constante1 :
        liste d'instructions1;
break;
  case constante2 :
        liste d'instructions2;
        break;
...
  default :
        liste d'instructionsn;
        Break;
}
```

8 LES BOUCLES

- **while :**

```
while (expression)
instruction;
```

- Tant que expression est non nulle, instruction est exécutée. Si expression est nulle instruction ne sera jamais exécutée

```
I=1;
While (i < 10)
{
        printf("\n i = %d",i);
I++;
}
```

Affiche les entiers de 1 à 9

- **do ---- while :**

```
do
instruction;
while (expression);
```

- Ici l'instruction est exécutée tant que expression est non nulle. Instruction est toujours exécutée au moins une fois.

```
int a;
do
{
        printf("\n Entrez un entier entre 1 et 10 : ");
        scanf("%d",&a);
}
while ((a <=0) || (a > 10))
```

saisie au clavier un entier entre 1 à 10

- **for :**

```
for ( expr1; expr2; expr3)
        instruction;
```

équivalent à

```
expr1;
while (expr2);
{
        instruction;
        expr3;
}
```

- *Exemple :*

```
for (i = 0; i < 10; i++)
```

```
printf("\n i = %d",i);
```

A la fin de la boucle i vaut 10

9 INSTRUCTIONS DE BRANCHEMENT NON CONDITIONNEL

- ***break*** : vu en switch, en général permet d'interrompre le déroulement d'une boucle, et passe à la première instruction qui suit la boucle.

```
main()
{
    int i;
    for (i = 0; i < 6; i++)
    {
        printf("i = %d\n ",i);
        if (i==3)
        break;
    }
    printf("valeur de i a la sortie de la boucle = %d\n,i);
}
```

imprime i= 0 jusqu'à i=3 et ensuite

valeur de i a la sortie de la boucle = 3

- ***continue*** : permet de passer directement de la boucle suivante sans exécuter les autres instructions de la boucle.

```
main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        if (i==3)
        continue;
        printf("i = %d\n",i);
    }
    printf("valeur de i a la sortie de la boucle = %d\n,i);
}
```

imprime i= 0, i=2 et i = 4

valeur de i à la sortie de la boucle = 5

10 LES FONCTIONS D'ENTREES/SORTIES CLASSIQUES

- Fonctions de la librairie standard *stdio.h* : clavier et écran, appel par la directive
  - ***#include<stdio.h>***
  - *Cette directive n'est pas nécessaire pour printf* et *scanf*.
- Fonction d'écriture *printf* permet une impression formatée:
  - *printf("chaîne de contrôle", expr1, ..., exprn);*

- *Chaîne de contrôle spécifie le texte à afficher et les formats correspondant à chaque expression de la liste.*
- *Les formats sont introduites par % suivi d'un caractère désignant le format d'impression.*

format	conversion en	écriture
%d	**int**	décimale signée
%ld	**long int**	décimale signée
%u	**unsigned int**	décimale non signée
%lu	**unsigned long int**	décimale non signée
%o	**unsigned int**	octale non signée
%lo	**unsigned long int**	octale non signée
%x	**unsigned int**	hexadécimale non signée
%lx	**unsigned long int**	hexadécimale non signée
%f	**double**	décimale virgule fixe
%lf	**long double**	décimale virgule fixe
%e	**double**	décimale notation exponentielle
%le	**long double**	décimale notation exponentielle
%g	**double**	décimale, représentation la plus courte parmi **%f** et **%e**
%lg	**long double**	décimale, représentation la plus courte parmi **%lf** et **%le**
%c	**unsigned char**	caractère
%s	**char***	chaîne de caractères

- En plus entre le % et le caractère de format, on peut éventuellement préciser certains paramètres du format d'impression:
  - %10d : au moins 10 caractères réservés pour imprimer l'entier
  - %10.2f : on réserve 12 caractères (incluant le .) pour imprimer le flottant avec 2 chiffres après la virgule.
  - %30.4s : on réserve 30 caractères pour imprimer la chaîne de caractères, mais que 4 seulement seront imprimés suivis de 26 blancs
- Fonction de saisie *scanf* : permet de saisir des données au clavier.

scanf("chaîne de contrôle",arg1,arg2,...,argn);

- Chaîne de contrôle indique le format dans lequel les données lues sont converties, ne contient pas le caractère "\n". Même format que *printf* une légère différence.

```
#include <stdio.h>
main()
{
    int i;
    printf("entrez un entier sous forme hexadécimale i =");
    scanf("%x",&i);)
    printf("i = %d\n",i);
}
```

Si la valeur 1a est saisie alors le programme affiche i = 26

- Impression et lecture de caractères :

getchar() et putchar() : fonctions d'entrées/sorties non formatées

- *getchar();* retourne un int, on doit écrire :

caractere = getchar();

lorsqu'elle détecte la fin du fichier elle retourne l'entier EOF valeur définie dans le *stdio.h* et vaut -1.

- *putchar(caractere)*; retourne un int ou EOF en cas d'erreur.

```
#include <stdio.h>
main()
{
    char c;
    while ((c = getchar()) != EOF)
    putchar(c);
}
```

on peut utiliser un fichier texte.

11 TABLEAUX ET POINTEURS EN C

11.1 DECLARATION D'UN TABLEAU

- Déclaration d'un tableau de taille *n* :

  type nom_tableau [n] ;

  ⇒ réservation de *n* cases contiguës en mémoire

- ex. : déclaration d'un tableau d'entiers de taille 10 appelé *tb1* :

  int tb1[10] ;

0	1	2	3	4	5	6	7	8	9
?	?	?	?	?	?	?	?	?	?

Accès aux éléments d'un tableau

- Modification du contenu d'un élément :

  ⇒ ex. : tb1[3] = 19 ;

⇒

0	1	2	3	4	5	6	7	8	9
?	?	?	19	?	?	?	?	?	?

Pour remplir tout le tableau : modification élément par élément…

- Utilisation de la valeur d'un élément :

  ⇒ ex. : x = tb1[3] + 1 ;

  ⇒ Pour afficher tout le contenu du tableau : affichage élément par élément…

Affichage d'un tableau X de réels de taille n

```
void affTableau (float X[ ], int n)
{
    int i ;
    for (i = 0 ; i < n ; i++)
        printf (" Contenu de la case %d : %g", i, X[i]) ;
}
```

Mise à zéro des éléments d'un tableau X de réels de taille n

```
void razTableau (float X[ ], int n)
{
        int i ;
        for (i = 0 ; i < n ; i++)
                X[i] = 0 ;
}
```

Attention : passage par ***adresse*** dans le cas des tableaux

```
void main ()
{
        float montableau [100] ;

        razTableau (montableau, 100) ;
        affTableau (montableau, 100) ;
}
```

⇒ Affiche le résultat suivant :

Contenu de la case 0 : 0

Contenu de la case 1 : 0

Contenu de la case 2 : 0

…

⇒ La modification dans la procédure razTableau est visible dans la procédure principale…

Initialisation au moment de la déclaration

- Ex.: int tb1[10] = { 21, 32, -4, 1, 37, 88, 9, -1, 0, 7} ;

0	1	2	3	4	5	6	7	8	9
21	32	-4	1	37	88	9	-1	0	7

- Initialisation d'une partie du tableau : int tb1[10] = { 21, 32, -4, 1} ;

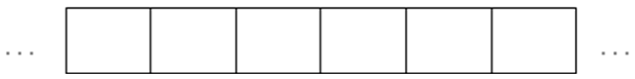
0	1	2	3	4	5	6	7	8	9
21	32	-4	1	0	0	0	0	0	0

- Initialisation sans spécifier la taille : int tb1[ ] = { 21, 32, -4, 1} ;

0	1	2	3
21	32	-4	1

12 ADRESSES ET POINTEURS

- La mémoire (RAM) est un immense tableau de cases contenant chacune 1 octet (ou *byte*) = 8 *bits*



- Une variable est stockée sur une ou plusieurs cases en fonction de son type et de la machine. En général :
  - Entiers « courts » (*short int*): 2 octets

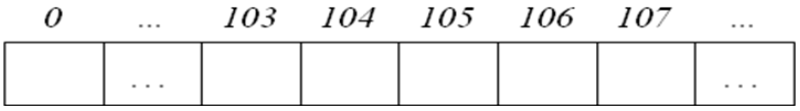
- Entiers (*int*) et réels en simple précision *(float)*: 4 octets
- Réels en double précision (*double*): 8 octets

⇒ Intervalle de valeurs possibles plus ou moins grand

Notion d'adresse

- L'adresse d'une variable correspond au numéro de la première case stockant cette variable

Ex: entier à l'adresse 103 est en fait stocké dans les cases 103, 104, 105 et 106

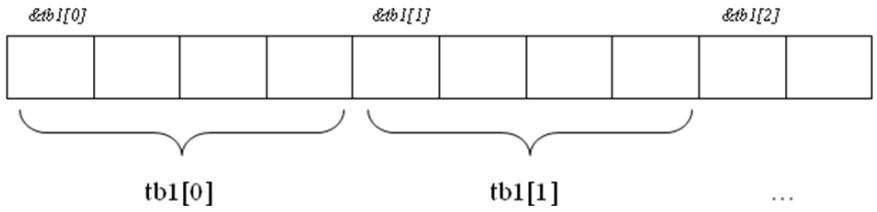


- Le nombre d'adresses disponibles dépend de la taille de la mémoire mesurée en Go

⇒ 10^9 octets ⇒ milliards de cases…

Opérateur &

- En langage C, l'adresse en mémoire d'une variable est donnée par &variable
- Même principe dans le cas des tableaux :
  - &(tb1[i]) correspond à l'adresse en mémoire de tb1[i]



- tb1 est en fait l'adresse en mémoire du premier élément du tableau, c'est-à-dire à &(tb1[0])

Arithmétique d'adresses dans les tableaux

- Les adresses-mémoire du tableau sont contiguës et toutes les valeurs sont stockées sur le même nombre de cases

⇒ on peut utiliser les opérations arithmétiques + et –

- Exemple 1:

tb1 + 4 correspond à l'adresse de la 1ère case à laquelle on ajoute 4 adresses, c'est-à-dire à &(tb1[4])

⇒ tb1 ou tb1 + 0 ⇔ &(tb1[0])

tb1 + 1 ⇔ &(tb1[1])

tb1 + 2 ⇔ &(tb1[2])

…

- Exemple 2 :

&(tb1[5]) – 2 correspond à l'adresse de tb1[5] à laquelle on soustrait 2 adresses ⇒ &(tb1[3])

- Exemple 3 :

&(tb1[7]) – &(tb1[3]) correspond au nombre d'adresses entre tb1[7]) et tb1[3]) ⇒ ***4***

Notion de pointeur

- Un ***pointeur*** est une variable permettant de stocker une adresse-mémoire
- Déclaration :

type *pointeur ;

⇒ le type est celui des variables auxquelles permet d'accéder le pointeur

- Exemples :

```
int *pt_int ; /* pointeur sur un entier */
char *pt_char; /* pointeur sur un caractère */
```

Affectation d'un pointeur

- Affectation de l'adresse d'une variable :

```
int x, *px ;
px = &x ;
```

- Affectation de l'adresse d'un tableau :

```
int tb1[10], *p ;
p = tb1; /* p pointe sur la première case de tb1 */
ou
p = &(tb1[0]) ;
```

Arithmétique de pointeurs

- L'addition et la soustraction fonctionnent comme avec les adresses…

```
int tb1[10], *p1, *p2 ;
p1 = tb1 + 3 ; /* p1 pointe sur tb1[3] */
p2 = p1 – 1 ;  /* p2 pointe sur tb1[2] */
```

Opérateur *

- Permet d'accéder au contenu d'une variable par un pointeur :

```
int x, y, *p ;
x = 10 ;
p = &x ;
y = *p ; /* y doit contenir 10 */
```

- Fonctionne aussi avec les adresses :

*(tab + 3) est équivalent à tab[3]

Notion de pointeur

int i = 3

*int *p;*

p = &i;

objet	adresse	valeur
i	431836000	3
p	431836004	431836000

*Toute modification de *p modifie i.*

```
main()
{
int i = 3, j = 6;
int *p1, *p2;
p1 = &i;
p2 = &j;
p1 = p2;
}
```

```
main()
{
int i = 3, j = 6;
int *p1, *p2;
p1 = &i;
p2 = &j;
*p1 = *p2;
}
```

avant la dernière ligne des deux programmes

objet	adresse	valeur
i	431836000	3
j	431836004	6
p1	431835984	431836000
p2	4831835992	431836004

*après l'affectation de *p2 à *p1*

objet	adresse	valeur
i	431836000	6
j	431836004	6
p1	431835984	431836000
p2	4831835992	431836004

après l'affectation de p2 à p1

objet	adresse	valeur
i	431836000	3
j	431836004	6
p1	431835984	431836004
p2	4831835992	431836004

13 LES FONCTIONS

Définition d'une fonction

C'est la donnée du texte de son algorithme qu'on appelle corps de la fonction.

type nom-fonction (type-1 arg-1, …, type-n arg-n)

{

[déclarations de variables locales]

liste d'instructions

}

- *type* désigne le type de la fonction i.e. le type de la valeur qu'elle retourne
- si la fonction ne renvoie pas de valeur elle est de type *void*.
- La fonction se termine par l'instruction *return* :
  - *return(expression);* *expression* du type de la fonction
  - *return;* fonction sans type
- Ex :

```
int produit (int a, int b)
{
return(a * b);
}
void imprime_tab (int *tab, int nb_elements)
{
        int i;
        for ( i = 0; i < nb_elements, i++)
                printf("%d \t",tab[i]);
```

```
int puissance (int a, int n)
{
        if ( n == 0)
           return(1);
        return(a * puissance(a, n-1));
}
```

```
    printf("\n");
    return;
}
```

Appel d'une fonction

- L'appel de fait par:
  - *nom-fonction(para-1,para-2, …, para-n);*

Déclaration d'une fonction

C n'autorise pas les fonctions imbriquées.

- On peut déclarer une fonction secondaire soit avant, soit après la fonction principale *main*.
- Toutefois, il est indispensable que le compilateur "connaisse" la fonction à son appel. Elle doit impérativement être déclarée avant :
  - type nom-fonction (type-1, …, type-n);

```
int puissance (int , int)p;
int puissance (int a, int b)
{
if (n == 0)
return(1)
return(a * puissance(a, n-1));
}
main()
{
int a = 2, b = 5;
printf("%d\n",puissance(a,b));
}
```

Durée de vie des variables

- Les variables manipulées dans un programme C n'ont pas la même durée de vie. 2 catégories de variables :
  - variables permanentes (ou statiques) : occupent une place mémoire durant toute l'exécution du programme (*segment de données*). Elles sont initialisées à zéro par le compilateur et elles sont caractérisées par le mot-clef *static*.
  - variables temporaires : se voient allouer une place mémoire de façon dynamique (*segment de pile*). Elles ne sont pas initialisées. Leur place mémoire est libérée à la fin d'exécution de la fonction secondaire. Variables dites automatique. Elles sont spécifiées par le mot-clef *auto* (rarement utilisé).
- Variable globale : variable déclarée en dehors des fonctions.

```
int n;
void fonction ();
void fonction ()
{
n++;
printf("appel numero %d\n",n);
return;
}
```

```
n est initialisée à 0 par le compilateur
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

```
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
```

- Variable locale : variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instruction).

```
int n = 10;
void fonction ();
void fonction ()
{
int n = 0;
n++;
printf("appel numero %d\n",n);
return;
}
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
```

- Variable locale : variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instruction).

```
int n = 10;
void fonction ();
void fonction ()
{
int n = 0;
n++;
printf("appel numero %d\n",n);
return;
}
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
```

```
n est initialisée à 0 par le compilateur
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

- Il est possible de déclarer une variable locale de classe statique qui reste locale à une fonction mais sa valeur est conservée d'un appel au suivant : *static type nom-de-variable;*

```
int n = 10;
void fonction ();
void fonction ()
{
static int n;
n++;
printf("appel numero %d\n",n);
return;
}
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
```

```
n est initialisée à 0 par le compilateur
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

Transmission des paramètres d'une fonction

- *Les paramètres de fonction sont traités de la même manière que les variables locales de classe automatique. On dit que les paramètres d'une fonction sont transmis par valeurs.*
- *Exemple :*

```
void echange (int , int);
void echange (int a, int b)
{
int t;
printf("debut fonction :\n a = %d \t b = %d\n",a,b);
t = a;
a = b;
b = t;
printf("fin fonction : \n a = %d \t b = %d\n",a,b);
return;
}
main()
{
    int a = 2, b = 5;
    printf("debut programme  principal : \n a = %d \t b = %d\n",a,b);
    echange(a,b);
    printf("fin programme  principal : \n a = %d \t b = %d\n",a,b);
}
```

imprime

debut programme principal :

a = 2 b = 5

debut fonction :

a = 2 b = 5

fin fonction :

a = 5 b = 2

fin programme principal :

a = 2 b = 5

- Pour q'une fonction modifie la valeur de ses arguments, il faut il faut passer les paramètres par adresse :

```
void echange (int *, int*);
void echange (int *adr_a, int *adr_b)
{
int t;
t = *adr_a;
*adr_a = *adr_b;
*adr_b = t;
return;
}
main()
{
    int a = 2, b = 5;
    printf("debut programme  principal : \n a = %d \t b = %d\n",a,b);
    echange(&a,&b);
    printf("fin programme  principal : \n a = %d \t b = %d\n",a,b);
}
```

La fonction main

- La fonction principale *main* est une fonction comme les autres. Elle est souvent déclarée sans type mais l'option -*Wall* de *gcc* provoque un message d'avertissement.
- En fait la fonction main est de type int dont la valeur est 0 si l'exécution se passe bien différente de 0 sinon
- On peut utiliser deux constantes définies dans la librairie *stdlib.h* :
    - *EXIT_SUCCESS* = 0
    - *EXIT_FAILURE* = 1
- En principe la fonction main sans arguments e pour prototype :
    - *int main(void)*
- La fonction *main* peut également posséder des paramètres formels.
- En effet un programme C peut recevoir une liste d'arguments au lancement de son exécution.
- La ligne de commande est dans ce cas là est composée du nom du fichier exécutable suivi par des paramètres.
- main possède 2 paramètres formels appelés par convention :
    - *argc* (argument count) : variable de type *int* fourni le nombre de mots composant la ligne de commande y compris l'exécutable.

- *argv* (argument vector) : est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande.
    - *argv[0]* contient le nom du fichier exécutable
    - *argv[1]* contient le premier paramètre
    - …

*int main (int argc, char *argv[]);*

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
int a, b;
if (argc !=3)
{
        printf("\nErreur : nombre invalide d'arguments");
        printf("\nUsage : %s int int \n",argv[1]);
        printf(EXIT_FAILURE);
        }
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        printf("\nLe produit de %d par %d vaut : %d\n", a, b, a * b);
        return(EXIT_SUCCESS);
}
```

On lance l'exécutable avec deux paramètres : *a.out 12 8*

Pointeur sur une fonction

- *Le langage C offre la possibilité de passer une fonction comme paramètre d'une autre fonction. On utilise un mécanisme de pointeur*
- *un pointeur sur une fonction ayant pour prototype*

  type fonction (type-1,type-2,…,type-n);

  est de type

  type (*)(type-1,…,type-2);
- Ex :

*int operateur_binaire(int a, int b, int (*f)(int, int));*

- sa déclaration est donnée

int operateur_binaire(int , int , int (*)(int, int));

- pour appeler la fonction opérateur en utilisant la fonction somme de prototype : *int somme(int, int);*
- on écrit : *operateur_binaire(a, b, somme)*

Dans le corps de la fonction *operateur_binaire* on écrit (*f)(a,b) :

*int operateur_binaire(int a, int b, int (*f)(int, int))*

```
{
        return((*f)(a,b));
}
```

```
#include<stdlib.h>
    #include<stdio.h>
#include<string.h>
void usage(char *);
int somme(int, int);
int produit(int , int);
int operateur-binaire(int, int, int (*)(int, int));
void usage(char *cmd)
{
    printf("\nUsage :%s int [plus|fois] int\n",cmd);
}
int somme (int a, int b)
{
    return(a + b);
}
int produit (int a, int b)
{
    return(a * b);
}
int operateur_binaire (int a, int b, int (*f)(int, int))
{
    return((*f)(a, b));
}
```

```
int main(int argc, char *argv[]);
{ int a, b;
  if (argc !=4)
  { printf("\nErreur : nombre invalide d'arguments");
    usage(argv[0]);
      return(EXIT_FAILURE);
  }
  a = atoi(argv[1]);
  b = atoi(argv[1]);
  if (!strcmp(argv[2], "plus"));
     { printf("%d\n",operateur_binaire(a,b,somme));
     return(EXIT_SUCCESS);
     }
  if (!strcmp(argv[2], "fois"));
     { printf("%d\n",operateur_binaire(a,b,produit));
     return(EXIT_SUCCESS);
     }
  else
     { printf("\nErreur : argument(s) invalide(s)");
     usage(argv[0]);
     return(EXIT_Failure);
     }
}
```

14 LA PROGRAMMATION MODULAIRE

Principes élémentaires

- Nécessité de fractionner un programme C en plusieurs fichiers sources, que l'on compile séparément
- 3 règles d'écriture d'un programme C :
    - l'abstraction des constantes littérales
    - la factorisation du code
    - la fragmentation du code
- L'abstraction des constantes littérales : éviter d'utiliser explicitement des constantes littérales dans le corps, ceci rend les modifications et la maintenance difficile :
    - *fopen("nom_fichier","r");*
    - *perimetre = 2 * 3.14 * rayon;*
    - sauf le cas particulier des constantes symboliques au moyen de la directive *#define*.
- La factorisation du code : éviter de dupliquer du code. Définition systématiquement des fonctions (même de petite taille)
- la fragmentation du code : découpage d'un programme en plusieurs fichiers pour plus de lisibilité.

- Placer une partie du code dans un fichier en-tête (ayant l'extension .h) que l'on inclut dans le programme principale par la directive #include.

```
/*************************************************************/
/****  fichier : main.c                                   ****/
/****  saisit 2 entiers et affiche leur produit           ****/
/*************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "produit.h"
int main(void)
{
    int a, b , c;
    scanf("%d",&a);
    scanf("%d",&b);
    c = produit(a,b);
    printf("\nle produit vaut %d\n",c);
    return EXIT_FAILURE
}
/*************************************************************/
/****  fichier : produit.h                                ****/
/****  produit de 2 entiers                               ****/
/*************************************************************/
int produit (int , int)
int produit (int a, int b)
{
    return(a * b);
}
```

La compilation séparée

- Si on reprend l'exemple précédent, on doit compiler les fichiers séparément :
  - *gcc -c produit.c*
  - *gcc -c main.c*
  - *gcc main.o produit.o*
  - Si on compile avec l'option -*Wall*, à la compilation il y aura un message de warning qui rappelle que la fonction *produit* n'a pas été déclarée dans le *main*.
  - On peut également faire une seule commande : *gcc produit.c main.c*
- Fichier en-tête d'un fichier source :
  - à chaque fichier source *nom.c* un fichier en-tête *nom.h* comportant les déclarations des fonctions non locales au fichier *nom.c* (ces fonctions sont appelées *fonctions d'interface*) ainsi que les définitions des constantes symboliques et des macros qui sont partagées par les 2 fichiers.
- Fichier en-tête d'un fichier source :
  - le fichier en-tête *nom.h* doit être inclus par la directive *#include* dans tous les fichiers sources qui utilisent une des fonctions définies dans *nom.c*.
  - il faut faire, par ailleurs, précéder la déclaration de la fonction du mot-clef *extern*, qui signifie que cette fonction est définie dans un autre fichier.
  - Exemple :

```
/**********************************/
/****  fichier  :  produit.h          *****/
/****  en-tete  de produit.c          *****/
/**********************************/
extern int produit (int , int);
/**********************************/
/****  fichier  :  produit.c          *****/
/****  produit de 2 entiers           *****/
/**********************************/
#include "produit.h"
int produit (int a, intb)
{
    return(a *b);
}
```

```
/*****************************************/
/****  fichier  :  main.c                          *****/
/****  saisit de 2 entiers et affiche leur produit****/
/*****************************************/
#include<stdlib.h>
#include<stdio.h>
#include "produit.h"

int main (void)
{
      int a, b, c:
      scanf("%d",&a);
      scanf("%d",&b);
      c = produit(a,b);
      printf("\n le produit vaut %d\n",c);
      return EXIT_SUCCESS;
}

gcc produit.c main.c
```

- Pour éviter une double inclusion de fichier en-tête, il est recommandé de définir une constante symbolique, souvent appelée *NOM_H* au début du fichier nom.h dont l'existence est précédemment testée.
- Si cette constante est définie alors le fichier *nom.h* a déjà été inclus.

```
/**********************************/
/****  fichier  :  produit.h          *****/
/****  en-tete  de produit.c          *****/
/**********************************/
#ifndef  PRODUIT_H
#define PRODUIT_H
extern int produit (int , int);
#endif   /* PRODUIT_H*/
```

- Les règles :
  - à tout fichier source *nom.c* on associe un fichier en-tête *nom.h* qui définit son interface
  - le fichier *nom.h* se compose :
    - déclarations des fonctions d'interface
    - éventuelles définitions de constantes symboliques et de macros
    - éventuelles directives au pré-processeur
  - le fichier *nom.c* se compose :
    - variables permanentes
    - des fonctions d'interface dont la déclaration se trouve dans *nom.h*
    - éventuelles fonctions locales à *nom.c*
  - le fichier *nom.h* est inclus dan *nom.c* et dans tous les fichiers qui font appel à une fonction définies dans *nom.c*

L'utilitaire *make*

- Plusieurs fichiers sources compilés séparément ⇒ compilation longue est fastidieuse ⇒ automatisation à l'aide de *make* d'UNIX
- Principe de base :
    - Avec *make*, effectuer uniquement les étapes de compilation nécessaires à la création d'un exécutable.
    - *make* recherche par défaut le fichier *makefile* ou *Makefile* dans le répertoire courant.
    - On peut utiliser un autre fichier dans ce cas lancer la commande *make* avec l'option -f :
        - *make -f nom_fichier*

Création d'un *Makefile*

Cible : liste de dépendances

<TAB> commande UNIX

- fichier cible ensuite la liste des fichiers dont il dépend (séparés par des espaces)
- après <TAB> il y a les commandes (compilation) UNIX à exécuter dans le cas où l'un des fichiers de dépendances est plus récent que le fichier cible.

```
## Premier exemple de Makefile
prod : produit.c main.c produit.h
        gcc -o prod produit.c main.c
```

- l'exécutable prod dépend des 2 fichiers produit.c, main.c et l'en-tête produit.h (les commentaires sont précédés de #)

make prod

- Dans le premier exemple on n'utilise pas pleinement les fonctionnalités de *make*

```
## Deuxieme exemple de Makefile
prod : produit.o main.o
        gcc -o prod produit.o main.o
main.o : main.c produit.h
        gcc -c main.c
produit.o : produit.c produit.h
        gcc -c produit.c
```

- On peut rajouter, si on veut être bien organisé, une cible appelée *clean* permettant de nettoyer le répertoire courant :
    - clean : *rm -f prod *.o*
- Pour simplifier l'écriture d'un fichier Makefile, on peut utiliser un certain nombre de macros sous la forme :

  nom_de_macro = corps de la macro
- quand la commande *make* est exécutée, toutes les instructions du type *$(nom_de_macro)* dans le *Makefile* sont remplacées par le corps de la macro.

```
## Exemple de Makefile avec macros
CC = gcc
prod : produit.o main.o
        $(CC) -o prod produit.o main.o
main.o : main.c produit.h
        $(CC) -c main.c
produit.o : produit.c produit.h
```

$(CC) -c produit.c

15 LES DIRECTIVES AU PREPROCESSEUR

Directives

- Préprocesseur est un programme exécuté lors de la première compilation
- il effectue des modifications textuelles sur le fichier source à partir de *directives* :
  - incorporation de fichiers source (#include)
  - définition de constantes symboliques et de macros (#define)
  - compilation conditionnelle (#if, #ifdef, …)

Directive *#include*

- Permet d'incorporer dans le fichier source le texte figurant dans un autre fichier :
  - fichier en tête de la librairie standard (stdio.h, math.h, …)
  - n'importe quel autre fichier
- Syntaxe ,:
  - *#include<nom-de-fichier>* : fichier se trouvant dans les répertoires systèmes (ex : /usr/include/)
  - *#include "nom-de-fichier"* : fichier dans le répertoire courant
- Possibilité de spécifier d'autres répertoires à l'aide de l'option -I du compilateur (voir ci-après)

Directive *#define*

- Permet de définir :
  - des constantes symboliques
    - *#define nom reste-de-la-ligne* : demande au préprocesseur de substituer toute occurrence de *nom* par la chaîne *reste-de-la-ligne*
    - ex : #define NB_LIGNES 10

```
#define NB_COLONNES 33
#define TAILLE_MATRICE NB_LIGNES * NB_COLONNES
```

  - des macros avec paramètres
    - *#define nom(liste-de-paramètres) corps-de-la-macro* : où liste-de-paramètres est une liste d'identificateurs séparés par des virgules
    - ex : #define MAX(a,b) (a>b ? a : b)

    le préprocesseur remplacera toutes les occurrences du type MAX(x,y) par (x > y ? x : y)

- Exemple :

```
#define IF if(
#define THEN ){
#define ELSE } else {
#define ELIF } else if (
#define FI ;}
#define BEGIN {
#define END }
#define SWITCH switch(
#define IN ){
#define FOR for(
    #define WHILE while(
    #define DO ){
    #define OD ;}
#define REP do{
    #define PER }while(
#undef DONE
    #define DONE );
#define LOOP for(;;){
```

#define ENDSW } #define POOL }

- Et voici un exemple de code :

```
assign(n,v)
NAMPTR n;
STRING v;
{
IF n->namflg&N_RDONLY
THEN failed(n->namid,wtfailed);
ELSE replace(&n->namval,v);
FI
}
```

La compilation conditionnelle

- A pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur
- Permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage
- directive en 2 catégories liée :
  - à la valeur d'une expression
  - à l'existence ou l'inexistence de symboles
- Condition liée à la valeur d'une expression

```
#if codition-1
partie-du-programme-1
#elif condition-2
partie-du-programme-2
#elif condition-n
partie-du-programme-n
#else partie-du-programme-∞
```

  - le nombre de *#elif* est quelconque et le *#else* est facultatif
  - chaque *condition-i* doit être une expression constante
  - une *seule partie-du-programme* sera compilée : celle qui correspond à la première *condition-i* non nulle
  - Exemple :

```
#define PROCESSEUR ALPHA
#if PROCESSEUR == ALPHA
taille_long = 64;
#elif PROCESSEUR == PC
taille_long = 32;
#endif
```

- Condition liée à l'existence d'un symbole

```
#ifdef symbole
partie-du-programme-1
```

#else

partie-du-programme-2

#endif

- si symbole est défini au moment où l'on rencontre la directive *#ifdef*, alors *partie-du-programme-1* sera compilée dans le cas contraire c'est

- Exemple :

```
#define DEBUG
    ...
#ifdef DEBUG
for (i = 0; i < N; i++)
    printf("%d\n",i);
#endif
```

- si la ligne *#define DEBUG* existe, l'instruction *for* sera compilée
- On peut remplacer l'instruction *#define DEBUG* au moment de la compilation par *gcc -DDEBUG fichier.c*