Algorithmique
Notions fondamentales

Bertrand LIAUDET

SOMMAIRE

Remarque : tout ce qui est précédé par « ++ » est à bien savoir !!!

INTRODUCTION

Edition sept 2019

1. Algorithme et algorithmique en général

Algorithmique

Nom commun : science des algorithmes. Étude des **objets** et des **méthodes** permettant de produire des algorithmes.

Adjectif : relatif aux algorithmes.

Algorithme

Définition 1

Texte décrivant sans ambiguïté une méthode pour résoudre un problème.

Définition 2

Texte décrivant sans ambiguïté une **suite d'actions ordonnée** à effectuer pour arriver au(x) résultat(s) attendu(s). En général, certains éléments sont fournis au départ pour effectuer les opérations.

Bilan

Un algorithme est une recette ! Dans l'idée d'un algorithme, il y a l'idée d'une **séquence** : une suite d'action ordonnée.

Domaine d'application

Un algorithme peut s'appliquer à n'importe quel domaine.

Mathématique

Algorithme du calcul du PGCD de deux nombres (Euclide au 3ème siècle avant JC).

- Éléments de départ : 2 entiers
- Résultat : le PGCD
- Opérations : des opérations mathématiques.

Informatique

Algorithme d'affichage d'un fond d'écran animé.

- Élément de départ : aucun.
- Résultat : l'affichage du fond d'écran.
- Opération : des opérations mathématiques et informatiques.

Cuisine

Algorithme (recette) du gâteau au chocolat

```
Recette du gâteau au chocolat
    Séparer le blanc des jaunes.
    Mélanger le sucre aux jaunes.
    Etc.
    Faire cuire
Fin
```

Eléments de départ : les ingrédients.

Résultat : le gâteau.

Opérations : les opérations de cuisine.

Le document de montage d'un meuble en kit

Eléments de départ : les planches, les vis, etc.

Résultat : le meuble (une armoire, par exemple).

Opérations : les opérations de montage.

Plusieurs façon de faire

Différents chemin possibles

Quel que soit l'algorithme, il y a plusieurs façon de faire.

Qualité des chemins : notion d'optimisation

La façon de faire sera considérée comme plus ou moins bonne en fonction de 2 critères principalement :

- La simplicité : il faut que ce soit facile à comprendre, autant que possible, pour pouvoir le corriger, le mettre à jour.
- L'efficacité : il faut que l'algorithme aille à une vitesse acceptable pour les utilisateurs.

Conclusion

C'est plus ou moins bien fait ! Ca vaut pour la description d'une recette comme pour les algorithmes informatiques !

Notion d'entrée, de sortie et d'instruction

Les actions décrites dans un algorithme manipulent des objets fournis au départ pour permettre de produire le ou les résultats.

- Les ENTREES sont les éléments de départ.
- Les SORTIES sont les résultats produits.
- Les INSTRUCTIONS sont les actions qui sont décrites dans l'algorithme.

Distinction entre les types d'algorithme

Trois éléments distinguent les différents types d'algorithme :

- Le type des SORTIES : produire un gâteau ou produire un nombre.
- Le type des ENTREES : des œufs ou des tableaux de nombres.
- Le type des INSTRUCTIONS : mélanger les œufs et le sucre ou comparer la valeur de deux données.

2. Algorithme informatique

Informatique

L'informatique est la science du traitement automatique de l'information.

Les anglo-saxons parlent de « computeur science ».

Est informatique ce qui est relatif à cette science.

Algorithme informatique

Les algorithmes informatiques sont ceux qui décrivent des traitements automatisées d'informations qui ont lieu dans des ordinateurs.

Objets et méthode de l'informatique

Algorithmique

L'algorithmique est l'étude des **objets** et des **méthodes** permettant de produire des algorithmes.

Objets

Les objets de l'informatiques, c'est l'information. Elle est organisée dans des variables qui vont contenir l'information.

En POO, les objets vont inclure leurs méthodes.

Les tableaux, les listes, les arbres, les pointeurs sont des objets spéciaux qu'on utilise en algorithmique.

Méthodes

Les méthodes sont les chemins parcourus (les suites d'instructions) permettant de résoudre un problème, quel que soit sa complexité.

Tests, boucle, fonctions sont des méthodes algorithmiques qui valent pour tous les algorithmes.

On a aussi des méthodes spéciales comme les méthodes de tri, les méthodes de recherche dans un ensemble, les méthode de parcours d'arbre et aussi la récursivité.

++ Ecran et clavier

L'écran et le clavier d'un ordinateur sont **les deux objets fondamentaux** qu'on va utiliser.

- Le clavier permet de récupérer les entrées des algorithmes.
- L'écran permet d'afficher les résultats des algorithmes.

Tous les algorithmes ne fonctionnent pas avec un écran et un clavier.

- Les entrées peuvent provenir de divers périphériques ou de fichiers.
- Les sorties peuvent être des fichiers ou des informations pour divers périphériques.
- Quand on écrira des procédures et des fonctions, on n'aura ni écran, ni clavier, ni aucun périphérique d'entrée ou de sortie.

3. Langages algorithmiques

Les 3 sortes de langages algorithmiques

- Les langages textuels.
- Les langages graphiques (diagramme d'activités UML, diagrammes de séquence UML).
- Les langages symboliques.

Ces trois sortes de langage ont les mêmes objectifs et ont la même sémantique. La différence se situe uniquement au niveau de la présentation (graphie et syntaxe).

Les langages textuels (pseudo-code)

Ce sont des langages qui décrivent l'algorithme :

- avec un langage alphabétique
- en écrivant les instructions les unes en dessous des autres
- en utilisant un principe d'indentation.

De ce fait, il y a une représentation spatiale à deux dimensions de l'algorithme.

Les langages textuels sont les langages utilisés le plus communément pour écrire des algorithmes : on parle aussi de **pseudo-code**.

➢ ***Exemple :***

```
PGCD (a, b)  // entier
/*   PGCD : plus grand commun diviseur de 2 entiers
     par exemple, pgcd(42, 30) = 6 car 42=6*7 et 30=6*5
*/
     tant que a != b // on boucle tant que les 2 entiers sont différents l'un de l'autre
          si a > b
              a = a-b
          sinon
              b = b-a
          fsi
     ftq
     return a ;
fin
```

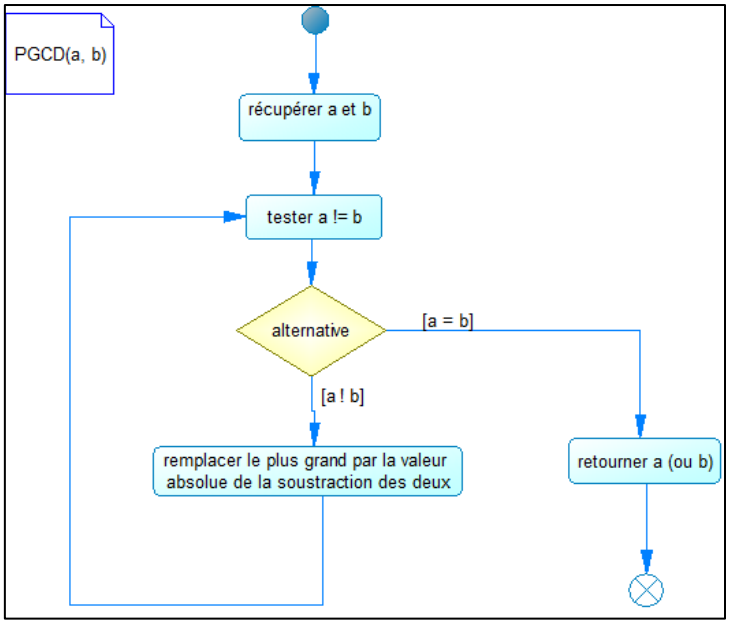
Les langages graphiques (organigrammes, diagramme d'activités)

Ce sont des langages qui utilisent des symboles graphiques à deux dimensions pour écrire l'algorithme.

On retrouve les organigrammes avec les diagrammes d'activités d'UML.

Les langages graphiques (organigrammes) ne sont plus utilisés pour décrire précisément les algorithmes informatiques de type procédure ou fonction car le formalisme prend trop de place. On les utilise généralement pour des descriptions plus générales (par exemple pour décrire le déroulement d'un cas d'utilisation avec ses différents scénarios ou décrire l'organisation générale d'une entreprise).

- ***Exemple avec le PGCD :***



Les langages symboliques

Ce sont des langages qui décrivent l'algorithme :

- avec un langage symbolique
- en écrivant les instructions les unes à la suite des autres sur la même ligne
- donc, sans utiliser de principe d'indentation.

Les langages symboliques ne sont utilisés que par quelques théoriciens. Leur principal intérêt est de pouvoir écrire très rapidement des algorithmes sur un ticket de métro !

Avec ce type d'algorithme, il n'y a pas de commentaires : l'idée est de faire court !

➢ ***Exemple :***

```
↑  ↓↓
PGCD(a, b) [ {/a !=b  a<b ? a←a-b | b←b-a  ¿ } return a]
```

Les différentes approches du pseudo-code (des langages textuels)

Il y a plusieurs approches dans l'usage du pseudo-code :

- avec ou sans **déclaration des types**.
- avec ou sans **déclaration des variables locales**.
- avec ou sans **usage de débranchements structurés** (break, return, continue).
- avec ou sans **duplication débuts de bloc**.
- avec ou sans **marquage des blocs** (numérotation par exemple)

Les commentaires

// **:** pour mettre des commentaires en fin de ligne ou sur une seule ligne.

/* … */ : pour mettre des commentaires sur plusieurs lignes.

On considère le /* comme un marqueur de début de bloc et le */ comme un marque de fin de bloc pour que ce soit bien lisible.

Exemle du PGCD textuel ci-dessus.

FORMALISME CHOISI

Le choix de ce cours suivra le principe suivant : **RIEN DE TROP !** L'écriture algorithmique se limitera au strict nécessaire tout en s'assurant de la clarté de l'algorithme.

On se rapproche d'une écriture qui ressemble, dans la forme, à du Python, avec des fin de blocs en plus.

Ce qui conduit à choisir les options suivantes :

- **Pas de déclaration des types** des paramètres formels grâce à l'utilisation d'une convention de nommage en fonction des types. Si nécessaire, on précise les types en commentaire.
- Usage des flèches pour préciser le **mode de passage des paramètres formels**. Par défaut, si on ne met rien, c'est en entrée. On peut aussi préciser le mode de passage entre parenthèses (in), (out), (inout)
- **Commentaires de l'en-tête**, sous l'en-tête, avant le corps, précisant la signification des variables et leurs types, l'objectif de l'algorithme, le principe de résolution, si nécessaire.
- En cas de fonction, **précision du type renvoyé en fin d'en-tête** avec commentaire de fin de ligne (//) précisant la signification et/ou le type de ce qui est renvoyé.
- Déclaration facultatives des variables locales dans les procédures et les fonctions.
- Usage de débranchements structurés.
- Pas de duplication des débuts de bloc.
- Marquage des blocs par numérotation de type numérotation de paragraphe facultatif, mais à savoir faire !

On essaiera de respecter ces principes d'écriture en n'oubliant pas :

- un langage alphabétique
- en écrivant les instructions les unes en dessous des autres
- un principe d'indentation des blocs.

Tester ses algorithmes : en python

Pour tester ses algorithmes, on peut utiliser n'importe quel langage. Mieux vaut toutefois qu'il soit simple d'utilisation et de syntaxe.

Le langage le plus adapté tant qu'on ne travaille pas directement avec des pointeur est le Python.

- ***Exemple :***

```
def pgcd(a,b): # retourne un entier : le pgcd
    while(a!=b):
        if a>b:
            a=a-b;
        else:
            b=b-a;
    return b;
```

Le code python est très proche du langage algorithmique qu'on va utiliser.

Ce sera donc facile de traduire les codes algorithmiques en python.

On donnera des exemples de base du code python tout au long du cours pour traduire les notions algorithmiques abordées.

- ***Avantage et inconvénient d'un langage***

L'avantage d'un langage est qu'il permet de tester son code.

L'inconvénient est qu'il toujours poser beaucoup de problèmes syntaxiques.

L'autre inconvénient est qu'il existe de nombreux langages avec chacun leurs spécificités.

4. Les paradigmes de programmation, d'après Bjarnes Stroustrup (C++)

Cf. POO janvier 2019

Paradigme 0 : pdg. du débutant : « Tout dans le main »

Codez tout dans le programme principal.
Débrouillez-vous comme vous pouvez.
Faîtes des tests jusqu'à ce que ça marche !

Ce paradigme est celui du débutant. C'est ce qu'on fait quand on commence la programmation.

Paradigme 1 : pdg. de la programmation procédurale

Choisissez les procédures (=fonctions).
Utiliser les meilleurs algorithmes que vous pourrez trouver.

Le problème est celui du bon découpage du main en procédures.

Il faut définir les entrées et les sorties pour chaque procédure.

Un principe général est de distinguer entre l'interface utilisateur (la saisie et l'affichage) et le calcul.

Ces 2 premiers paradigmes sont la bases de l'algorithmique. C'est ceux qu'on va mettre en œuvre.

Paradigme 2 : pdg. de la programmation modulaire : masquer l'information

Choisissez vos modules (fichiers avec des fonctions et regroupements de fichiers)

Découpez le programme de telle sorte

que les données soient masquées par les modules

Le paradigme de la programmation modulaire est un **aboutissement de la programmation procédurale**.

Il consiste à **regrouper les fonctions dans des fichiers** et à **organiser ces fichiers en modules** (un module est un regroupement de fichiers) qui permettent de **masquer les données** du programme.

Techniquement, il conduit à l'utilisation de variables globales, de static, d'extern, de compilation conditionnelle, d'espace des noms, d'organisation du type : outils.c, outils.h (interface), utilisation.c.

Les parties « masquage de l'information » et « regroupement des fonctions dans un fichier » du paradigme est rendue obsolète par l'usage de la programmation objet.

La partie « organisation des fichiers » reste présente.

Paradigme 3 : pdg. de l'abstraction des données : type structurés = classe

Choisissez les types dont vous avez besoin.
Fournissez un ensemble complet d'opérations pour chaque type.

Une classe est un type, en général structuré, auquel on ajoute des procédures appelées méthodes.

Une classe en tant que type qui peut donner lieu à la fabrication d'une variable appelée « objet ».

Fabriquer un type structuré consiste à regrouper des types (simples ou déjà structurés) dans un même type structuré : c'est déjà un mécanisme d'abstraction.

Principe d'encapsulation

Le principe d'encapsulation est le principe de fonctionnement du paradigme d'abstraction.

Y sont associées les notions de : **visibilité** des attributs et des méthodes, **constructeur**, **destructeur**, **surcharge**.

Paradigme 4 : pdg. de l'héritage

Choisissez vos classes.
Fournissez un ensemble complet d'opérations pour chaque classe.
Rendez toute similitude explicite à l'aide de l'héritage.

Principe de substitution

Le principe de substitution est le principe de fonctionnement de l'héritage.

Y sont associées les notions de : **polymorphisme**, **redéfinition**, **interface**, **implémentation**, **classe et méthode abstraites**.

Paradigme 5 : pdg. de la généricité (type variable)

Choisissez vos algorithmes.
Paramétrez-les de façon qu'ils soient en mesure de fonctionner
avec une variété de types et de structures de donnée.

La généricité au sens stricte consiste à ce que le type des paramètres des algorithmes devienne un paramètre lui aussi. Elle s'appuie souvent sur l'utilisation d'interfaces.

Paradigme final : pdg. de la productivité : factoriser encore plus le code

- **Les interfaces :** elles permettent d'élargir la possibilité de coder de la généricité stricte et sont largement utilisées dans les design patterns. Une interface est un type abstrait (tandis que la classe est un type concret). Ce type abstrait est utilisé de façon générique dans le code. Seule l'instanciation concrète différencie ensuite les comportements.
- **Les design patterns :** ce sont des solutions classiques à des petits problèmes de codage. Ils s'appuient souvent sur les interfaces.
- **Les patterns d'architecture :** ce sont des solutions classiques d'architecture. Le MVC est un pattern d'architecture.
- **Les bibliothèques :** elles offrent des méthodes permettant d'éviter de les réécrire et organisant même la façon de réfléchir à la solution des problèmes à résoudre.
- **Les framework :** ce sont des architecture semi-finies qu'on peut va ensuite paramétrer et compléter en fonction des spécificités du produit à réaliser.

5. Utilité de l'algorithmique

- L'algorithmique permet d'analyser un problème complet en le découpant en sous-problèmes, sans avoir à rentrer dans les détails. Si c'est bien fait, ça accélère considérablement le travail.
- L'algorithmique permet de mieux comprendre le fonctionnement des « boîtes noires » et des interfaces et donc des API en générale.
- L'algorithmique permet de passer facilement d'un langage à un autre. Il faudra toutefois en plus avoir compris les principes de la programmation objet pour être à l'aise avec tous les langages.
- L'algorithmique permet de s'habituer aux structures de données fondamentales associées aux algorithmes.

NOTIONS FONDAMENTALES

1. Programme, afficher, instruction, bloc, indentation, sortie

Premier exemple

```
Programme AfficherBonjour
/*      Entrée : aucune
        Sortie : affichage de « Bonjour »
*/
        ecrire (« Bonjour ») ;
Fin
```

Mots-clés

Les mots-clés sont les mots constants du langage algorithmique. Dans l'exemple proposé, ils sont soulignés. Dans la pratique, on ne les souligne pas. On a ici 3 mots-clés : **Programme**, **Ecrire** et **Fin**.

Programme

Un programme est un algorithme. On lui donne un nom (« afficherBonjour «).

Commentaires

On précise les entrées et les sorties du programme. Ici pas d'entrée, une sortie : l'affichage de « Bonjour ».

Entrée

Il n'y a pas d'entrée : l'algorithme n'a besoin de rien pour fonctionner.

Sortie

Une sortie est un résultat produit par le programme. Ici la sortie c'est l'affichage de « Bonjour ».

Ecrire

Il n'y a qu'une seule instruction : l'instruction **ecrire**( ) (ou **afficher**() ou **print**( )).

Ecrire est une instruction du langage algorithmique. C'est l'instruction qui permet d'écrire quelque chose à l'écran.

Notion d'instruction

« ecrire » est une instruction. Une instruction est une action à réaliser. En général, et écrit les instructions les unes en dessous des autres et on termine chaque instruction par un point-virgule. C'est toutefois facultatif.

Notion de bloc

L'algorithme commence avec « Programme » et finit avec « Fin ». Tout ce qui se trouve entre « Programme » et « Fin » constitue un bloc d'instructions, c'est-à-dire un ensemble d'instructions. « Programme » marque le début du bloc. « Fin » marque la fin du bloc.

Indentation : question de présentation

Le bloc d'instructions est décalé (d'une tabulation) par rapport au début du bloc qui la contient. La fin du bloc est mise juste en dessous du début du bloc.

Programme AfficherBonjour

La première ligne consiste à donner un nom au programme juste après le mot-clé : Programme.

Forme générale d'un programme

Programme *NomDuProgramme*

    Instructions ;

Fin

2. Lire, variable, affectation, expression, évaluation, entrée

Deuxième exemple

```
Programme Fahrenheit
/*      E : lecture au clavier du celsius
        S : affichage du fahrenheit
*/
        lire (celsius)
        fahr = celsius * 9 / 5 + 32
        ecrire (fahr) ;
Fin
```

Variables

On a deux variables dans l'algorithme : Fahr et Celsius.

Entrée

Une entrée est une une information fournie au programme pour qu'il puisse faire son calcul. Elle peut avoir diverses provenances. Ici l'entrée, c'est la valeur de Celsius qui est lue au clavier.

Affectation

« fahr = 9 / 5 celsius + 32 ; » est une affectation.

Une affectation comporte trois parties :

- A gauche : le nom d'une variable
- Au milieu : le symbole d'affectation : « ← ». On utilise aussi le symbole « = » ou aussi « := » mais ce dernier ne doit alors pas être confondu avec le l'opérateur booléen d'égalité qu'on écrit souvent « == ».
- A droite : une expression mathématique à évaluer

Une affectation est une instruction. C'est l'instruction de base de toute la programmation.

L'affectation est l'instruction qui permet de donner la valeur de l'expression de droite à la variable de gauche (appelée parfois « left value »).

Expression et évaluation

Les algorithmes contiennent souvent des expressions mathématiques.

Ici : « Celsius * 9 / 5 + 3 »

Ces expressions sont évaluées selon les règles d'évaluation classiques des mathématiques.

L'évaluation d'une expression retourne une valeur qui sera utilisée dans l'instruction où l'expression se l'expression se trouve.

++ Affectation et évaluation

Evaluation et Affectation sont **les deux actions élémentaires fondamentales** de tout algorithme.

Evaluer consiste à trouver la valeur d'une expression.

Affecter consiste à mettre une valeur dans une variable.

++ Lire

La fonction de lecture permet de récupérer la valeur d'une variable

A cette fonction, on passe la ou les variables à lire.

Les valeurs saisies au clavier sont affectées dans les variables de la fonction lecture.

```
lire (Celsius)
```

Est équivalent à :

```
celsius = lire ( )
```

Est équivalent à :

```
celsius = Clavier
```

On affecte le clavier (ce qui est saisi au clavier) dans la variable Celsius.

On peut passer plusieurs paramètres à la fonction de lecture :

```
lire (a, b, c)
```

++ Ecrire

Pour afficher la valeur d'une variable, on la passe en paramètre à la fonction afficher. A noter qu'il n'y a plus de guillemets : on utilisait les guillemets pour distinguer les textes qu'on veut afficher des variables qu'on veut aussi afficher.

```
ecrire (fahr) ;
```

Est équivalent à :

```
Ecran = fahr
```

On affecte l'écran (on affiche à l'écran) la valeur de la variable Fahr.

++ Forme générale d'un programme

```
Programme
        Lecture
        Traitement
        Affichage
Fin
```

Cette forme est très importante à retenir.

++ Circulation de l'information

Un programme consiste en une circulation d'information.

Notre algorithme peut être décrit ainsi :

- L'utilisateur choisit une valeur dans sa tête
- Cette information est transmise à ses mains
- Ses mains transmettent l'information au clavier
- Le clavier transmet l'information à la variable « Celsius »

- La variable « Celsius » transmet l'information à la variable « Fahr »
- La variable « Fahr » transmet l'information à l'écran
- L'écran transmet l'information aux yeux de l'utilisateur
- Les yeux de l'utilisateur transmettent l'information au cerveau de l'utilisateur.

Retour sur la notion d'instruction

++ Toute instruction peut se ramener à une ou plusieurs affectations.

- instructions-affectations qui permettent de récupérer de l'information de l'extérieur : « lire ».
- instructions-affectations qui permettent d'envoyer de l'information à l'extérieur : « écrire ».
- instructions-affectations qui permettent de modifier les informations dans le programme : « affectation » et « appel de procédures » (les appels de procédure seront abordés un peu plus tard).

Bilan du vocabulaire d'un algorithme

Dans un algorithme on trouve :

- Des mots-clés (noms fixes) : Programme, Afficher, Lire, ←
- Des noms variables : Fahrenheit, Celsius, Fahr, etc.
- Des valeurs : 9, 32, « bonjour », etc.
- Des expressions : 9 / 5 celsius + 32, etc. Une expression est constituée de valeurs, de variables et d'opérateurs (+, /, etc.).

++ Vérification et simulation

Principe

Pour vérifier un algorithme, une méthode consiste à simuler son exécution et à regarder le contenu des variables après chaque instruction.

Mise en œuvre

Pour mettre en œuvre la simulation, il faut faire un tableau. Chaque colonne contient une variable. Chaque ligne correspond à une instruction. On remplit au fur et à mesure la valeur des variables dans le tableau.

Exemple

- ***Algorithme***

```
Programme Fahrenheit
    lire (celsius)
    fahr = celsius * 9 / 5 + 32
    ecrire (fahr) ;
Fin
```

- ***Simulation***

	Celsius	Fahr	Ecran
lire (celsius)	20		
fahr ← celsius * 9 / 5 + 32		68	
ecrire (fahr) ;			68

3. Variables, types et expression

Distinction entre variable mathématique et variable informatique

En mathématique, la variable est un symbole, généralement une lettre, défini de telle sorte que ce symbole peut être remplacé par **0, 1 ou plusieurs valeurs**. Les valeurs possibles d'une variable dans une situation donnée (x dans une équation) ne sont **pas forcément connues**. Les exercices d'algèbres consistent souvent à mettre au jour les valeurs possibles pour les variables. L'ensemble des valeurs possible pour une variable n'est **pas modifiable**.

En informatique, une variable a toujours **1 et 1 seule valeur**. Cette valeur est toujours **connue**. Cette valeur **peut être modifiée**.

++ Description d'une variable informatique

++ Caractéristiques d'une variable

- un nom : celsius : c'est l'identificateur de la variable. Le nom peut faire référence à la valeur ou au contenant selon son usage.
- une valeur (ou contenu): 3 : c'est la traduction en fonction du type de l'état physique réel de la variable.
- un contenant : c'est le bout de mémoire dans lequel la valeur est stockée. Il est symbolisé par un carré.
- une adresse : ad_celsius, c'est la localisation du contenant.
- un type : Il permet de définir l'ensemble des valeurs possibles pour la variable (entier, réel, caractère, booléen, etc.).
- une signification (ou sens) : par exemple, celsius c'est la température en °C. La signification est conventionnelle (c'est le programmeur qui la choisit). Mais elle est très importante.

Représentation schématique des variables

Il faut se représenter clairement ce qu'est une variable C'est quelque chose où je peux mettre une information (un tiroir, une page blanche, une ardoise, un contenant). Cette chose est fixe : elle a une position géographique (c'est son adresse : ad_n) et un nom (n) qui me facilite la vie (c'est pour cela qu'il vaut mieux qu'il soit significatif, sinon on pourrait la nommer par son adresse!). Le nom, l'adresse et le contenant ne sont pas modifiables.

Une variable a aussi un contenu : c'est sa valeur, c'est-à-dire l'information qu'elle contient. On peut modifier la valeur. Une variable a toujours un contenu.

Du bon usage : bien nommer les variables

Il faut nommer significativement les variables !

En effet, dans le programme précédent, on pourrait appeler fahr et celsius x et y, ou même celsius et fahr ! Comme ça, on aurait toutes les chances de ne plus rien y comprendre ! Pensez à la relecture!

++ Double usage du nom d'une variable

Le nom d'une variable fait référence soit la valeur, soit le contenant.

- Quand le **nom** "celsius" est utilisé dans une expression à évaluer, "celsius" c'est la **valeur** de "celsius". "celsius" est utilisé en entrée de l'affectation : il est utilisé mais pas modifié.
- Quand le **nom** fahr est utilisé à gauche de l'affectation. fahr c'est le **contenant** de fahr. On parle aussi de "lvalue" ("left value", "valeur à gauche" de l'affectation, ou "valeur_g"). fahr est utilisé en sortie de l'affectation : il est modifié mais pas utilisé.

++ Les 3 types élémentaires

Le type permet de définir l'ensemble des valeurs possibles pour la variable.

Les types élémentaires sont ceux qui permettent de stocker une seule information.

Il y a **3 types de base sont :**

- **Les nombres.** Certains langages distinguent entre les **Entiers** : domaine de définition = IN et les **Réels** : domaine de définition = IR. Les réels sont souvent appelés « float ».
- **Les chaînes de caractères** : elles correspondent à du texte. Les chaines de caractères sont appelées « string ». Les chaînes de caractères ne sont pas vraiment un type élémentaire. Les chaînes de caractères sont représentées entre guillemets ou entre apostrophes selon les langages. Une chaîne de caractères contient des caractères. Certains langages définissent le type **Caractère** : domaine de définition = l'alphabet + les chiffres, la ponctuation, les symboles, etc. Un caractère est codé sur 1 octet : il y a donc 256 caractères possibles. Les caractères, quand ils existent, sont représentés entre apostrophes. Dans un algorithme, tout ce qui est mis entre apostrophes est un caractère : 'a' est un caractère, ' »' est un caractère, 'F12' est un caractère (celui correspondant à la touche F12 du clavier), etc.
- **Les booléens** : domaine de définition : {vrai, faux}. Certains langages ne proposent pas de type booléen.

Type et convention algorithmique

Dans les algorithmes, on précisera le type que quand ce sera nécessaire.

On choisira les noms de variables de telle sorte qu'ils correspondent au type.

Voici les noms habituellement associés aux types élémentaires :

- Entier : i, j, k, n, m
- Réel: x, y, z
- Caractère : c, c1, c2
- Chaîne de caractères : ch, ch1, ch2, …, ou st1, st2, st3 (st pour string).
- Booléen : flag, fg, fg1, drapeau, dp, dp1, bool, ok
  et les symboles  pour un drapeau à vrai et  pour un drapeau à faux.

++ Expression et évaluation

Constituants d'une expression

Une expression est constituée par :

- Des constantes littérales : 3, -4.5, 'C', « bonjour »
- Des noms de variables
- Des opérateurs : +, -, *, /, etc.
- Des noms de fonctions : sin( ), cos( ), racine( )
- Des parenthèses

➢ ***Remarques :***

On ne doit pas utiliser le symbole de mise au carré dans une expression algorithmique (5^2), mais soit faire l'opération (5*5), soit se doter d'une fonction qui fait l'opération : carré(5).

De même, on ne doit pas écrire : $\frac{4*x}{5+y}$ mais $(4 * x) / (5+y)$

Dans tous les cas, les expressions doivent être ramenées à une expression sur une seule ligne en utilisant des opérateurs standards, des fonctions et des parenthèses.

Evaluation d'une expression

Pour trouver la valeur d'une expression, on évalue l'expression, ce qui consiste à faire les opérations contenue dans l'expression en suivant l'ordre de priorité des opérateurs, des fonctions et des parenthèses.

Valeur et type d'une expression

Une expression à une valeur qui est d'un certain type parmi les types élémentaire.

On verra que les expressions peuvent aussi avoir un type structuré (ce qui sera d'autant plus vrai avec l'utilisation de la « surcharge » des opérateurs).

++ L'affectation

L'affectation ou assignation est l'opération qui permet de donner une valeur à une variable.

« x ← 3 » veut dire que x prend la valeur 3 (on met 3 dans x).

« x ← 3 * y » veut dire qu'on met le résultat de l'évaluation de 3 * y dans x.

« x ← x + 1 » veut dire qu'on met le résultat de l'évaluation de x * 1 dans x, autrement dit, x est incrémenté de 1.

« 5 ← x + 1 » ne veut rien dire ! On ne peut pas mettre x+1 dans 5 !

Python de base

```
celsius = float(input('entrez une temperature en Celsius : '))
fahr = celsius * 9 / 5 + 32
print (celsius, " degrés Celsius = ", fahr, " degrés Fahrenheit ")
```

4. ++ Test

Troisième exemple

```
Programme racineCarré
        Lire (x)
        si x  < 0 alors
                afficher(« Pas de racine »)
        sinon
                racine = racineCarrée(x)
                afficher (racine)
        finsi
Fin
```

si – sinon – finsi

Pour pouvoir effectuer une instruction sous condition, on utilise le « si ».

Si une condition est réalisée, alors, on exécute la série d'instructions du bloc « si ». Le début du bloc « si » est marqué par le « si », la fin du bloc « si » est marquée par le « sinon », ou bien, s'il n'y a pas de « sinon », par le « finsi ».

Le sinon correspond à l'alternative du si : ici le cas sinon correspond à x >= 0.

Le finsi vient fermer le bloc ouvert par le sinon.

Evaluation de la condition

La condition (x <0 dans l'exemple) est une expression de type booléen qui est évaluée. Elle vaut soit VRAI, soit FAUX. Si elle vaut vrai, c'est le bloc « si » qui sera exécuté. Si elle vaut faux, c'est le bloc « sinon ». A noter que FAUX est équivalent à 0 et VRAI à tout sauf 0.

Bloc d'instructions

Un « si » ouvre un bloc d'instructions : les instructions dans le bloc sont donc décalées d'une tabulation par rapport au « si ».

Le « alors » est facultatif.

Le « sinon » ferme le bloc d'instructions du « si » : il est donc placé sous le « si ».

Le « sinon » ouvre aussi un nouveau bloc d'instructions : les instructions du cas « sinon » sont donc décalées d'une tabulation par rapport au « sinon ».

Le « finsi » ferme le bloc d'instruction du « sinon » ou celui du « si » quand il n'y a pas de « sinon ». Il est donc placé sous le « sinon » ou sous le « si ».

++ Principe d'imbrication

Dans un bloc instruction, on peut mettre n'importe quel type d'instruction. On peut donc imbriquer les instructions les unes dans les autres, les tests dans les tests, dans les boucles, etc.

En cas d'imbrications multiples et d'algorithme long, on peut numéroter les débuts et fin des blocs d'instructions dans une logique de paragraphe : B1, B1.1, B1.2, B1.1.1, etc.

On peut aussi tirer un trait entre le début et la fin du bloc.

Fonction racine

Dans l'exercice, il ne s'agit pas de trouver une méthode pour calculer la racine carrée. On peut donc utiliser la fonction racineCarrée. C'est un principe général. On peut se donner toutes les fonctions mathématiques dont on a besoin, sauf quand l'exercice précise qu'il faut la fabriquer.

Remarque sur l'écriture algébrique dans les algorithmes

Dans les algorithmes, on n'utilise jamais de barres de fraction, ni de symboles comme racine carrée, carré, ou autre.

Les expressions algébriques doivent être formées avec les 4 opérateurs de base : +, -, *, / et avec les parenthèses en respectant l'ordre de priorité des opérateurs, et avec des fonctions du genre de racineCarrée(x), carré(x), puissance(x, n), log(x), ln(x), factoriel(n), etc.

Remarque sur la structure générale du programme

On retrouve le « lire » en premier.

Ensuite, traitements et affichage sont un peu mixés.

Malgré cela, la structure : « lire, traiter, afficher » reste pertinente.

++ Les 3 différents types de tests

si – sinon - finsi

```
si condition alors
        bloc1
sinon
        bloc2
finsi
```

Le sinon est facultatif :

```
si condition alors
        bloc1
finsi
```

si – sinonsi – etc. – sinon - finsi

```
si condition alors
        bloc1
sinonsi
        bloc2
sinonsi
        bloc3
…
sinon
        bloc4
finsi
```

Le sinon est facultatif.

Switch

```
switch variable vaut
cas 1 : expression ou liste d'expressions
        bloc1
cas 2 : expression ou liste d'expressions
        bloc2
…
défaut
        blocDefaut
finSwitch
```

Le « switch» fait une comparaison d'égalité entre la variable et l'expression ou une liste d'expressions.

Le « défaut » est facultatif.

++ Expression booléenne

La condition est une expression booléenne.

Une expression booléenne est une comparaison (égalité ou inégalité). « a >0 » est une expression booléenne dont l'évaluation renverra « vrai » ou « faux » selon la valeur de « a ».

On peut aussi utiliser les opérateurs ET et OU dans l'expression. « a > 0 ET b < 3 ». L'évaluation de cette expression renvoie à la table de vérité de l'opérateur ET.

++ Méthode d'analyse d'un algorithme avec tests : l'arbre dichotomique

Toute situation avec des tests pourra être analysée en construisant un arbre de décision dichotomique.

++ Vérification et simulation

Principe

Pour vérifier un algorithme, une méthode consiste à simuler son exécution et à regarder le contenu des variables après chaque instruction.

Mise en œuvre

Pour mettre en œuvre la simulation, il faut faire un tableau. Chaque colonne contient une variable. Chaque ligne correspond à une instruction. On remplit au fur et à mesure la valeur des variables dans le tableau.

Exemple

- ***Algorithme***

```
Programme RacineCarré
    Lire (x)
    si x  < 0 alors
        afficher(« Pas de racine »)
    sinon
        racine = racineCarrée(x)
        afficher (racine)
    finsi
Fin
```

- ***Simulation***

	x	x < 0	racineCarrée(x)	racine	écran
Lire (x)	4				
x < 0		faux			

racineCarrée(x)			2		
racine = racineCarrée(x)				2	
Afficher (racine)					2

- ***Simulation***

	x	x < 0	écran
Lire (x)	-4		
x < 0		vrai	
Afficher(« Pas de racine »)			Pas de racine

Python de base

```
from math import * # importation des functions mathématiques
x=float(input('entrez un reel : '))
if x<0 :
    print("pas de racine")
else :
    racine = sqrt(x)
    print ("la racine de ",x," vaut ", racine)
    print ("la racine de ",x," vaut ", round(racine,2)) # round arrondit à 2 chiffres
    print ("la racine de %g vaut %.2f" % (x, racine)) # %g : adapté, %.2 : 2 après la
                                                                                virgule
```

5. ++ Boucle

Quatrième exemple – boucle tant que (while)

On veut afficher tableau de conversion des fahrenheits en celsius pour des fahrenheits allant de 0 à 300 de 20 en 20.

```
Programme listeFahr
    fahr = 0
    Tant que fahr < = 300
        celsius = 5 * (fahr – 32) / 9
        afficher (fahr, celsius)
        fahr = fahr + 20
    Fin tant que
Fin
```

Tant que

Pour pouvoir répéter des actions on utilise une boucle. Ici une boucle « tant que ».

Les actions répétées sont celles situées dans le bloc de la boucle.

Principe de fonctionnement

Quand on arrive sur la ligne du tant que, il y une condition à vérifier.

Le « tant que » ouvre un bloc d'instructions.

Si la condition n'est pas vérifiée, on passe directement après le bloc d'instruction

Si la condition est vérifiée, on exécute les instructions du bloc.

Arrivé en fin de bloc, on remonte à la ligne du « tant que » et on recommence.

On peut noter que la variable testée dans le « tant que », ici « fahr », a été initialisée avant d'entrer dans le « tant que » : initialisée à 0 en l'occurrence.

On peut noter aussi que la variable testée est modifiée dans le corps de la boucle : ici elle est incrémentée de 20, ce qui l'amènera à 300 progressivement.

Cinquième exemple – boucle pour (for)

On veut afficher les 10 premiers entiers et leurs carrés en partant de 1

```
Programme lesCarrés
        Pour i de 1 à 10
                afficher ( i + ' au carré = ' +  i*i )
        Fin pour
Fin
```

Boucle pour

Pour pouvoir répéter des actions on utilise une boucle. Ici une boucle « pour ».

On choisit la boucle pour quand on sait combien de fois on va boucler.

Les actions répétées sont celles situées dans le bloc de la boucle.

Construction de l'affichage

afficher (i + ' au carré = ' + i*i)

On utilise le signe « + » pour concaténer les morceaux d'affichage.

D'abord la variable « i » concaténée à la chaine de caractère « au carré = » concaténé au réslutat de l'expression « i*i »

Principe de fonctionnement

Quand on arrive sur la ligne du pour pour la première fois, la variable i est initialisée à la valeur fournie, ici 1.

Le « pour» ouvre un bloc d'instructions.

Quand on arrive sur la ligne du pour, il y une condition à vérifier.

Si la condition n'est pas vérifiée, on passe directement après le bloc d'instruction

Si la condition est vérifiée, on exécute les instructions du bloc.

Arrivé en fin de bloc, on remonte à la ligne du « tant que » et incrémente la variable i du pas (1 par défaut).

Ensuite on recommence :

Quand on arrive sur la ligne du pour, il y une condition à vérifier, etc.

++ Les 4 différents types de boucle

Boucle « Tant que »

```
Tant que condition
    Instructions
Fin tant que
```

Boucle « Répéter » (do … while)

```
Répéter
    Instructions
Tant que condition
```

Même logique que le tant que, mais celui-ci est à la fin.

Boucle avec compteur : pour i de 1 à n par pas de 1 (boucle for)

```
Pour i de 1 à N / Pas
    Instructions
Fin pour
```

Par défaut le Pas vaut 1 et n'est pas écrit.

Boucle sans fin

```
Boucle
    Instructions
Fin de boucle
```

Transformation d'une boucle for en boucle while

Le code précédent peut s'écrire ainsi :

```
Programme lesCarrés
    i=1
    tant que i <=10
        afficher (i, i*i)
        i=i+1
    Fin pour
Fin
```

Explications

On initialise i à 1 avant d'entrer dans la boucle.

La boucle tant que vérifie si i est <= 10.

Si c'est le cas, on entre dans la boucle. La dernière instruction de la boucle incrémente i.

On revient en début de boucle et on recommence le test.

Quand le test n'est plus vérifié, on sort de la boucle.

Transformation d'une boucle while en boucle for

La conversion des fahrenheits peut s'écrire ainsi :

```
Programme listeFahr
    fahr ← 0
    Tant que fahr < = 300
    Pour fahr de 0 à  300 par Pas de 20
        celsius ←  5 * (fahr – 32) / 9
        afficher (fahr, celsius)
        fahr ← fahr + 20
    Fin tant que
Fin
```

Explications

L'initialisation de fahr à 0 et l'incrémentation de 20 sont intégrées directement dans la boucle for.

On voit que le code est finalement plus court. Le principe est qu'il vaut mieux utiliser une boucle for quand on sait combien de fois on boucle (quand on connaît le début, la fin et le pas).

++ Les débranchements de boucle : débranchements structurés

Break

Le break permet de quitter la boucle.

Continue

Le continue permet de sauter à la fin de la boucle et de passer au suivant, mais sans quitter la boucle.

Return

Le return permet de quitter la fonction dans laquelle la boucle se trouve.

++ Le goto : débranchement non-structuré : INTERDIT !!!

Le GOTO

Le GOTO est une rupture de séquence qui permet de sauter le déroulement continu des instructions (la séquence) en allant directement à la borne précisée par le goto.

Le GOTO permet d'aller n'importe où : on n'utilisera jamais le GOTO.

CF : « GO TO statement considered harmful » de Dijkstra (1968).

Notion de rupture de séquence

Les tests et les boucles sont des ruptures de séquences : les instructions ne vont pas s'exécuter l'une après l'autre.

Le test fait sauter en avant.

La boucle fait sauter en arrière, et éventuellement en avant.

Le test et la boucle sont des GOTO structurés.

Traduction d'une boucle tant que avec un GOTO structuré

```
Programme listeFahr
    Fahr ← 0
    borne tantque
    si fahr > 300
        goto borne fintantque
    finsi
    Celsius ← 5 * (fahr – 32) / 9
    Afficher (fahr, celsius)
    Fahr ← fahr + 20
    goto borne tantque
    borne fintantque
Fin
```

++ Méthode d'analyse d'un algorithme avec boucle

- Dès qu'un traitement doit être répété, on fera appel à une boucle.
- La boucle de base est la boucle tant que. Toute répétition doit donc être traitée en première hypothèse par un tant que.
- Si la première occurrence du traitement est nécessairement effectuée, on aura intérêt à utiliser un « répéter ».
- Si on sait combien de répétitions on doit effectuer, on utilisera une boucle « pour ».

++ Vérification et simulation

Principe

Pour vérifier un algorithme, une méthode consiste à simuler son exécution et à regarder le contenu des variables après chaque instruction.

Mise en œuvre

Pour mettre en œuvre la simulation, il faut faire un tableau. Chaque colonne contient une variable. Chaque ligne correspond à une instruction. On remplit au fur et à mesure la valeur des variables dans le tableau.

Exemple

➢ ***Algorithme***

```
Programme listeFahr
    Fahr ← 0
    Tant que fahr < = 300
        Celsius ← 5 * (fahr – 32) / 9
        Afficher (fahr, celsius)
        Fahr ← fahr + 20
    Fin tant que
Fin
```

➢ ***Simulation***

	Fahr	Fahr <=300	Celsius
Fahr ← 0	0		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			-17,6
Fahr ← fahr + 20	20		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			-6,7
Fahr ← fahr + 20	40		

		Vrai	
Celsius ← 5 * (fahr – 32) / 9			4,4
	Etc.		
Fahr ← fahr + 20	280		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			137,8
Fahr ← fahr + 20	300		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			148,9
Fahr ← fahr + 20	320		
		faux	

Python de base

```
# Programme listeFahr - boucle while
fahr =0
while fahr <= 300:
    celsius = 5*(fahr-32)/9
    print("%3d fahr = %6.2f celsius" % (fahr, celsius))
    fahr =fahr + 20
```

```
# Programme listeFahr - boucle for
for fahr in range(0,301,20):
    celsius = 5*(fahr-32)/9
    print("%3d fahr = %6.2f celsius" % (fahr, celsius))
```

```
# Programme lesCarrés - boucle for
for i in range(1, 11):
    print("%2d au carré = %3d" % (i,i*i))
```

```
# Programme lesCarrés - boucle while
i=0
while i <=10:
    print("%2d au carré = %3d" % (i,i*i))
    i+=1
```

6. ++ Fonction et procédure

Sixième exemple : la notion de fonction – paramètre en entrée

Première définition

Une fonction est algorithme qui **retourne une valeur**. On l'appelle « sortie standard » ou « retour le la fonction ».

Une fonction a donc une valeur.

L'algorithme a un nom et est **utilisable dans une expression**.

Exemple

Ecrire une **fonction** qui calcule la surface d'un rectangle et un programme qui permet de l'utiliser.

```
surface (larg , long) // retourne la surface : Réel
// E : larg : Réel // la largeur
// E : long : Réel // la longueur
Début
        return  larg * long;
Fin

Programme afficheSurface
        Lire (larg, long)
        Surf = surface(larg, long) ;
        Afficher (surf) ;
Fin
```

Septième exemple : fonction d'affichage – pas de sortie

Principes

Certaines fonctions n'auront pas de sortie. Toutefois, une fonction fait forcément quelque-chose et modifie donc forcément quelque chose.

Une fonction d'affichage n'a pas de sortie mais elle modifie l'écran.

Une fonction d'enregistrement dans la BD peut ne pas avoir de sortie mais elle modifie la BD. Etc.

Exemple

Fonction d'affichage de la table de multiplication d'un entier donné en entrée :

```
afficherTable (n) // pas de retour – affichage à l'écran
// E : n: entier // l'entier dont on affiche la table
Début
    Pour i de 1 à 10
        afficher n + '*' + i + ' = ' + n*i
    Fin pour
Fin
```

```
Programme afficheSurface
    Lire (larg, long)
    Surf = surface(larg, long) ;
    Afficher (surf) ;
Fin
```

Huitième exemple : la notion de procédure – paramètre en sortie

Première définition

Une procédure est algorithme qui **ne retourne pas de valeur**. Elle n'a **pas de sortie standard**.

L'algorithme a un nom et est **utilisable dans une instruction et pas dans une expression.**

exemple

Ecrire une **procédure** qui calcule la surface d'un rectangle et un programme qui permet de l'utiliser.

```
                        ↓   ↓   ↑
surfaceRectangle (larg, long, surf) // pas de retour
// E : larg : Reel // largeur
// E : larg : Reel // longeur
// S : surf : Reel // surface
Début
        surf = larg * long;
Fin

Programme afficheSurface
        Lire (larg, long)
        surface(larg, long, surf)
        Afficher (surf)
Fin
```

Une procédure peut avoir plusieurs paramètres en sortie.

Neuvième exemple : Fonction mixte

Première définition

Une fonction mixte possède une sortie standard et des paramètres en sortie.

exemple

Ecrire une **fonction** qui calcule la surface et le périmètre d'un rectangle et un programme qui permet de l'utiliser.

```
                                    ↓   ↓       ↑
perimetreEtSurfaceRectangle (larg, long, perimetre) // retourne la surface : Réel
// E : larg : Reel // largeur
// E : larg : Reel // longeur
// S : perimetre: Reel // perimetre
Début
        perimetre = larg * long;
        return  larg * long;
Fin
```

Dixième exemple : utilisation de structure, de tableau ou d'objet – paramètre en entrée et en sortie

Exemple avec une structure

Une structure est une variable contenant plusieurs informations.

On peut passer une structure en paramètre qui sera en entrée et en sortie.

Par exemple une structure « rectangle » pourrait contenir la largeur, la longueur et la surface.

On peut aussi passer des tableaux ou des objets en entrée et en sortie

```
                                    ↓↑
Procédure surfaceRectangle (rectangle) {
        retangle.surf = rectangle.larg * rectangle.long;
}
```

Généralisation

On peut aussi passer des **tableaux** ou des **objets** en entrée et en sortie

```
                                ↓↑
Procédure triTableau (tableau) {
        // code de l'algorithme de tri
}
```

++ Précisions sur les fonctions

En-tête de la fonction – paramètres formels

```
surfaceRectangle (larg , long)  // Réel,  surface d'un rectangle
```

L'en-tête de la fonction commence par le nom de la fonction.

Puis les variables utilisées par l'algorithme de la fonction entre parenthèses (« larg , long »). On appelle ces variables : « **paramètres formels** ».

Au bout, de l'entête, un commentaire pour préciser le type et la signification du résultat renvoyé.

La sortie standard

La valeur renvoyée par la fonction peut être appelé : sortie standard.

Paramètres formels de la fonction – variables en entrée

Les variables de l'en-tête de la fonction sont appelés : **paramètres formels**.

Dans cet exemple, ces paramètres sont passés en entrée.

Une **variable en entrée** est une variable dont on utilise la valeur pour le calcul de l'algorithme mais dont **la valeur ne sera pas modifiée.**

Commentaires de l'en-tête

```
// E : larg : réel // la largeur
// E : long : réel // la longueur
```

Le commentaire de l'en-tête consiste à préciser le mode de passage (E), le type et la signification des variables de l'en-tête.

Corps de la fonction

```
Début
        return  larg * long;
Fin
```

Le corps de la fonction décrit l'algorithme de la fonction.

Les instructions sont écrites entre les mots-clés « Début » et « Fin ».

Le mot-clé « Début » est facultatif : en général, on ne le mettra que si l'on met des commentaires.

L'instruction return permet de renvoyer le résultat de la fonction sur la sortie standard.

Débranchement de fonction : le return

```
        return  larg * long;
```

L'instruction return permet de renvoyer le résultat de la fonction sur la sortie standard.

Cette instruction permet aussi de quitter la fonction à tout moment.

Version compacte

```
surfaceRectangle (larg , long) {
        return  larg * long;
```

```
}
```

On ne commente pas les paramètres : tout est évident par le nom.

On utilise des accolades à la place de Début et Fin.

Utilisation d'une fonction : les paramètres d'appels de la fonction – passage par valeur

```
surf = surface(larg, long) ;
```

Quand un programme fait appel à une fonction, les paramètres passés à la fonction sont appelés paramètres d'appel.

Ces paramètres sont des expressions qui sont évaluées. Il peuvent donc être : des constantes littérales (3, 'A', «Bonjour »), des noms de variables ou des expressions. (Quand on a des variables de type complexe, ce sont en général des adresses qui sont passées en paramètre.)

C'est la valeur de l'évaluation qui est passée en paramètre à la fonction (**passage par valeur**).

Cette valeur est affectée à la variable du paramètre formel correspondant.

La correspondance entre les paramètres d'appels et les paramètres formels se fait par l'ordre des paramètres : le premier paramètre formel prend la valeur du premier paramètre d'appel, et ainsi de suite avec les suivants.

++ Précisions sur les procédures : paramètres en sortie

Entête et corps

Une procédure est constituée, comme une fonction, d'un en-tête et d'un corps.

En-tête de la procédure – paramètres formels

```
                        ↓   ↓   ↑
surfaceRectangle (larg, long, surf) // pas de retour
```

L'en-tête de la procédure commence par le nom de la procédure (comme une fonction).

Puis les variables utilisées par l'algorithme de la fonction entre parenthèses (« larg , long »). On appelle ces variables : « **paramètres formels** ».

A la différence d'une fonction, une procédure a des paramètres en sortie dans son en-tête.

Il faut donc préciser le mode de passage des paramètres : on utilisera les flèches.

Au bout, de l'entête, un commentaire pour préciser qu'il n'y a pas de retour

Une procédure peut avoir plusieurs paramètres en sortie.

Débranchement de procédure : le return

L'instruction return permet de quitter la procédure à tout moment.

Comme il n'y a pas de sortie standard, le return n'est jamais accompagné d'une expression.

Ecriture simplifiée

Dans l'algorithme traité, le type des paramètres est évident, leur nom suffit à dire ce qu'ils sont, on peut donc éviter les commentaires :

```
                        ↓   ↓   ↑
surfaceRectangle (larg, long, surf) {  // pas de retour
        surf = larg * long;
}
```

Paramètres en sortie d'une procédure : passage par référence (ou par adresse)

Pour les paramètres en entrée, le principe est le même pour une procédure et pour une fonction pour les paramètres en entrée.

Par contre, pour les paramètres en sortie, le paramètre d'appel ne peut pas être une expression : c'est obligatoirement une variable. On parle alors de **passage par référence** ou par adresse.

Usage concret

Beaucoup de langages ne permettent les paramètres en sortie que sur certain type de variables : principalement les tableaux, les structures et les objets.

C'est le cas en python.

Boite noire et tests unitaires

Boite noire

Une fonction a vocation à devenir une « boite noire », c'est-à-dire un outil qui fonctionne, qu'on peut utiliser d'un point de vue externe (ou fonctionnel).

Etant donné que la fonction fonctionne, on peut se désintéresser de comment elle est faite. On ne s'intéresse plus à l'intérieur : c'est l'idée de la « boite noire ».

Tests unitaires

- ***1 fichier de tests unitaire par fonction***

  Les tests unitaires sont tous les tests qu'on va effectuer sur la fonction pour vérifier qu'elle fonctionne bien quelles que soit l'usage qu'on en a.

  Un programme de tests unitaires est un programme qui teste une fonction. On peut concevoir un programme de test par fonction.

- ***1 fichier de tests unitaire pour plusieurs fonctions réunies dans un même fichier***

  On peut aussi regrouper les fonctions qui « marchent ensemble » (l'une appelant l'autre par exemple, ou les unes et les autres travaillant sur les mêmes données) dans un même fichier et faire un fichier de tests unitaires pour toutes ces fonctions.

  Ainsi si on modifie une fonction ou si on en crée une nouvelle, on part du programme de tests unitaires existant pour ajouter de nouveaux tests.

Python de base

Fonction avec 2 paramètres réels en entrée

```
def surface(larg,long): # Réel
# E : larg : Réel // la largeur
# E : long : Réel // la longueur
    return  larg * long

# programme
larg=float(input('entrez la largeur : '))
long=float(input('entrez la longueur : '))

surf = surface(larg, long)
print(surf)
```

Fonction avec une structure en entrée-sortie

```
def surfaceStruct(rectangle): # pas de retour
# E : structure rectangle (longueur et largeur)
# S : structure rectange (surface et perimetre)
    rectangle['surface']=rectangle['largeur'] * rectangle['longueur']
    rectangle[‘perimetre’]=2*(rectangle['largeur'] + rectangle['longueur'])

# programme
larg=float(input('entrez la largeur : '))
```

```
long=float(input('entrez la longueur : '))

rect={'largeur':larg,'longueur':long}
print(rect)
surfaceStruct(rect)
print(rect)
```

7 - Les tableaux

1. Tableau à une dimension

Le tableau

Un tableau est un nouveau type de variable qui permet de regrouper dans une seule variable plusieurs valeurs de même type. L'idée est de pouvoir en avoir beaucoup (des milliers, des millions, etc.). On peut aussi avoir des petits tableaux.

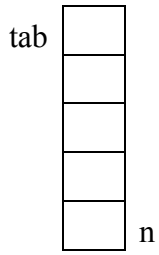
Une variable de type tableau s'appelle aussi un tableau. On les appelle classiquement « tab ».

Le tableau est souvent associé à sa taille (le nombre d'éléments qu'on peut mettre dedans). On l'appelle classiquement « n ».

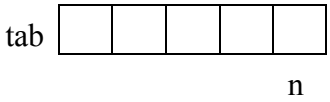
Représentation d'un tableau tab de taille n

On représente un tableau comme une suite de variables simples collées les unes aux autres.

A la verticale :



A l'horizontal :

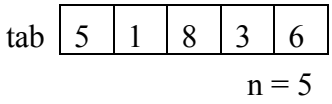


Création d'un tableau

Pour créer un tableau, on peut écrire :

tab=[] // création d'un tableau vide

tab = [5, 1, 8, 3, 6] // création d'un tableau avec 5 éléments, 5 entiers en l'occurrence.



Préciser le type de valeur contenu dans le tableau

On peut préciser le type de valeurs contenues dans le tableau en écrivant par exemple :

Entier[] tab=[] // création d'un tableau vide d'entiers. A noter l'écriture Type[]

Entier[] tab = [5, 1, 8, 3, 6] // création d'un tableau avec 5 éléments, 5 entiers en l'occurrence.

Ce sera surtout utile quand on aura des tableaux de structures (cf. chapitre sur les structures).

Accès aux éléments d'un tableau

Pour accéder aux éléments d'un tableau on écrit :

- tab[0] : pour accéder au 1er élément.
- tab[1] : pour accéder au 2 ème élément.
- tab[i] : pour accéder au i ème élément.

tab	5	1	8	3	6
	tab[0]	tab[1]	tab[2]	tab[3]	tab[4]

Manipulation des éléments d'un tableau

tab[i] est une variable comme une autre. On peut lui appliquer tous les traitements qu'on applique aux variables de son type :

```
lire (tab[0])
tab[0] = tab[0]*2
afficher tab[0]
```

Accès au nombre d'éléments d'un tableau : len(tab) ou tab.len

Le nombre d'éléments d'un tableau est constitutif du tableau : il permet par exemple d'accéder à tous les éléments du tableau.

On peut écrire **len(tab)** ou **tab.len** pour récupérer les éléments d'un tableau.

Manipuler tous les éléments d'un tableau : la boucle for

Pour manipuler tous les éléments d'un tableau, on utilise une boucle for.

Ainsi pour afficher les éléments d'un tableau, on écrira

```
pour i de 0 à tab.len -1
    Afficher(tab[i])
fin pour
```

Tableau à taille fixe et tableau à taille variable

- ***Tableau à taille fixe***

A l'origine, les tableaux sont des tableaux à taille fixe. On considère que le nombre d'éléments ne va pas changer ou bien qu'on gérera nous même le nombre d'éléments à un moment donné dans un tableau pouvant contenir des cases vides à la fin du tableau. Dans ce cas, la fonction len() renvoie la taille fixe qui ne correspond pas forcément aux nombres d'éléments du tableau.

Cet usage est important à connaître mais compliqué à mettre en œuvre et il ne correspond plus aux principaux services offerts par les langages.

Pour préciser que le tableau est à taille fixe dans les algorithmes on écrit à la création :

tab[5]=[] // création d'un tableau à taille fixe de 5 éléments, sans initialisation.

tab[5] = [5, 1, 8, 3, 6] // création d'un tableau à taille fixe de 5 éléments, avec initialisation.

- ***Tableau à taille variable***

Dans un tableau à taille variable, on peut ajouter et supprimer des éléments dans le tableau. Il n'y a pas de cases vides. La fonction len() renvoie toujours le nombre d'éléments du tableau.

- ***Usage algorithmique***

  On travaille en algorithmique avec des tableaux à taille variable. On peut toutefois les considérer selon les cas comme des tableaux à taille fixe.

Insertion et suppression d'un élément dans un tableau à taille variable

- ***Insertion à la fin : append()***

  Pour ajouter une valeur à la fin du tableau, on écrit

  tab.append(valeur)

  Ca augmente de 1 la taille du tableau.

- ***Suppression à la fin : pop()***

  Pour supprimer une valeur à la fin du tableau (la dernière), on écrit :

  valeur=tab.pop()

  Ca diminue de 1 la taille du tableau.

2. Tableau à 2 dimensions : matrice

La matrice

Une matrice est un tableau à deux dimensions.

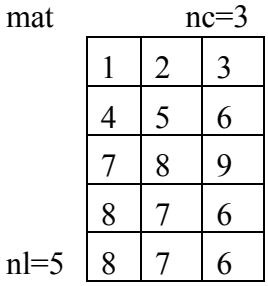
On peut se le représenter comme un tableau de tableaux.

On l'appelle classiquement « mat».

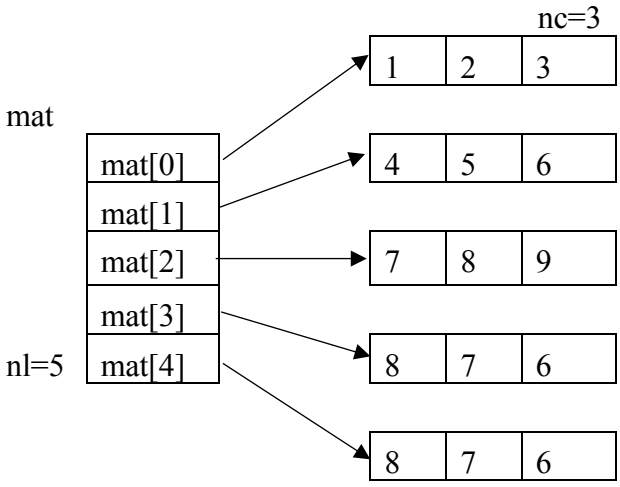
La matrice est souvent associé à son nombre de lignes : « nl » et à son nombre de colonnes « nc ».

Représentation d'une matrice mat de taille nl, nc

On représente un tableau comme un damier.



On peut aussi le représenter plus clairement comme un tableau de tableaux :



Création d'une matrice

Pour créer un tableau, on peut écrire :

mat=[] // création d'une matrice vide

mat = [[1,2,3], [4,5,6], [7,8,9], [8,7,6], [5,4,3]] // création d'une matrice de 5 lignes avec un tableau de 3 éléments par ligne.

Accès aux éléments d'une matrice

Pour accéder aux éléments d'un tableau on écrit :

- mat[0] [0] : pour accéder à l'élément en première ligne et première colonne : « 1 »
- mat[2][1] : pour accéder à l'élément en troisième ligne et deuxième colonne : « 1 »
- mat[i][j] : pour accéder à l'élément en i, j

mat[i][j] est une variable comme une autre. On peut lui appliquer tous les traitements qu'on applique aux variables de son type.

```
lire ( mat[0][0] )
tab[0] = mat[0][0]*2
afficher( mat[0][0] )
```

Accès au nombres de lignes d'une matrice: len(mat) ou mat.len

Une matrice est un tableau de tableaux. Pour avoir son nombre de lignes, il suffit d'utiliser la fonction len(à.

On peut écrire **len(mat)** ou **mat.len** pour récupérer les éléments d'un tableau.

Accès au nombres de colonnes d'une matrice: len(mat[0]) ou mat[0].len

Une matrice est un tableau de tableaux. Pour avoir son nombre de lignes, il suffit d'utiliser la fonction len(à.

On peut écrire **len(mat)** ou **mat.len** pour récupérer les éléments d'un tableau.

Manipuler tous les éléments d'une matrice: 2 boucle for imbriquées

Pour manipuler tous les éléments d'un tableau, on utilise une boucle for.

Ainsi pour afficher les éléments d'un tableau, on écrira

```
pour i de 0 à mat.len -1
    pour j de 0 à len( mat([i])-1
        Afficher(mat[i][j])
    Finpour
    Afficher (passage à la ligne)
fin pour
```

3. Méthodes de recherche

Recherche dans un tableau non trié

Pour rechercher un élément dans un tableau non trié, il faut parcourir tout le tableau.

Si on a N éléments dans le tableau, la recherche se fera en au maximum N tests, N/2 tests en moyenne.

Recherche dans un tableau trié : la recherche dichotomique

Pour rechercher un élément dans un tableau trié, on fait une recherche dichotomique.

Si on a N éléments dans le tableau, la recherche se fera en au maximum log2(N), soit environ 20 pour N = un million.

4. Méthodes de tri

Tri à bulles

Le principe est de réorganiser (inverser) les couples non classés tant qu'il en existe.

Tri par extraction (élémentaire, heapsort)

L'opération de base consiste à rechercher l'extremum dans la partie non triée et à le permuter avec l'élément frontière, puis à déplacer la frontière d'une position.

Tri par insertion

L'opération de base consiste à prendre l'élément frontière dans la partie non triée et à l'insérer à sa place dans la partie triée, puis à déplacer la frontière d'une position.

Autres méthodes

Il existe de nombreuses autres méthodes qui ne sont pas abordées ici.

Python de base

Tableau

```
# création d'un tableau de 5 éléments
tab=[5, 1, 8, 3, 6]
print(tab)
for i in range(0, len(tab)):
    print(tab[i])

#il faut un saut de ligne à la fin de bloc car on n'est pas dans une fonction
tab.append(10)
print(tab)
valeur=tab.pop()
print(tab)
```

Matrice

```
# création d'une matrice de 5 lignes avec un tableau de 3 éléments par ligne
```

```
mat=[[1,2,3], [4,5,6], [7,8,9], [8,7,6], [5,4,3]]
def printMat(mat): # function d'affichage d'une matrice
      nc=len(mat)
      for i in range(0, nc):
            nl=len(mat[i])
            for j in range(0, nl):
                print("%3d" %(mat[i][j]), end='')
                if(j!=nl-1): print(', ', end='')
            print();

printMat(mat)
mat[0].append(999)
printMat(mat)
mat[1].pop()
printMat(mat)
```

8 - Les structures

Structure et type structuré

Définition

Une structure est une variable contenant plusieurs informations de types différents. Chaque information est accessible par son nom, en plus du nom de la variable.

Chaque partie s'appelle « champ » ou « attribut ».

On parle de structure pour la variable comme pour le type. Pour le type, on parle aussi de « type structuré ».

Représentation d'une structure

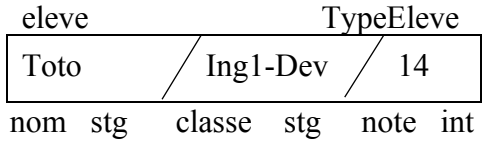
On représente une structure à l'horizontal, comme une suite de variables simples, collées les unes aux autres, mais séparées par un trait oblique.

- ***Exemple***

On représente une variable elève de type TypeEleve (en programmation objet, on écrira de type Eleve, la distinction entre la variable et le type étant qu'une variable commence par une minuscule et un type par une majuscule).

La structure TypeEleve a 3 champs : nom, classe et note.

Un élève pourrait être : « Toto », « Ing1-Dev », 14



- « eleve » est du type TypeEleve.
- « nom » est de type string (stg)
- « classe » est de type string
- « note » est de type int

Analogies

- ***Structure en tant que variable***

  C'est une **ligne** d'un **tableau de données lignes colonnes.**

Tableau d'élèves :

Le type :	**nom**	**classe**	**note**
variable 1 :	Toto	Ing1-Dev	14
variable 2 :	Tata	Ing1-Dev	13
…			

C'est un **tuple** (une ligne) d'une table de **base de données**.

C'est un **objet** de la programmation orientée objet (**P.O.O.**).

- ***Structure en tant que type***

  C'est **la première ligne d'un tableau lignes-colonnes**, ligne contenant les intitulés de chaque colonne.

  C'est la **définition de la table** en base de données.

  C'est une **classe** de la P.O.O.

Définition du type : type Struct ou Class

On est obligé de définir clairement le type, puisqu'il contient des champs avec des noms choisis comme on veut.

On utilise le **mot clé Struct** puis on donne le nom du type, puis on liste les attributs.

Ici, on crée le type TypeEleve qu'on pourra utiliser comme n'importe quel type.

```
Struct TypeEleve
    nom : chaîne
    prenom : chaîne
    note : entier
fin
```

On peut aussi utiliser le **mot clé Class :**

En général, on donne au nom de la classe directement de nom de l'ensemble concerné, sans passer par le préfixe « Type ».

```
Class Eleve
    nom : chaîne
    prenom : chaîne
    note : entier
fin
```

Notion de référence

```
e3 = e1                    // création d'une 2ème référence sur l'élève e1.
```

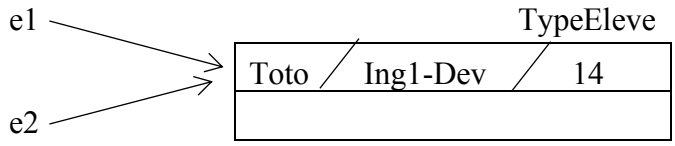
En général, la variable de type structure correspond à une référence et pas la structure elle-même.

Une référence, c'est une information très simple qui permet d'accéder à la structure elle-même. Concrètement, c'est une adresse.

Ainsi quand on écrire e3=e1, on se donne une nouvelle référence vers la même structure concrète.

C'est la même chose avec les tableaux.

On peut représenter les choses ainsi :



Si on écrit :

e1.note = 15, c'est la structure référée qui est modifiée et donc ça concerne aussi e2.

Notion de pile et de tas

- ***Variable avec ou sans référence***

  Certaines variables fonctionnent avec une référence : les structures et les tableaux.

  Les entiers, réels, booléens, caractères, chaines de caractère n'ont pas de référence.

- ***Pile et tas***

  Les variables sans référence sont stockées dans une pile de variables.

  La références des variables avec référence est aussi stockée dans la pile.

  Ce qui est référencé par les variable avec référence (les éléments du tableau ou les attributs de la structure) est stocké dans ce qu'on appelle « le tas ».

- ***Fonction***

  Les variables qui interviennent dans une fonction sont toutes stockées (empilées) dans une pile de variables appelées « la pile ».

  Elles seront supprimées de la pile à la sortie de la fonction (elles sont dépilées de la pile).

  Quand on passe un tableau ou une structure en paramètre à une fonction, c'est en réalité la référence en tant que valeur qu'on passe en paramètre. Cette référence permettra de modifier le contenu du tableau ou de la structure qui se trouve dans le tas.

  Quand on crée un tableau ou une structure dans une fonction, la référence va dans la pile et sera dépilé (supprimé) à la sortie de la fonction. Le contenu du tableau ou de la structure va dans le tas et sera conservée après la sortie de la fonction. Toutefois, il faut retourner une référence vers ce qui est conservé pour ne pas le perdre.

Usages

- ***Exemple traité***

  On veut définir un type de donnée correspondant à un élève. Un élève a un nom, un prénom et une note.

- ***Définition du type***

```
Struct TypeEleve
      nom : chaîne
      prénom : chaîne
      note : entier
fin
```

- ***Déclaration de variables***

```
e1, e2, e3 : TypeEleve
```

ou

```
TypeEleve : e1, e2, e3
```

- ***Utilisation des variables***

```
e1.nom = « toto »
e1.prenom = « olivier »
e1.note = 15
e2.note = e1.note      // affectation d'un champs
```

Exemple d'algorithme

- ***Fonction « newEleve » qui crée un nouvel élève avec ses valeurs et fonction d'affichage***

```
newEleve(nom, classe, note) : Eleve
// la fonction reçoit nom, classe et note en paramètre
// la fonction renvoie le nouvel élève c
      TypeEleve : eleve
      eleve.nom = nom
      eleve.classe = classe
      eleve.note = note
      return eleve
fin

printEleve(eleve) : rien
// la fonction reçoit un élève en paramètre et affiche son contenu
// la fonction ne renvoie rien
      print(eleve.nom)
      print(eleve.classe)
      print(eleve.note)
fin

// usage de la fonction dans un programme
Programme test :
      TypeEleve : eleve
      eleve=newEleve(« toto », « IPI », 15)
      print(eleve)
```

- ***A noter que :***
  - On crée une structure dans la fonction newEleve : la référence va dans la pile, le contenu de la structure dans le tas.
  - On retourne la valeur de la référence qui permettra d'accéder au contenu dans le tas.

- ***Fonction « setNote(note) » qui donne une note à un éléve et « getNote() » qui récupère la note d'un élève :***

```
setNote(eleve, nouvelleNote ) // E : nouvelleNote // S : eleve
    eleve.note = nouvelleNote
fin

getNote(eleve) // E : eleve  // S : la note de l'élève
    return eleve.note
fin

// usage de la fonction dans un programme
Programme test :
      TypeEleve : eleve
      eleve=newEleve(« toto », « IPI », 15)
      print(eleve)
      setNote(eleve, 18)
      print(eleve)
      print( getNote(eleve) )
```

- ***A noter que :***
  - Toute cette écriture est très proche d'une écriture de P.O.O.

Tableau de structures

Principes

Un tableau peut contenir des structures.

Définition des types

- ***Exemple traité***

  On veut définir un tableau contenant des élèves ayant un nom, un prénom et une note.

- ***Définition du type de l'élève***

```
Struct TypeEleve
    nom : chaîne
    prénom : chaîne
    note : entier
fin
```

En algorithmique, la définition du type de la structure est obligatoire pour clarifier les choses.

- ***Définition du type du tableau d'élèves***

```
TypeEleve []
```

En précisant le type devant les [], on précise qu'il s'agit d'un tableau de TypeEleve.

Cette définition peut s'avérer superflue.

Déclaration (création) d'un tableau

Pour créer un tableau quelconque, on écrit :

```
tab=[] // création d'un tableau vide
```

Pour préciser le type du contenu du tableau on écrit :

```
TypeEleve []  tabEleves = [] ;
```

En algorithmique, la précision du type du tableau n'est pas obligatoire. Le contexte ou le nom du tableau peuvent suffire à clarifier les choses.

Ainsi la déclaration d'un tableau d'élèves peut s'écire :

```
tabEleves = [] ;
```

On comprend très bien qu'il s'agit d'un tableau d'élèves.

Exemple de manipulation

```
TypeEleve : e1, e2
e1.nom= »toto » ; e1.note=15
tabEleves[0]=e1

tabEleves[1].nom=”tata”
tabEleves[1].note=14
```

Structures complexes

Principes

Chaque champ d'une structure peut contenir une variable de type simple mais aussi un tableau ou une structure. On parle alors de structure complexe.

Tableau dans la structure

Au lieu d'avoir 1 note par élève, on a un tableau de notes.

- ***Déclaration***

```
Struct TypeEleve
      nom : chaîne
      prenom : chaîne
      Entier[]  tabNotes = []   //  à noter l'écriture Type[]
fin
```

- ***Usage***

```
TypeEleve : e1
e1.nom= »toto » ;
e1.tabNotes=[12,15,9]

e.tatNotes.append(18) // ajoute un 18 au tableau
```

Structure dans la structure

Au lieu d'avoir 1 note par élève, on a 1 note et la matière correspondante.

- ***Déclaration***

```
Struct TypeMatiere
      note: entier
      matiere: chaine
fin
Struct TypeEleve
      nom : chaîne
      prenom : chaîne
      noteMatiere : TypeMatiere
fin
```

- ***Usage***

```
TypeEleve : e1, e2
e1.nom= »toto » ;
e.noteMatiere.note=15
e.noteMatiere.matiere=”algo”

e2.nom=”tata”
TypeNoteMatiere : nm;
nm.note=18;
nm.matiere=”algo”
e2.noteMatiere=nm
```

Tableau de structures dans la structure

Au lieu d'avoir 1 note par élève, on a plusieurs notes et la matière correspondante.

- ***Déclaration***

```
Struct TypeMatiere
      matiere: chaine
      note: entier
fin
Struct TypeEleve
      nom : chaîne
      prenom : chaîne
      TypeMatiere[] tabNotes = []  // à noter l'écriture Type[]
fin
```

- ***Usage, par exemple***

```
TypeEleve : e1
TypeNoteMatiere : nm;

e1.nom= »toto » ;
nm.note=18;
nm.matiere=”algo”
e1.tabNotes[0]=nm

nm.note=15;  // ici il faudra faire attention à bien gérer les références.
nm.matière=”math”
e1.tabNotes[1]=nm

afficher( e1.tablNotes[1].note )
```

Python de base

Structure

```
e1={}; # création de e1
e1['nom']='toto'
e1['prenom']='olivier'
e1['note']=15
print("e1 : ",e1)

e2={}; # # création de e2
e2['note']=e1['note']
print("e2 : ", e2) # e2 n'a qu'un attribut

e3=e1 # on a 2 référence vers le même élève
print("e3 : ",e3)
e3['note']=20 # ça modifie aussi e1
print("e3 : ",e3)
print("e1 : ",e1)

# fonction pour créer un éleve : plus pratique
def setNote(eleve, note):
    eleve['note']=note

setNote(e1,15) # ça modifie aussi e3
print("e3 : ",e3)
print("e1 : ",e1)

setNote(e2,0)
print("e2 : ", e2)

# fonction pour créer un éleve : plus pratique
def newEleve(nom, prenom, note):
    eleve={}; # creation d'un eleve
    eleve['nom']=nom
    eleve['prenom']=prenom
    eleve['note']=note
    return eleve;

e1=newEleve('toto', 'olivier', 15)
print("e1 : ",e1)

# fonction pour créer un éleve à partir d'un autre
def copieEleve(eleve):
    unEleve={}; # creation d'un eleve
    unEleve['nom']=eleve['nom']
    unEleve['prenom']=eleve['prenom']
    unEleve['note']=eleve['note']
    return unEleve;
```

```
e3=copieEleve(e1)
print("e3 : ",e3)
setNote(e1,20) # ça ne modifie pas e1
print("e3 : ",e3)
print("e1 : ",e1)
```

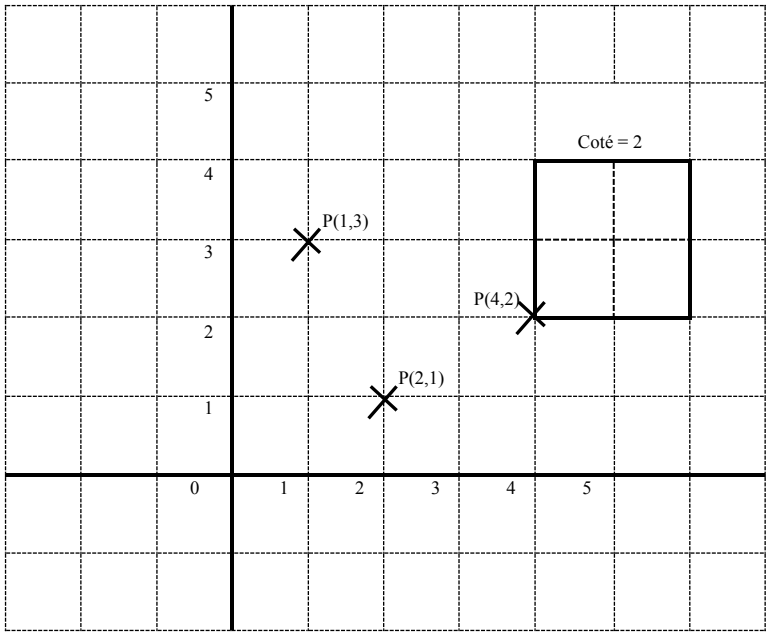
Exercices

Exercice 1 – des points et des carrés

Un point est caractérisé par ses coordonnées x et y. Un carré à base horizontale est défini par les coordonnées de son point en bas à gauche et par son côté. Les coordonnées et le côté sont des réels.

On travaille sur un ensemble de carrés.

- Définir les structures qui permettent de gérer ce problème. Créer le carré du schéma ci-dessous.
- Ecrire une procédure qui calcule la surface d'un carré.
- Ecrire une procédure qui calcule les coordonnées du point en haut à droite d'un carré.
- Ecrire une procédure qui détermine si un carré est inclus dans un autre carré.
- Ecrire une procédure qui détermine quel est le plus grand nombre de carrés inclus dans un autre carré.



Exercice 2 – élève et classe d'élèves

Un élève est caractérisé par son nom, son prénom, sa date de naissance. Il y a 3 matières d'informatique : algo, C++ et SQL. Chaque matière donne lieu à 2 QCM et à 1 examen. Les QCM comptent pour 25%. L'examen pour 50%. On connaît les dates d'examen et de QCM. Chaque élève porte toutes les informations le concernant. On connaît les notes pour chaque examen, la note finale pour la matière et la moyenne des 3 matières.

- Définir la ou les structures de données permettant de gérer un élève.
- Définir la structure de donnée permettant de gérer une classe. On enregistre aussi la moyenne générale de la classe dans la structure de la classe.
- Ecrire une procédure ou une fonction qui permet de mettre à jour la note finale de chaque élève pour chaque matière et sa moyenne.
- Ecrire une procédure ou une fonction qui permet de calculer la moyenne générale de la classe.
- Ecrire une procédure ou une fonction qui affiche la liste des élèves avec leur moyenne générale triée par notes décroissantes.

Exercice 3 : tableau d'utilisateurs

On veut gérer des utilisateurs avec les caractéristiques suivantes :

- Prénom et NOM (dans un seul champ),
- mail,
- motDePasse,
- annee de naissance

En vous appuyant sur les exemples du cours :

1) Créez une fonction newUser permettant de créer un utilisateur.

```
# creation d'un utilisateur
newUser( parametres à déterminer);
```

2) Créez une fonction qui permette d'afficher un utilisateur.

```
# affichage d'un utilisateur
printUnUser( parametres à déterminer);
```

3) Utilisez les deux fonctions pour créer un utilisateur et l'afficher.
4) A la place de l'année de naissance, on veut afficher l'âge. Adapter le programme. On se dote d'une fonction **calculerAge**(parametres à déterminer);

   Pour calculer l'âge, on fera une simple soustraction entre l'année en cours et l'année de naissance.

   En python, pour récupérer l'année du jour, on peut écrire :

```
from datetime import date

year=date.today().year
```

5) Créez une fonction qui permette de créer un tableau d'utilisateurs, vide dans un premier temps.

```
# création d'un tableau d'utilisateurs
créerLesUsers( parametres à déterminer );
```

6) Créez une fonction qui permette d'ajouter un utilisateur dans le tableau précédemment créé.

```
# création d'un tableau d'utilisateurs
ajouterDansLesUsers( parametres à déterminer );
```

7) Créez une fonction qui affiche les utilisateurs.

```
# affichage du tableau d'utitlisateur
printLesUsers ( parametres à déterminer );
```

8) Utilisez les fonctions pour créer un tableau de 5 utilisateurs et affichez-le.

9) Créez une fonction qui permette de trier par nom un tableau d'utilisateurs et testez la fonction.

```
# initialisation d'un tableau d'utilisateurs
triParNom( parametres à déterminer );
```

On utilisera un tri à bulles. Rappel de la méthode de résolution la plus simple : on parcourt le tableau. Si une valeur est plus grande que la suivante, on les inverse les deux valeurs pour que la plus grande passe en dessous. On répète l'opération autant de fois qu'il y d'éléments dans le tableau.

On peut ensuite réfléchir à optimiser l'algorithme.

10) Créez une fonction qui permette de trier par âge un tableau d'utilisateurs et testez la fonction.

```
# initialisation d'un tableau d'utilisateurs
triParAge( parametres à déterminer );
```

11) Créez une fonction qui permette de trier par n'importe quel critère un tableau d'utilisateurs et testez la fonction.

```
# initialisation d'un tableau d'utilisateurs
triParCritere( parametres à déterminer );
```

Exercice 4 : Select SQL

- Ecrire un algorithme qui recherche les employés qui gagne plus de 2000 et qui ont été embauché après 2016.
- On part d'une structure employe (NE, nom, prenom, fonction, salaire, dateEmbauche, ND)
- Avec NE, numéro de l'employé et ND numéro de son département.
- On s'appuiera sur le code de l'exercice précédent (tableau d'utilisateurs).

Exercice 5 : Jointure SQL

- Ecrire un algorithme qui recherche les employés parisiens qui gagne plus de 2000.
- On part d'une table d'employés avec une structure employe (NE, nom, prenom, fonction, salaire, dateEmbauche, ND)
- Avec NE, numéro de l'employé et ND numéro de son département.
- On a aussi une table de départements avec une structure departement(ND, nom, ville).
- On s'appuiera sur le code de l'exercice précédent (tableau d'utilisateurs).

9. Les chaînes des caractères

Présentation

Déclaration d'une chaîne de caractères

```
ch1=”bonjour” // le texte est mis entre guillemets ou entre apostrophes
ch2=”” // chaine vide
len( ch2) // vaut 7
len( ch1 ) // vaut 0
```

Précision du type

En algorithmique, on utilise des variables appelées ch, ch1, chaine, st, st1, str, string, etc.

Elles permettent de savoir que ce sont des chaines de caractères.

Si on veut préciser on utilise le type String

```
String ch1=’bonjour ‘
```

Consultation caractère par caractère : comme un tableau

On peut écrire :

```
ch1=”bonjour”
n=len (ch1)
print ( ch1[0] )
pour i de 0 à n-1 {
      print( ch [i] )
}
```

Algorithmique du traitement de chaînes

L'algorithmique du traitement des chaînes consiste à définir des **fonctions de bases** qui permettent de faire toutes sorte de manipulation sur le chaînes.

Toutefois, quand on utilise un langage, il faut regarder les fonctions déjà écrites qu'il met à disposition.

Affectation

```
ch1=ch2
```

A noter que selon les langages, il peut s'agir de référence ou pas. En python, il ne s'agit pas de référence.

Longueur

```
n=len( ch)
```

Comparaison

On peut comparer deux chaînes entre elles :

```
ch1 == ch2 // sera Vrai ou Faux
```

Ou bien

```
ch1 < ch2
```

Avec l'inégalité, on compare, la valeur ASCII de chaque caractère.

Ca correspond à l'ordre alphabétique.

Concaténation

Il faut pouvoir coller des chaînes les unes derrière les autres : c'est la concaténation.

```
ch1=ch1+' '+ch2
```

ou

```
ch1+=' '+ch2
```

A noter qu'en réalité, on ne modifie pas ch1 mais on le change complètement.

Extraction <=> substr ou substring

Il faut pour pouvoir extraire un morceau de chaine d'une autre chaine

```
ch2 = substr (ch1, 3, 2) ;
```

avec ch1 valant "bonjour", ch2 vaut « jo »

j est en position 3 en démarrant à 0 et on prend 2 caractères.

En python, on écrit ch1[3, 3+2] : on démarre en position 3, on va jusqu'à la position 3+2

Majuscules et minuscules

majus(ch1) : retourne ch1 en majuscules

minus(ch1) : retourne ch1 en minuscules

ASCII : ord() et chr()

ord('a') : retourne le code ASCII de 'a' : 97

chr(97) : retourne le caractère ASCII correspondant à 97 : 'a'

Ecriture formatée

La syntaxe python est une bonne syntaxe pour faire de l'écriture formatée qu'on retrouve dans d'autres langage issus du langage C :

```
st="la racine de %g vaut %.2f" % (x, racine)
```

Petites fonctions booléenne pratiques à l'occasion

Ces fonctions s'appliquent à une chaine ou un caractère et renvoie Vrai ou Faux selon le type de caractère trouvé :

FONCTION	**type caractère**	**Liste correspondante**
isAlpha (str)	lettres	A-Z, a-z
isUpper (str)	majuscules	A-Z
isLower (str)	minuscules	a-z
isDigit (str)	chiffres	0-9
isAlnum (str)	lettres et chiffres	isalpha et isdigit
isSpace (str)	espaces	' '
Etc.		

Les tableaux de chaînes de caractères

Principes et déclaration

Un tableau peut contenir des chaines de caractères.

On le déclare aini

```
Sring [] tabEleves = [] ;  //  à noter l'écriture Type[]
```

Il s'utilise comme un tableau. Chaque élément du tableau est une chaine et s'utilise comme telle.

Argc, argv : arguments en ligne de commande

Principes

Quand on exécute un programme en ligne de commande, on peut lui passer des paramètres et consulter les paramètres qu'on a passées.

Le tableau d'arguments argv

argv est un tableau qui contient en 0 le fichier du programme, en 1 et suivant les arguments passés.

Exemple python

```
import sys                                  # pour accéder à argv
nbArg=len(sys.argv)                         # nombre d'éléments de argv
print('Nombre d\'arguments :', nbArg-1)     # affiche du nombre d'arguments
for i in range(0,nbArg):                    # boucle
      print('arg',i,' : ' ,sys.argv[i])     # affichage des
import sys
```

Résultats

```
C:>python testArgv.py a, b
Nombre d'arguments : 3
arg 0  :  test.py
arg 1  :  a,
arg 2  :  b,
```

Tests Unitaires

On peut utiliser argv pour les tests unitaires.

Exercices

Palindrome

Ecrire une fonction qui détermine si un mot est un palindrome ou pas. Un palindrome est un mot qui se lit pareil à l'endroit et à l'envers.

Anagramme

Ecrire une fonction qui détermine si un texte est une anagramme d'un autre ou pas. Une anagramme est un texte constitué avec les même lettres qu'un autre. Par exemple : « argent » et « gérant » ou encore : « connaître » et « actionner ».

Avec des expressions : « La gravitation universelle » devient « Loi vitale régnant sur la vie » et aussi « Entreprise Monsanto » devient «poison très rémanent ». Plus : ici.

Nombre d'occurrences d'un mot dans un texte

Écrire une fonction qui calcule le nombre d'occurrences d'un mot ou d'une suite de mots dans un texte (combien de fois il apparaît).

Nombre d'occurrences de tous les mots dans un texte

Ecrire une fonction qui calcule le nombre d'occurrence de tous les mots dans un texte, les séparateurs de mot étant définis dans une chaine de caractères

Problème : Algorithme de César

Principes

- ***Cryptage, chiffrement, codage***

  Le **cryptage** ou **chiffrement** est un procédé pour rendre la compréhension d'un document impossible sans avoir la **clé de déchiffrement**.

  On parle aussi de codage à la place de cryptage ou chiffrement, mais le codage est une notion plus large.

- ***Algorithme de César***

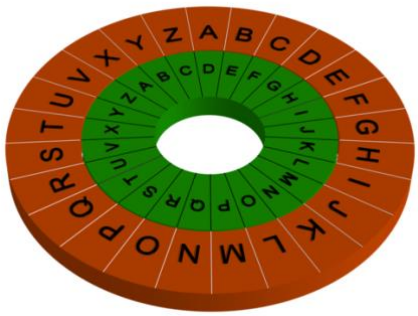
  César utilisait une technique simple pour coder ses messages. Cette technique est connue sous le nom de « algorithme de César ». C'est un algorithme de cryptage. C'est un algorithme simple.

  Le principe est de décaler l'alphabet dans le message crypté. **La valeur du décalage est la clé de cryptage**.

  Ainsi, si la clé vaut 3, A devient D, B devient E, C devient F, …, X devient A, Y devient B, Z devient C.

  « CESAR » est crypté par « FHVDU »

  On peut représenter les choses ainsi :



  L'alphabet rouge est celui du texte en clair (non crypté), l'alphabet vert est celui du texte crypté.

- ***Chiffrage***

  On peut faire correspondre chaque lettre de l'alphabet à un entier. On définit une bijection « f » d'un ensemble de lettres vers un ensemble d'entiers avec l'image de chaque lettre.

  f : {A, B, .., Y, Z} → {0, 1, …, 24, 25}

  A → 0, B→ 1, …, Y→24, Z→25

  Les lettres sont donc numérotées de 0 à 25 (A=0, B=1, …, Z=25).

  Le **chiffrement** consiste donc à :

- Remplacer chaque lettre du texte par l'entier lui correspondant.
- Puis à utiliser la clé pour « chiffrer » la valeur de départ. Dans le cas de l'algorithme de César, il suffit d'ajouter la valeur de la clé (la valeur du décalage).
- Il reste à remplacer le nouveau nombre par la lettre qui lui correspond. A vaut 0. 0 chiffré devient 3. 3 vaut C.

- ***Ensemble des clés***

  L'ensemble des clés, c'est l'ensemble des valeurs possibles pour la clé.

  Dans l'algorithme de César, l'ensemble des clés = {0, 1, …, 24, 25}

  Le cardinal de l'ensemble des clés donne la complexité de la clé.

  Dans l'algorithme de César, la complexité de la clé vaut 26, ce qui est très peu.

- ***Décodage – Déchiffrement : craquer le code !!!***

  Pour décoder un texte (pour le déchiffrer) il faut **connaître la valeur de la clé** (la valeur du décalage).

  On peut aussi **tester toutes les clés possibles** : si le cardinal de l'ensemble des clés est petit, c'est une opération faisable.

  De plus, dans le cas de l'algorithme de César, si on en connaît pas la valeur de d, on peut appliquer le principe suivant : **dans un texte, la lettre « e » est le plus souvent la plus représentée**. Ainsi, on peut trouver le décalage à partir du texte codé.

Exercices

- ***« A la main »***

1. **Coder** « bonjour » avec une **clé codage de 6.**
2. **Décoder** «olssv dvysk » avec une **clé de codage de 7**
3. **Chercher « manuellement » à décoder** le texte suivant : « tew wm jegmpi uyi gipe pi gsheki hi giwev »
4. Combien y a-t-il de clés possibles pour l'algorithme de César ?

- ***Algorithmes***

1. Ecrivez une fonction qui permette de coder des textes non codés à partir d'une clé. On l'appellera « coderMessage() ». A vous de déterminer les paramètres et le code.
2. Ecrivez une fonction qui permette de décoder des textes codés à partir d'une clé.
3. Ecrivez une fonction de tests unitaires qui permettra de mettre en œuvre ces deux fonctions sur 1 ou 2 exemples.

4. On cherche maintenant à « craquer » le code avec le principe du « e » le plus représenté. Ecrire une fonction frequenceDesLettres () qui renvoie un tableau avec la fréquence de chaque lettre de l'alphabet dans le message.
5. Testez la fonction en affichant le tableau de fréquences des lettres du message codé et le tableau de fréquences des lettres du message décodé.
6. Une fonction cleDeDecodage() qui renvoie la clé de décodage probable du message (le décalage à appliquer pour décoder).
7. Testez la fonction pour vérifier que le décodage fonctionne.

10. Les fichiers

Généralités

La mémoire

A la différence des variables qu'on a vues jusqu'à présent, le fichier est une mémoire durable et non plus volatile.

Il y a deux grandes dichotomies pour caractériser la mémoire :

1. une dichotomie concernant l'usage (qu'est-ce qu'on fait de la mémoire)
   - modifiable : on peut lire et écrire dessus autant que l'on veut
   - non modifiable : on ne peut écrire qu'une seule fois dessus, ensuite on ne peut plus que lire
2. une dichotomie concernant la durée de conservation
   - durable : la mémoire est "éternelle"
   - non durable : la mémoire est vouée à s'effacer après usage

Ca fait 3 types de mémoire :

Mémoires	modifiable	non modifiable
durable	disquette, disque dur	ROM
non durable	RAM	rien !!!

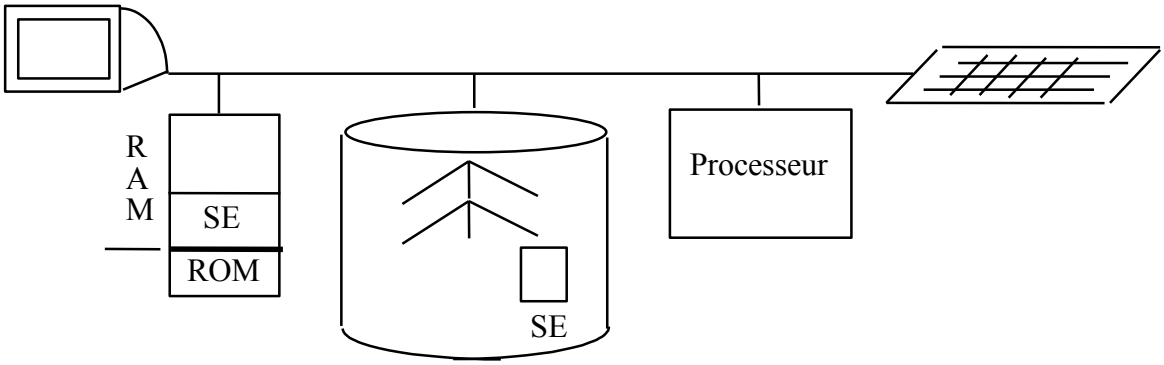
Les types de mémoires

On peut distinguer 4 types de mémoire dans les ordinateurs :

1. **les registres :** un registre est une mémoire ultra-rapide située directement dans le processeur. Un registre est une mémoire de très petite taille.
2. **la mémoire cache** : le cache permet de stocker des fichiers temporairement pour accélérer les traitement et éviter de relire les données sur le disque sur ou sur internet.
3. **la mémoire vive :** c'est la mémoire utilisée par les programme pendant leur exécution pour faire les cacul.
4. **le disque dur** : c'est une mémoire durable qui permet d'enregistrer les fichiers et de les récupérer plus tard.

Les registres et la mémoire vive sont des mémoires non durables. Le disque dur et le cache sont des mémoires durables
Les registres sont les mémoires les plus rapides. Le disque dur est la mémoire la plus lente.

Rappels d'architecture élémentaire des ordinateurs



Rappelons que quand l'ordinateur démarre :

- Le petit programme codé dans la ROM va chercher le fichier du logiciel système d'exploitation (Windows10, MacOS, linux, SE sur le schéma) sur le disque dur et le recopie sur la RAM et l'exécute.
- L'utilisateur communique alors avec le système d'exploitation.

Le système d'exploitation offre **trois grands types de fonctionnalité** :

- la gestion de fichier (dir, cd, grep, etc.),
- la gestion des périphériques (dont le réseau)
- le lancement des logiciels.

Les types de fichiers

Deux grandes dichotomies :

1. dichotomie concernant l'usage (qu'est-ce qu'on fait du fichier)
   - fichier document : c'est fichier que je produit avec word, excel, notepad, etc.
   - fichier logiciel : c'est un programme : word-2000
2. dichotomie concernant le type (le format)
   - fichier texte (ou fichier ASCII). Les fichiers "texte" contiennent des caractères ASCII : on peut donc les lire directement (commande type sous dos ou avec un éditeur).
   - fichier binaire (ou fichier typé). Les fichiers binaires sont interprétables via un logiciel (les .doc par word, les .xls par excel, etc.) ou par la machine (les .exe ou les .com).

Ca fait quatre types de fichiers :

Fichiers	**doc**	**prog**
binaire	projet.doc comptes.xls	word_2000 prog.exe
texte (ascii)	projet.txt	prog.bat

Organisation du rangement des fichiers

Sur les mémoires durables, les fichiers sont rangés dans des **répertoires** (ou directory, ou dossier). Un répertoire peut contenir des fichiers et des répertoires. L'imbrication des répertoires les uns dans les autres est appelée une arborescence. On parle de l'arborescence des répertoires.

Quand on travaille sur un fichier, il faut connaître son nom, mais aussi le nom du répertoire où il se trouve, ainsi que la branche complète des répertoires dans lequel il se trouve imbriqué jusqu'à ce qu'on appelle la racine.

Fichier et variable en programmation

Un fichier est une variable contenant un ensemble de variables de même type (entier, float, structure, char). Cet ensemble est fini, ordonné et de cardinalité variable.

Pour la programmation : à la différence des variables, le fichier contient des données qui sont en dehors du programme. Les fichiers sont stockés sur une mémoire durable (disque dur, disquette, etc.), alors que les variables qu'on a étudiées jusqu'à présent sont stockées dans la mémoire vive (la RAM) : elles ne sont pas durables, elles n'ont d'existence que pendant la durée du programme.

L'intérêt du fichier c'est que les données sont conservées de manière durable.

Comme toutes les variables, un fichier est caractérisé par son nom. Ce nom devra préciser la branche de l'arborescence des répertoires dans lequel se trouve le fichier.

Algorithmique des fichiers

Avant de présenter les fonctions C de traitement de fichier,

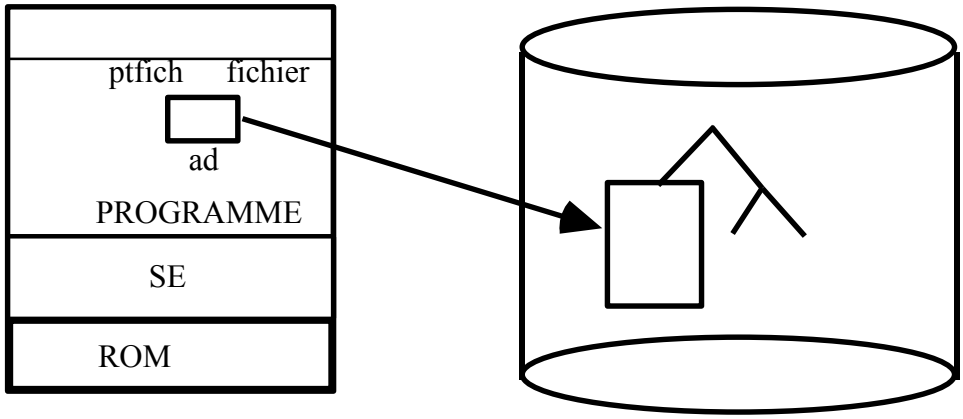
On va présenter le jeu minimal des fonctions qui permet de faire tous les traitements possibles sur les fichiers.

Le type fichier

Ce type est essentiellement un pointeur qui donne l'adresse d'un élément du fichier (d'abord l'adresse du premier élément du fichier.

Mais il va aussi contenir d'autres informations : ce sera en fait une structure dont l'un des champs contient l'adresse de l'élément courant du fichier.

Il faut bien comprendre que dans le programme, il y a une variable habituelle (une structure comme n'importe quelle structure) qui contient l'adresse d'un élément d'un fichier.



Créer un fichier

La fonction **creerFichier** permet de créer un fichier dont on donne le nom. Elle renvoie l'adresse du premier élément. Si le fichier existait déjà, l'ancien fichier est détruit.

```
pt_fichier = creerFichier (nom_du_fichier (in)) ;
```

Ouvrir un fichier pour le consulter

La fonction **ConsulterFichier** permet d'accéder en consultation au fichier dont on donne le nom. Elle renvoie l'adresse du premier élément, NULL si le fichier n'existe pas :

```
pt_fichier = consulterFichier (nom_du_fichier (in) ) ;
```

Ouvrir un fichier pour le modifier

La fonction **modifierFichier** permet d'accéder en consultation et en modification au fichier dont on donne le nom. Elle renvoie l'adresse du premier élément, NULL si le fichier n'existe pas :

```
pt_fichier = modifierFichier (nom_du_fichier (in)) ;
```

La différence entre ces deux fonctions consiste dans le fait que la première permet à d'autres de modifier le fichier, tandis que la seconde empêche toute modification du fichier par quelqu'un d'autre.

Lire un élément dans le fichier à partir du pointeur de fichier et passer au suivant

La fonction **lireFichier** lit dans le fichier l'élément pointé. Elle renvoie la valeur de l'élément lu. Le pointeur de fichier est modifié : il passe à l'élément suivant.

```
a = lireFichier (pt_fichier (inout) )
```

Ecrire un élément et passer au suivant

La fonction **ecrireFichier** écrit dans le fichier l'information à la position de l'élément pointé. Le pointeur de fichier est modifié : il passe à l'élément qui suit celui qu'on vient d'ajouter.

```
ecrireFichier (pt_fichier (inout), a (out) )
```

Se déplacer de n élément

La fonction **deplaceFichier** se déplace de n éléments dans le fichier, vers l'avant ou vers l'arrière (n positif ou négatif), à partir de la position de pt_fich.

```
deplaceFichier (pt_fichier out, n in)
```

Repérer la fin du fichier

La fonction booléenne **EOF** (pronocer "end of file" pour fin de fichier) répond vrai quand on pointe sur la fin du fichier, c'est-à-dire après le dernier élément (par exemple : après la création d'un fichier, EOF vaut vrai).

```
oui_non = EOF (pt_fichier in)
```

Fermeture d'un fichier

La fonction **fermeFichier** ferme le fichier. Fermer un fichier permet de signaler qu'on n'est plus en train d'utiliser le fichier et donc que d'autres peuvent l'utiliser à leur tour.

```
fermeFichier (pt_fichier  in) ;
```

Exemples

Les neufs fonctions précédentes permettent de concevoir tous les algorithmes de traitements de fichier.Ces fonctions permettent de se familiariser avec les notions essentielles sans se préoccuper des problèmes particuliers au langage C.

récupérer dans un tableau tous les éléments d'un fichier

```
initialiseTab (nomDuFichier, tab, N)
/*   E : nomDuFichier : nom du fichier à créer et remplir
     E : disque (la mémoire durable est utilisée)
     S : tab, N : tableau de N éléments
La fonction remplit le tableau tab avec les éléments du fichier ayant pour nom :
     nomDuFichier. Le nombre d'éléments final du tableau est donné par N. On
     considère que le tableau est suffisamment grand pour accueillir tous les
     éléments du fichier */

{
     ptFichier <- consulterFichier(nomDuFichier) ;
     N <- 0 ;
     tant que (not EOF(pt_fichier)) {
          N <- N+1 ;
          tab(N) <- LireFichier(ptFichier) ;
     }
     FermerFichier(ptFichier);
}
```

écrire dans un fichier tous les éléments d'un tableau

```
ecritTabDansFich (tab, N, nomDuFichier)
/*   E : tab, N : tableau de N éléments
     E : nomDuFichier : nom du fichier à créer et remplir
     S : disque (seule la mémoire durable est modifiée)
La fonction crée un fichier à partir de nomDuFichier et remplit ce fichier des N
     éléments du tableau tab */
{
     pt_fichier <- CréerFich(nom_fichier) ;
     N <- 0 ;
     pour i allant de 1à N faire
          EcrireFich(pt_fichier, tab(i)) ;
          N <- N+1 ;
     fin pour
}
```

11. La récursivité

Principe

Une procédure (ou une fonction) est récursive quand elle s'appelle elle-même.

La récursivité est l'équivalent d'une boucle sans fin : le problème est de déterminer une condition de sortie.

Le problème algorithmique se ramène souvent à l'expression d'une suite mathématique.

Exemple : somme des N premiers nombres

Approche itérative

S = 1+2+3 ... + N-1 + N. On boucle N fois et on somme tous les nombres. Le problème est d'initialiser S : ici à 0.

- ***Algorithme***

```
Somme(N) : entier // somme des N premiers nombres
    S=0 ;
    Pour i de 1 à N
        S=S+i
    FinPour
    Return S
Fin
```

Approche récursive : formule de suite

S(0) = 0 : c'est la condition de sortie

S(N) = S(N-1) + 1 : c'est la formule récursive

- ***Algorithme***

```
Somme(N) : entier // somme des N premiers nombres
    Si N=0 return N ;
    Return S(N-1) + 1;
Fin
```

Remarque

Il n'est pas toujours évident de trouver une formulation sous la forme d'une suite. Il faut toutefois chercher la condition de sortie et la formule récursive.

Intérêt et limite de la récursivité

Intérêt

L'intérêt de la récursivité est qu'elle facilite considérablement l'écriture de certains algorithmes, particulièrement les algorithmes sur les arbres.

Limites

La récursivité est limitée car le nombre d'appels récursifs est limité : environ un centaine au maximum (chaque appel récursif implique pour le programme de mémoriser tout l'environnement, ce qui est coûteux en mémoire. En général, les compilateurs limite le nombre d'appels récursifs à une centaine. Chaque environnement est empilé pour être ensuite dépilé, d'où le message d'erreur : « stack overflow ».).

Il faut donc vérifier que cette limitation n'est pas contraignante.

Dans l'exemple traité, l'usage de la récursivité est à proscrire car on peut souhaiter calculer la somme des 10 000 premiers nombres. L'usage de la boucle est toutefois à optimiser en utilisant la formule = S = N(N+1)/2 !

Exercice 1

Ecrire l'algorithme récursif permettant de calculer N !

Le choix de la récursivité est-il pertinent ?

Exercice 2

On désire calculer le terme d'ordre n de la suite de Fibonacci définie par :

F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2).

Écrire une fonction itérative et une fonction récursive qui permet de calculer F(n) et un programme pour la tester.

Le choix de la récursivité est-il pertinent ?

Exercice 3

Ecrire l'algorithme récursif permettant de calculer X puissance N

Le choix de la récursivité est-il pertinent ?

Exercice 4

Ecrire l'algorithme récursif permettant de faire une recherche dichotomique dans un tableau trié.

Le choix de la récursivité est-il pertinent ?

Exercice 5

Ecrire l'algorithme récursif permettant de faire une recherche dans une liste chaînée.

Le choix de la récursivité est-il pertinent ?

12. ++ Complexité et optimisation

++ Les 3 types de complexité : apparente, spatiale et temporelle

On distingue 3 types de complexité : apparente (de structure), spatiale (de mémoire) et temporelle (de durée).

La complexité temporelle se divise en complexité temporelle théorique et pratique.

La complexité apparente (de structure)

Elle concerne la difficulté qu'éprouve le lecteur à comprendre l'algorithme. C'est une notion assez subjective. Toutefois, on prendra en compte plusieurs éléments :

1) La dénomination significative des variables
2) Usage standard des minuscules, majuscules, camel case.
3) La modularité de l'algorithme (utilisation de procédures et de fonctions)
4) Le bon usage des indentations.
5) La limitation des niveaux d'imbrication. Les débranchements structurés et la modularité permettent de le limiter.
6) La distinction, autant que possible, entre l'interface utilisateur et les traitements.
7) La distinction entre le traitement du cas général et le traitement des cas particuliers.
8) Le bon usage des commentaires.

La complexité spatiale (de mémoire)

La complexité spatiale correspond à l'encombrement mémoire de l'algorithme.

Elle peut être calculée théoriquement à partir de la connaissance des données mises en œuvre dans l'algorithme.

La complexité temporelle théorique (de durée d'exécution)

- ***Présentation***

  C'est un ordre de grandeur théorique du temps d'exécution de l'algorithme.

  Ce calcul concerne toujours des algorithmes avec des boucles.

  La mesure de la complexité est donnée par : « O ».

  « O » signifie : « complexité de l'ordre de ».

  On écrit : O(N), ou O(N/2) ou O(N^2) pour une complexité de l'ordre de N, de N/2, de N*N.

- ***Valeurs possibles***

  1, N, N/2, 2N, 3N, N_2, $2N_2$, $N_2/2$, racine(N), racine(N)/2, etc.

- ***Approximations***

  $N+1 = N$, $N_2+N = N_2$, etc.

➢ ***Exemple : complexité de la recherche du plus petit diviseur***

Algo 1 :

```
fonctionPPD (N) : entier // plus petit diviseur
    Pour i de 2 à N-1
        Si N mod i = 0
            Return i
        Finsi
    Finpour
    Return 1
FIN
```

Complexité: O(N)
N=100 : O(100)

Algo 2 :

```
fonctionPPD (N) : entier // plus petit diviseur
  si N mod 2 = 0
        return 2
  finsi
  pour i de 3 à racine(N) par pas de 2
        si N mod i = 0
              return i
        finsi
  Finpour
  Return 1
FIN
```

Complexité : O (racine(N)/2)
N=100 : O(5)

➢ ***Remarques***

- On voit dans l'exemple qu'on ne calcule pas la complexité à l'unité près. On n'écrit pas : O(N-2) mais O(N). O n'est qu'une approximation. Si le temps d'exécution n'est pas fonction de N, la complexité vaut : O (1)
- Il faudrait aussi prendre en compte le temps de calcul de racine(N).

➢ ***La différentes complexités temporelles théoriques***

- Temps constant : O(1)
- Temps linéaire : O(N)
- Temps proportionnel : O(N/2)
- Temps quadratique : O(N*N)
- Temps quadratique amélioré : O(N*N/2)
- Temps logarithmique : O(log2(N)) : c'est le cas de la recherche dichotomique.

La complexité temporelle pratique

C'est la mesure réelle du temps d'exécution de l'algorithme.

Elle est calculée en faisant des tests avec différents jeux de données.

Par exemple, on produit les résultats pour les algorithmes de tris selon la taille des tableaux.

Exemple de résultats :

Nombre de lignes :	10	100	1000	10000	50000
BULLE	0,16	20	2400		
EXTRACTION	0,12	7,3	680		
INSERTION	0,12	6,7	610		
SHELL	0,07	2	37	600	4200
ARBRE	0,2	3,5	50	660	3960
RAPIDE		2	28	365	2140

(d'après C. BAUDOIN et B. MEYER, cité dans GUYOT et VIAL)

Les 3 types d'optimisation

Corrélativement à la complexité, on peut distinguer 3 types d'optimisation : apparente, spatiale et temporelle.

L'optimisation temporelle se divise en optimisation temporelle théorique ou pratique

L'optimisation apparente (d'écriture)

L'optimisation d'écriture consiste à améliorer tous les points abordées dans la complexité apparente.

L'optimisation spatiale

Elle consiste à choisir des structures de données qui prennent moins de place et à éviter certaines méthodes algorithmique coûteuses en place (comme la récursivité, par exemple).

L'optimisation temporelle théorique

Elle consiste à diminuer la valeur de la complexité « O »,

L'optimisation temporelle pratique

Elle consiste à faire des tests de performance parmi plusieurs algorithmes en concurrence.

13. ++ Méthode pour écrire un algorithme

Méthode générale

1. **Comprendre le problème :** bien lire le sujet et bien comprendre ce qu'il y a à faire, c'est à dire le cas général correspondant au problème posé.
2. **Lister les entrées-sorties :** ce dont on a besoin pour résoudre le problème (les données) et ce qu'on va produire (les résultats).
3. **Lister les cas particuliers de départ** : c'est-à-dire les valeurs et situations particulières des données pour la résolution le problème.
4. **Trouver un principe de résolution pour les cas particuliers de départ.** Souvent, le principe de résolution des cas particuliers de départ est simple, mais ce n'est pas forcément le cas.
5. **Trouver un principe de résolution pour le cas général** : se donner les grandes lignes, c'est-à-dire les grandes actions, en français, de la méthode de résolution. Eventuellement, on peut se doter d'actions très générales, mais dans ce cas, on a intérêt à préciser ce dont on a besoin pour ces actions et ce qu'elles produisent (comme pour tout algorithme). Autrement dit, on peut se donner des procédures et des fonctions dans le principe de résolutions.
6. **Chercher des alternatives dans le cas général.** Trouver un principe de résolution pour ces alternatives. Souvent, le principe de résolution des alternatives au cas général est simple, mais ce n'est pas forcément le cas.
7. **Ecrire l'algorithme en détail, avec ou sans actions générales.** Ecrire en détail veut dire lister les opérations dans l'ordre et préciser ce qu'utilisent et ce que produisent les actions générales.

Méthode pour le principe de résolution

Pour trouver un principe de résolution, on peut essayer de trouver une solution « manuelle » : on imagine les données concrètement (dans des boîtes, sur des cartes, sur un schéma sur le papier, etc.) et on essaye de résoudre le problème « à la main ». La méthode qu'on utilise pour résoudre le problème « à la main » doit nous servir de fil conducteur pour écrire l'algorithme. Pour un tri, on peut penser à la façon de trier un jeu de cartes par exemple.

Méthode résumé

5. Entrées-Sortie,
6. Cas particuliers de départ,
7. Cas général,
8. Alternatives