AP11
ALGORITHMIQUE, LANGAGE C ET STRUCTURES DE DONNÉES

GIL PROTOY

TÉLÉCOM INT — 1ÈRE ANNÉE

AOÛT 2006

Table des matières

Ce support de cours est une compilation et une remise en forme de différents documents réalisés au département informatique de l'INT depuis une dizaine d'années.
Je tiens à remercier Dominique Calcinelli, Jean Michel Augier et Olivier Volt pour les apports les plus anciens. Je remercie également Luc Litzler pour le support de langage C ainsi que Daniel Millot, Philippe Lalevée, Christian Schüller et Christian Parrot pour leurs contributions.

Je remercie particulièrement Daniel Millot et Pierre Lanchantin pour leurs remarques, apports et commentaires durant la rédaction de ce document.

Les transparents et le polycopié ont été réalisés grâce à LaTeX et aux travaux de Philippe Lalevée et Denis Conan.

C1

PRÉSENTATION DU MODULE AP11

C1

2

Objectifs du module AP11

- **Algorithmique**
  - Savoir spécifier un problème : ce qui est en donnée, ce qui est en résultat
  - Savoir définir un algorithme permettant de résoudre ce problème
- **Langage C**
  - Savoir transcrire cet algorithme dans le langage cible
  - Savoir utiliser l'environnement de développement du langage C
- **Structures de données**
  - Connaître les structures de données plus complexes

C1

3

Contenu du C1

C1

4

1 Objectifs détaillés

1.1 Algorithmique : objectifs

5

■ **Vous devrez connaître** :

- ♦ les principes de conception d'un algorithme
- ♦ la syntaxe et la sémantique de notre langage algorithmique
- ♦ les structures de contrôle de base
- ♦ les concepts de base de l'abstraction procédurale
- ♦ les structures de données de base
- ♦ le concept de l'abstraction de données
- ♦ le concept de récursivité
- ♦ les méthodes de tri usuelles

- **les principes de conception d'un algorithme** :
  - « diviser pour résoudre »
  - « retarder le plus possible l'instant du codage »
- **les structures de contrôle de base** :
  - schéma séquentiel
  - schéma alternatif
  - schéma itératif
- **les concepts de base de l'abstraction procédurale** :
  - fonction
  - procédure
- **les structures de données de base** :
  - tableau
  - structure
- **le concept de l'abstraction de données** :
  - type abstrait
- **les méthodes de tri usuelles** :
  - tri par sélection
  - tri par insertion
  - tri à bulles
  - tri rapide
  - tri par fusion

1.2 Langage C : objectifs

6

■ **Vous devrez maîtriser** :

- ♦ la syntaxe et la sémantique du langage C
- ♦ l'utilisation simple de la chaîne de production
- ♦ l'utilisation de l'environnement de développement
- ♦ la programmation modulaire
- ♦ un outil de mise au point
- ♦ la gestion des fichiers en C

– **la chaîne de production** :
  – la commande *gcc* et ses options de base
  – une présentation plus complète de la commande *gcc* sera faite durant le module AM12

– **l'utilisation de l'environnement de développement** :
  – deux bibliothèques indispensables :
    – la bibliothèque standard incluant :
      – les fonctions d'allocation/désallocation
      – les fonctions d'entrée-sortie
      – les fonctions de traitement des chaînes de caractères
    – la bibliothèque mathématique

– **la programmation modulaire** :
  – la notion de module
  – la notion d'interface d'un module
  – la commande make et le fichier *makefile*

– **un outil de mise au point** :
  – *gdb* ou *ddd*

– **la gestion des fichiers en C** :
  – les fichiers texte
  – les fichiers binaires

7

1.3 Structures de données : objectifs

- connaître le principe de gestion d'une pile, d'une file et d'une liste
- savoir mettre en œuvre une liste (linéaire) triée gérée par pointeurs
- connaître le principe de gestion d'un arbre binaire
- savoir mettre en œuvre un arbre binaire de recherche géré par pointeurs

Structures de données : une bonne occasion pour approfondir la maîtrise des pointeurs en C

- **Mise en œuvre d'une liste (linéaire) triée gérée par pointeurs** :
  - application : gestion d'un répertoire de personnes
- **Mise en œuvre d'un arbre binaire de recherche géré par pointeurs** :
  - application : tri d'un ensemble de valeurs entières

C1

8

2 Organisation pédagogique

2.1 Présentiel

9

- **12 cours de 1h30** :
  - Présentation des concepts
- **10 séances de TD de 1h30** :
  - **Algorithmique** : validation des algorithmes préparés en hors présentiel
  - **Structures de données** : définitions des premières fonctions C
- **14 séances de TP de 1h30** :
  - Codage des algorithmes et exercices divers

2.2 Hors Présentiel

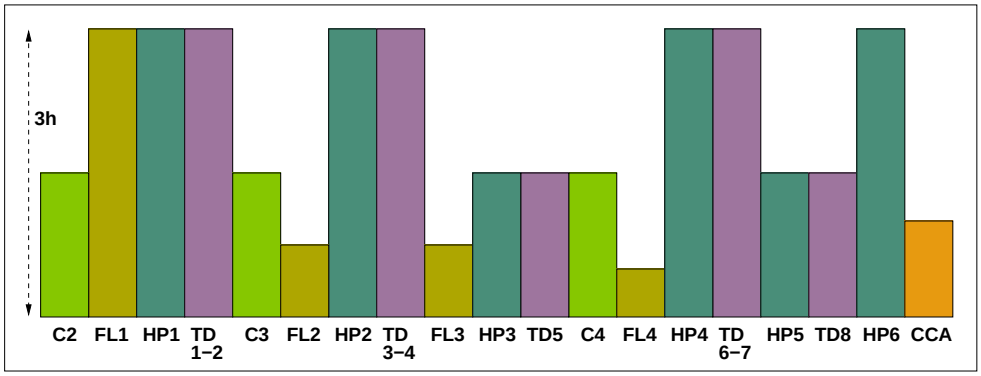
10

■ **Ce que vous aurez à faire en hors présentiel** :

- ♦ Etude détaillée du cours avec auto-évaluation sur la plate-forme *moodle*
- ♦ QCM en ligne sur la plate-forme *moodle*
- ♦ Définition des algorithmes
- ♦ Une partie du codage des algorithmes
- ♦ Une application de lissage d'image

11

2.3 Séquencement : Algorithmique



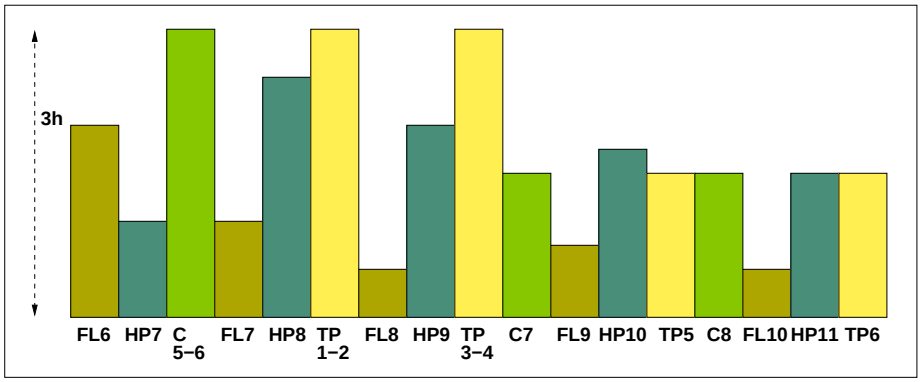
- **FL** : Formation en Ligne : étude détaillée du cours avec auto-évaluation, QCM
- **HP** : Hors Présentiel : définition des algorithmes
- **CCA** : Contrôle continu d'algorithmique

Séquencement détaillé :

- **C2** (1h30) : Introduction à l'algorithmique
- **FL1** (3h00) : Etude détaillée du cours avec auto-évaluation, QCM1
- **HP1** (3h00) : Définition des algorithmes
- **TD1-TD2** (3h00) : Compléments de cours, corrections des algorithmes
- **C3** (1h30) : Les types composites, les tris itératifs
- **FL2** (0h45) : Etude détaillée du cours sur les types composites avec auto-évaluation, QCM2
- **HP2** (3h00) : Définition des algorithmes (types composites)
- **TD3-TD4** (3h00) : Compléments de cours, corrections des algorithmes
- **FL3** (0h45) : Etude détaillée du cours sur les tris itératifs avec auto-évaluation, QCM3
- **HP3** (1h30) : Définition des algorithmes (tris itératifs)
- **TD5** (1h30) : Compléments de cours, corrections des algorithmes
- **C4** (1h30) : La récursivité, les tris récursifs
- **FL4** (0h30) : Etude détaillée du cours (récursivité) avec auto-évaluation, QCM4
- **HP4** (3h00) : Définition des algorithmes (récursivité)
- **TD7-TD8** (3h00) : Compléments de cours, corrections des algorithmes
- **FL5** (0h30) : Etude détaillée du cours (tris récursifs) avec auto-évaluation, QCM5
- **HP5** (1h30) : Définition des algorithmes (tris récursifs)
- **TD8** (1h30) : Compléments de cours, corrections des algorithmes
- **HP6** (3h00) : Préparation du contrôle continu d'algorithmique
- **CCA** (1h00) : Contrôle continu d'algorithmique

12

2.4 Séquencement : Langage C (1)



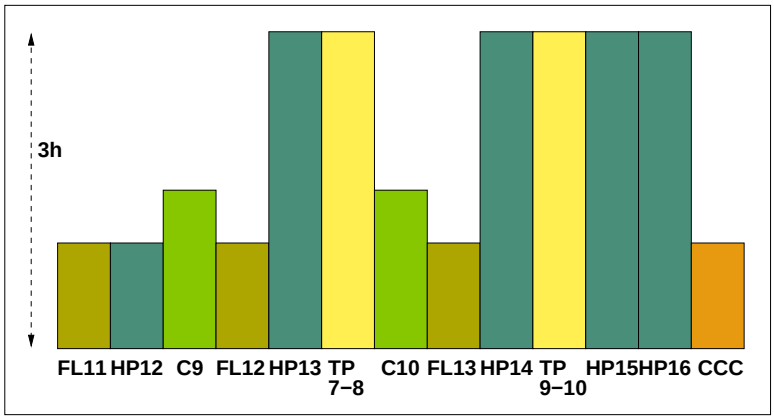
- **FL** : Formation en Ligne : étude détaillée du cours avec auto-évaluation, QCM
- **HP** : Hors Présentiel : codage des algorithmes

Séquencement détaillé :

- **FL6** (2h00) : Première approche du langage C avec auto-évaluation, QCM6
- **HP7** (1h00) : Etude de la section « L'essentiel du langage C » 1er niveau
- **C5-C6** (3h00) : Présentation du langage C, le passage de paramètres
- **FL7** (1h00) : Etude auto-évaluée du cours, QCM7
- **HP8** (2h30) : Etude de la section « L'essentiel du langage C » 2er niveau, début du codage des algorithmes
- **TP1-TP2** (3h00) : Codage des algorithmes
- **FL8** (0h30) : QCM8
- **HP9** (2h00) : Codages des algorithmes
- **TP3-TP4** (3h00) : Codages des algorithmes
- **C7** (1h30) : Abstraction de données, modularité en C, le fichier *makefile* et la commande *make*
- **FL9** (0h45) : Etude auto-évaluée du cours, QCM9
- **HP10** (1h45) : Découpage du fichier *triAbulles.c*, étude de la section « L'essentiel du langage C » 3er niveau
- **TP5** (1h30) : Application *Complexe*
- **C8** (1h30) : Options du compilateur, bibliothèques, mise au point
- **FL10** (0h30) : Etude auto-évaluée du cours, QCM10
- **HP11** (1h30) : Tri rapide utilisant le module *tabio (.h et .c)*
- **TP6** (1h30) : Mise au point d'un programme en utilisant la commande *ddd*, tri par insertion en utilisant *tabio.so* et *tabio.h*

13

2.5 Séquencement : Langage C (2)



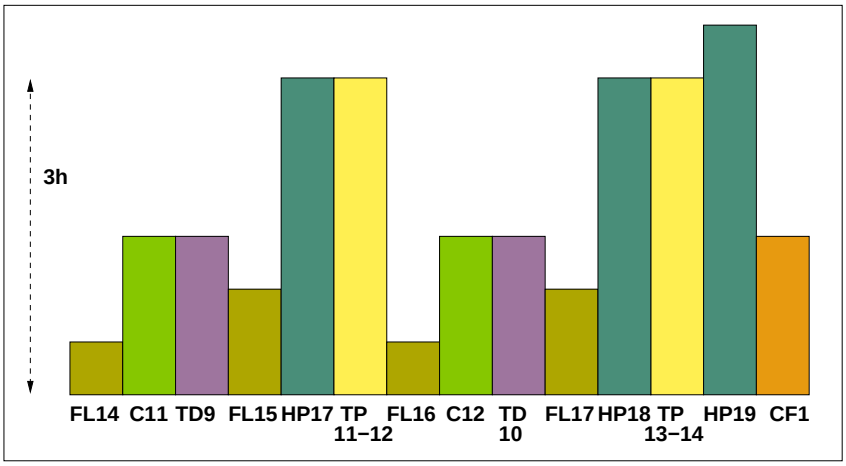
- **FL** : Formation en Ligne : étude détaillée du cours avec auto-évaluation, QCM
- **HP** : Hors Présentiel : codage des algorithmes, application Image, révisions
- **CCC** : Contrôle continu de langage C

Séquencement détaillé :

- **FL11** (1h00) : Etude auto-évaluée du cours « Généralités sur le fichiers », QCM11
- **HP12** (1h00) : Définition de l'algorithme du calcul du chiffre d'affaire
- **C9** (1h30) : Validation de l'algorithme du calcul du chiffre d'affaire, les fichiers en C
- **FL12** (1h00) : Etude auto-évaluée du cours, QCM12
- **HP13** (3h00) : Exercice en C sur les fichiers
- **TP7-TP8** (3h00) : Codage de l'algorithme du calcul du chiffre d'affaire, conversion binaire → binaire codé hexadécimal
- **C10** (1h30) : Le passage de paramètres (rappels), pointeurs et allocation dynamique
- **FL13** (1h00) : Etude auto-évaluée du cours, QCM13
- **HP14** (3h00) : Tri par fusion en utilisant des fichiers, arguments du *main*
- **TP9-TP10** (3h00) : Exercices de codage, retour sur *ddd*
- **HP15** (3h00) : Application Image : lissage d'une image en niveaux de gris
- **HP16** (3h00) : Préparation du contrôle continu de langage C
- **CCC** (1h00) : Contrôle continu de langage C

14

2.6 Séquencement : Structures de Données



- **FL** : Formation en Ligne : étude détaillée du cours avec auto-évaluation, QCM
- **HP** : Hors Présentiel : définitions des fonctions C
- **CF1** : Contrôle final 1

Séquencement détaillé :

- **FL14** (0h30) : Etude détaillée du C11 (partie Pile) avec auto-évaluation, QCM14
- **C11** (1h30) : Les structures linéaires
- **TD9** (1h30) : Définitions des premières fonctions C gérant une liste linéaire triée
- **FL15** (1h00) : Etude détaillée du cours avec auto-évaluation, QCM15
- **HP17** (3h00) : Suite de la définitions des fonctions C
- **TP11-TP12** (3h00) : Gestion d'un répertoire de personnes
- **FL16** (0h30) : Etude détaillée du C12 (transparents 1.1 à 1.6) avec auto-évaluation, QCM16
- **C12** (1h30) : Les structures arborescentes
- **TD10** (1h30) : Définitions des premières fonctions C gérant un arbre binaire de recherche
- **FL17** (1h00) : Etude détaillée du cours, QCM17
- **HP18** (3h00) : Suite de la définitions des fonctions C
- **TP13-TP14** (3h00) : tri de valeurs entières
- **HP19** (4h00) : Préparation du contrôle
- **CF1** (1h30) : Contrôle final 1

2.7 Validation

15

- **Contrôle continu (CC)** :
  - Note de participation (NP) :
    - Notation de vos solutions algorithmiques
    - Participation à la formation en ligne via la plate-forme *Moodle*
    - Les codages réalisés en hors présentiel et durant les séances de TP
    - la présence aux cours, TDs et TPs
  - Contrôle continu d'algorithmique (CCA) sur table : 1 heure
  - Contrôle continu de langage C (CCC) sur table : 1 heure
  - CC = (NP + CCA + CCC) / 3
- **2 contrôles finals de 1h30 sur table (CF1 et CF2)**
- **Note finale (NF)** :
  - NF = max ((CC + CF1)/2 , min (13 , max ((CC + CF2)/2 , CF2)))

C2

ALGORITHMIQUE ET PROGRAMMATION STRUCTURÉE

2

Objectifs du C2

- Prendre conscience de la problématique de l'activité de programmation
- Avoir quelques repères fondamentaux sur cette activité
- Connaître les fondements de la programmation procédurale
  - Notions de **type** (prédéfini) et de **variable**
  - Modification du contenu d'une variable par **affectation**
  - Organisation de l'enchaînement des actions via **les structures de contrôle**
- Connaître les principes de la programmation structurée :
  - La notion de **fonction**
  - La notion de **procédure**
  - Maîtriser la **cohérence** entre l'**appel** d'une fonction/procédure et sa **définition**
    - Respect de l'**ordre** et du **typage** des paramètres
    - Respect de la **visibilité** des objets
- Savoir construire des **algorithmes structurés**

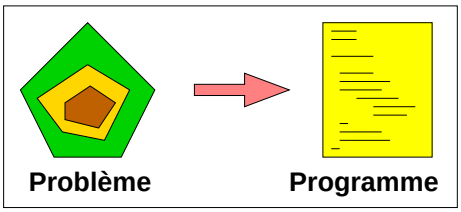
3

Contenu du C2

1 La programmation

4

1.1 Problème et Programme



5

- **Problème et Instance d'un problème**
  - **Problème = Généricité ; Son énoncé est sans ambiguïté = Spécification**
    Déterminer **un** élément de valeur minimum parmi un ensemble d'entiers
  - **Instance d'un problème**
    Un ensemble donné d'entiers
- **Programme** caractérisé par :
  - Un ensemble de **données** et un ensemble de **résultats**
  - Une **solution informatique** : description d'un ensemble d'**actions** à exécuter dans un certain **ordre** et dans un certain **langage**

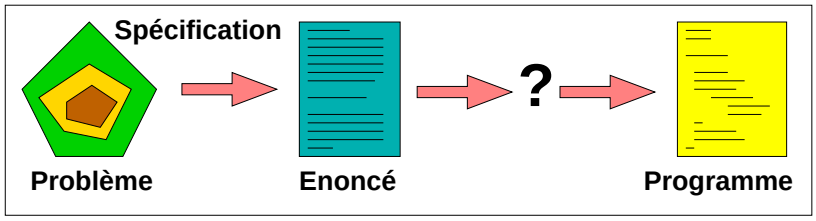
– **Enoncé d'un problème** = texte où sont définis sans ambiguïté :
  – Les données du problème
  – Les résultats recherchés
  – Les relations entre données et résultats

5	3	7	3	9
1	2	3	4	5

– **Un mauvais exemple** :
  – Rechercher l'indice du plus petit élément de cette suite d'entiers
  – Les indices 2 et 4 répondent tous les deux à l'énoncé du problème

– **Le problème bien spécifié** :
  – Soit I l'ensemble des indices des éléments de valeur minimum d'une suite d'entiers, déterminer le plus petit élément de I

1.2 Principes méthodologiques



6

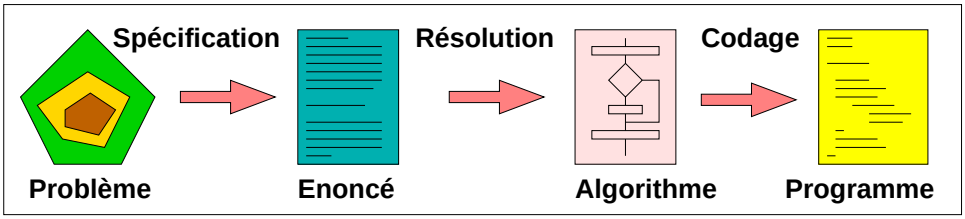
- **Abstraire**
  - Retarder le plus longtemps possible l'instant du codage
  - A partir de la spécification du problème, décrire les actions à exécuter dans un langage simple et non-contextuel
- **Décomposer**
  - *«...diviser chacune des difficultés que j'examinerais en autant de parties qu'il se pourrait et qu'il serait requis pour les mieux résoudre.» Descartes*
- **Combiner**
  - Résoudre le problème global par combinaisons successives de solutions de sous-problèmes déjà résolus

– **La pire des méthodes** :
  – Se lancer dans le codage, sans avoir au préalable analysé le problème, l'avoir résolu et avoir décrit dans un langage simple l'enchaînement des actions à exécuter, est une garantie de perte de temps

– **Langage non-contextuel** :
  – La sémantique d'une construction grammaticale, faite à partir des mots du langage, ne doit pas dépendre de l'environnement où elle se trouve

– **Décomposer et combiner** :
  – C'est la base d'une démarche puissante de conception : l'**approche descendante**

1.3 Algorithme



7

- **Description précise du processus de résolution d'un problème bien spécifié**
  - Lisible sans pour autant connaître la syntaxe du langage cible
- **Séquence ordonnée d'actions qui vont agir sur les données du problème pour produire le résultat escompté**
  - Quand on additionne deux entiers à la main, on déroule un algorithme
  - Nous sommes tous des **algoristes**

- **Langages algorithmiques** :
  - **Ordinogramme** : représentation graphique
    - Une action est définie dans un rectangle, une condition dans un losange
    - L'ordre des actions est spécifié par des flèches
    - Peu utilisé car ingérable pour un problème complexe
  - **Pseudo-code** : représentation textuelle
    - Ecrit en langage naturel, dépourvu d'éléments syntaxiques exotiques
    - Bien adapté pour décrire des méthodes de résolution complexes
    - C'est ce que nous utiliserons
- **Remarque sur le transparent** : la notion d'algorithme a été représentée par un petit ordinogrammme pour la distinguer clairement du programme en langage cible
- **Un peu d'histoire** :
  - Le mot **algorithme** vient du nom de l'astronome et mathématicien **Al-Huwarizmi**. Né dans l'actuelle république d'Uzbekistan, il fût un des membres de la « Maison de la Sagesse » fondée par *Al-Mâmûm*, calife de Bagdad, au début du IX^{e} siècle. Il nous a laissé au moins deux ouvrages :
    - Un livre d'arithmétique intitulé en latin *De numero Indorum* (l'original a été perdu) ; il y décrit les règles du calcul numérique développées par les Indiens quelques siècles auparavant : c'est notre système moderne de numération en base 10, incluant le zéro et utilisant l'arithmétique de position. On lui a attribué à tort la paternité de cette invention. Ce système fût introduit en Europe, quatre siècles plus tard, par *Léonard de Pise*. Il mettra deux siècles à s'y imposer définitivement face à l'opposition du clergé et des scribes qui employaient le système de numération romain et l'abaque ; d'où cette « querelle » entre **abacistes** et **algoristes**.
    - Son principal ouvrage s'intitule *Hisâb al-jabr wa'l_muqqâbala* qui signifie « science de la transposition et de la réduction ». Le mot arabe « al-jabr » est devenu plus tard « algèbre » en francais.
  - Nous lui sommes redevables de deux termes essentiels en mathématique.

8

1.4 Algorithme vs Programme C

```
i, n : Entier

saisir ( n )
pour i = 1 : Naturel à n
   si i mod 2 = 0 alors
      afficher ( i )
      allerAlaLigne ( )
   fsi
fpour i
```

```
#include <stlib.h>
#include <stdio.h>
int main ( ) {
    int n, i;
    scanf ( "%d", & n );
    for ( i  = 1 ; i <= n ; i++ )
        if ( ! ( i % 2 ) )
            printf ( "%d \n" , i );
    return EXIT_SUCCESS;
}
```

- **Met en avant l'essence de la méthode**
- **Lisible et plus concis**

– **A gauche, l'algorithme** :
  – Sans connaître pour l'instant notre langage algorithmique, on peut facilement deviner ce que va réaliser cet algorithme : l'affichage des entiers pairs compris entre 1 et n inclus
  – Cependant l'opérateur de modulo correspond ici au mot réservé *mod*

– **A droite, le programme C** :
  – On peut facilement faire le rapprochement entre les mots :
    – *pour* et *for*
    – *si* et *if*
  – Tout le reste est quasiment incompréhensible si on ne connaît pas les bases du langage

1.5 Méthodologie de programmation

9

- **Programmer, c'est communiquer** :
  - Avec la machine, avec soi même, avec les autres
    - Désignations évocatrices
    - Algorithmes en pseudo-code clairs et concis
    - Programmes indentés et commentés
- **Augmentation de la taille et la complexité des logiciels**
  - Travail en équipes de plus en plus importantes
- **Nécessité de créer des programmes corrects, efficaces, vérifiables et modifiables**
  - Conséquences économiques et humaines de plus en plus coûteuses

- **Communiquer avec soi-même** :
  - Reprendre un programme mal présenté et non-commenté, ne serait-ce que six mois plus tard, n'est pas une chose facile

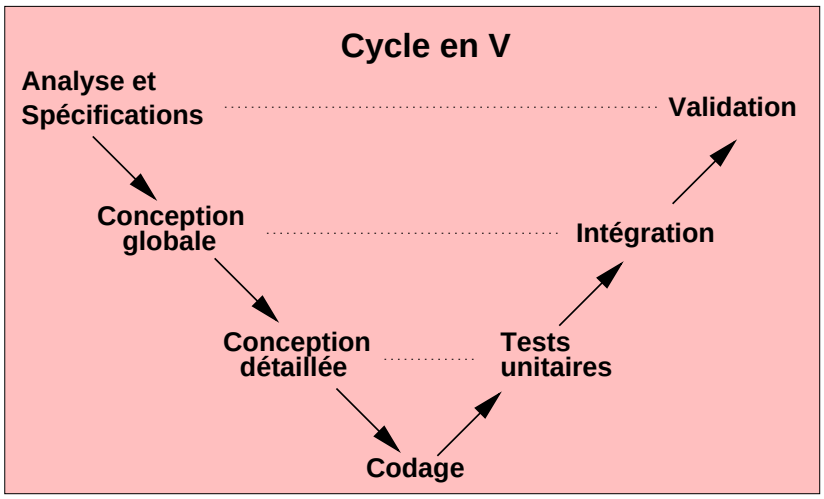
- **Désignations évocatrices** :
  - Elles permettent de réduire les commentaires nécessaires

- **Taille et complexité des logiciels** :
  - Le temps de développement s'évalue en **années-hommes**
  - Ordre de grandeur pour un projet conséquent : 50 années-hommes

- **Correction des programmes et conséquences économiques et humaines** :
  - Deux exemples spectaculaires :
    - L'explosion de la première fusée Ariane V après 40 secondes de vol ; dûe à un débordement de mot mémoire dans le système embarqué ; un demi milliard de dollars de pertes en matériel
    - La dérive d'un missile antimissile Patriot pendant la première guerre du Golfe ; dûe à une succession d'erreurs d'arrondi dans une horloge du système embarqué ; 28 soldats américains tués

10

1.6 Introduction au génie logiciel -



- **Cycle de vie du développement d'un logiciel**
- **Mise en œuvre durant votre projet informatique PI12**

- **Analyse et spécifications** :
  - Définir clairement le problème
  - Recenser les données
- **Conception globale** :
  - Dégager les grandes fonctionnalités
  - En déduire le découpage de l'application
- **Conception détaillée** pour chaque entité :
  - Décrire et représenter les données
  - Définir les algorithmes en pseudo-code
- **Codage** :
  - Transcription algorithmes → programmes en langage cible
- **Tests unitaires** :
  - Vérifier chaque entité
  - Utiliser des jeux de test
- **Intégration** :
  - Réunir les différentes entités pour produire l'application
  - Mise au point
- **Validation** :
  - Vérifier que l'application respecte toutes les spécifications du problème

- C'est aussi un **processus itératif** :
  - Correction des erreurs
  - Ajout de nouvelles fonctionnalités

1.7 Complexité d'un algorithme -

11

- **Caractériser les performances d'un algorithme indépendamment de la machine, du système, du compilateur et du programmeur**
- **Evaluation de la complexité d'un algorithme**
  - Taille du problème : n
  - Nombre d'opérations significatives : T(n)
  - Espace mémoire nécessaire : M(n)
  - Evaluation au mieux, **au pire**, en moyenne
- **Notations asymptotiques**
  - **borne asymptotique supérieure** : notation O
  - $f(n) = O(g(n))\ si\ \exists\, c > 0\ et\ n_0 > 0\ tel\ que\ f(n) \leq cg(n)\ \forall\, n \geq n_0$
  - **borne asymptotique approchée** : notation Θ
  - $f(n) = \Theta(g(n))\ si\ \exists\, c_1, c_2 > 0\ et\ n_0 > 0\ tel\ que\ c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall\, n \geq n_0$
  - **borne asymptotique inférieure** : notation Ω
  - $f(n) = \Omega(g(n))\ si\ \exists\, c > 0\ et\ n_0 > 0\ tel\ que\ cg(n) \leq f(n)\ \forall\, n \geq n_0$

– **Nombre d'opérations et Espace mémoire** :
  – L'analyse temporelle de complexité est prioritaire.
  – La complexité mémoire ne peut excéder la complexité temporelle
  – Imaginer un algorithme linéaire (en $O(n)$) qui nécessiterait n^2 objets en mémoire

– **Evaluation au pire** :
  – C'est une **garantie** de performances quelle que soit l'instance du problème traitée

– **Evaluation en moyenne** :
  – Le tri rapide a une complexité en $O(nlogn)$ en moyenne mais en $O(n^2)$ dans le pire des cas

– **Notations asymptotiques** :
  – $f(n) = O(g(n))$: f(n) ne croît pas plus rapidement que g(n)
    – Problèmes irréguliers → problèmes de graphe
  – $f(n) = \Theta(g(n))$: f(n) croît de la même façon que g(n)
    – Problèmes réguliers → analyse numérique
  – $f(n) = \Omega(g(n))$: f(n) ne croît pas plus lentement que g(n)
    – C'est la borne inférieure de complexité du problème

1.8 Temps d'exécution -

- **10^9 opérations par seconde**
- **n : nombre de données à traiter**
- **T(n) : complexité de l'agorithme**

12

$n \backslash T(n)$	n	$nlog_2n$	n^2	n^3	2^n	$n!$
10	0,01μs	0,03μs	0,1μs	1μs	1μs	3,6 ms
10^2	0,1μs	0,65μs	10μs	1 ms	4×10^{13} a	3×10^{141} a
10^3	1μs	10μs	1 ms	1 s	3×10^{284} a	∞
10^4	10μs	0,13 ms	0,1 s	16,6 mn	∞	∞
10^5	0,1 ms	1,66 ms	10 s	11,5 j	∞	∞
10^6	1 ms	20 ms	16,6 mn	31,7 a	∞	∞

- **10^9 opérations par seconde** :
  - Cohérent avec la fréquence actuelle de cadencement des processeurs : $\simeq$ 3 gigahertz
  - Il y a vingt ans : 10^6 opérations par seconde

- **Croissance exponentielle** :
  - Pour une taille de problème de 10 : T(n)=n^3 $\simeq$ T(n)=2^n ($2^{10} \simeq 1000$)
  - Mais pour une taille de problème de 100 ...
  - Quant à T(n)=n ! ; 100 ! dépasse le nombre présumé d'atomes de l'univers
  - le symbole ∞ dans le tableau ci-dessus s'interprête plutôt comme une durée inaccessible à l'échelle humaine mais néanmoins finie

13

1.9 Exemples de fonctions de complexité -

■ **Produit de matrices carrées n x n**

- Nombre de multiplications
- Algorithme classique : $c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$
  - $T(n) = \Theta(n^3) \rightarrow$ trop d'opérations
- Borne inférieure de complexité : $\Omega(n^2)$
- Algorithme de *Strassen* : $T(n) = \Theta(n^{log_2 7}) = O(n^{2,81})$
- Meilleur algorithme connu : $T(n) = O(n^{2,376})$

■ **Tri de n valeurs**

- Nombre de comparaisons (dans le pire des cas)
- Algorithmes basiques : $T(n) = O(n^2)$
- Borne inférieure de complexité : $\Omega(nlogn)$
- Meilleurs algorithmes connus : $T(n) = O(nlogn)$

– **Produit de matrices** :
  – Algorithme classique :
    – à mettre en relation avec le tableau précédent
  – Algorithme de Strassen :
    – Algorithme récursif
    – Quand on multiplie deux matrices 2 x 2, il faut 8 multiplications
    – En calculant certains termes intermédiaires, il suffit de 7 multiplications
    – D'où la complexité en $\Theta(n^{log_2 7})$
    – Problème « en principe » ouvert
– **Tri** :
  – Algorithmes basiques : tri par sélection, par insertion, à bulles
  – Meilleurs algorithmes connus : tri par tas, par fusion
  – Problème résolu : la borne inférieure de complexité est atteinte

2 Le langage algorithmique

14

15

2.1 Types

- **Les types**
  - **Classer les objets selon** :
    - l'ensemble des valeurs que peut prendre l'objet
    - l'ensemble des opérations permises sur cet objet
  - **Tous les types ont un nom** (identificateur)
  - **Tous les objets ont un type** (prédéfini ou défini explicitement) **et une désignation**
- **Les types prédéfinis**
  - Naturel (entiers positifs ou nuls)
  - NaturelNonNul (entiers strictement positifs)
  - Entier
  - Réel
  - Booléen : VRAI et FAUX
  - Caractère
  - Chaîne (de caractères)

– Les types **définis explicitement** seront abordés lors du C3

– **Types dans notre langage algorithmique et dans le langage cible**

  – **Plage de valeurs** :
    – Dans notre langage algorithmique, la plage des valeurs du type Entier correspond à l'**ensemble dénombrable** $\mathbb{Z}$
    – Dans notre langage algorithmique, la plage des valeurs du type Réel correspond à l'**ensemble indénombrable** $\mathbb{R}$
    – Dans le langage cible, le type choisi correspond à un **ensemble fini** ; le cardinal de cet **ensemble** dépendra du nombre d'octets utilisés pour stocker les variables de ce type

2.2 Variables

16

■ **Ce sont des objets manipulés et modifiables par l'algorithme**

♦ **Identificateur** :

► Permet de distinguer un objet sans ambiguïté

► Pour éviter les confusions, les identificateurs de type commenceront par une majuscule, les identificateurs de variable par une minuscule

♦ **Définition** obligatoire et **initialisation** facultative :

► n : Entier

► x , y = 0.5 : Réel

♦ **Contenu d'une variable** :

► Une variable est un contenant dont le contenu (valeur) va changer au cours de l'algorithme

► Abstraction d'un emplacement mémoire de la machine

♦ **Elles ne sont pas forcément visibles partout**

– **Identificateur** :
  – C'est votre choix
  – Des identificateurs représentatifs permettent d'alléger les commentaires
  – Ne soyez pas radin sur leur longueur, sans être excessif

– **Définition et initialisation** :
  – **Attention** : une variable non initialisée contient en général une valeur indéterminée ! ! !

– **Une variable est stockée en mémoire** pendant l'exécution du programme
  – Elle occupe un nombre d'octets dépendant de son type et parfois de la plate-forme : Dos, Windows, Linux, Solaris, HPUnix

– **Visibilité** :
  – Dans notre langage algorithmique, la visibilité d'une variable sera toujours restreinte à l'entité où elle a été définie

2.3 Changement d'état

17

- **Etat courant d'un algorithme**
  - Ensemble des valeurs des variables à l'étape considérée
- **Changement d'état** :
  - **Action permettant de modifier le contenu d'une variable** :
    - Durée finie, comme toute action
    - Produit un résultat prévisible et sans ambiguïté
  - **Préconditions** :
    - Caractérisation de l'état courant **avant** l'exécution de l'action : { Q }
  - **Postconditions** :
    - Caractérisation de l'état courant **après** l'exécution de l'action : { R }

– **Préconditions et postconditions**

Un exemple simple :

incrémenter (x) $\{ x < n \}$ $\{ x \leq n \}$

18

2.4 Actions

- **Action élémentaire**
  - **Affectation : désignation d'objet ← valeur**
    - **désignation** : identificateur de variable,. . .
    - **valeur** : expression invoquant des variables, des constantes et des opérateurs ; évaluée lors de l'exécution et donc dépendante de l'état courant
    - n ← 1
    - x ← x + 1
    - y ← x
  - **Affichage et saisie** (pour les besoins du langage algorithmique)
    - afficher (x + 1) // expression d'un type simple
    - allerAlaLigne ( ) // pour passer à la ligne suivante
    - saisir (x) // x désigne une variable de type simple
- **Action composée**
  - Ensemble d'actions simples à exécuter pour produire un résultat complexe
  - Exprimée en terme d'actions simples à l'aide des structures de contrôle

- **Affectation** : là, il faut être rigoureux
  - x ← (y + 1) / 2
  - Opérande gauche : une désignation d'objet obligatoirement
  - Opérande droit : une expression éventuellement réduite à l'invocation d'une constante ou du contenu d'une variable

- **Affichage et saisie** :
  - Ces actions sont considérées comme simples uniquement pour les besoins du langage algorithmique
  - Dans le langage cible, il s'agit d'appel de « sous-programmes » dont la définition n'a rien de simple

- **Actions composées** :
  - Dans notre langage algorithmique, elles seront mises en évidence uniquement par indentation du texte

- **Commentaires**
  - Ils seront introduits par les deux caractères consécutifs : //
  - Ils courent jusqu'à la fin de la ligne

2.5 Les opérateurs de notre langage algorithmique

19

- **Opérateurs logiques spécifiques au type Booléen**
  - Négation logique : **non**
  - Et logique : **et**
  - Ou logique : **ou**
- **Opérateurs arithmétiques communs aux types « entiers » et Réel**
  Les types "entiers" sont : NaturelNonNul, Naturel et Entier
  - Addition : $+$
  - Soustraction : $-$
  - Multiplication : $\times$
- **Opérateurs arithmétiques spécifiques aux types « entiers »**
  - Division entière : **div**
  - Opérateur modulo : **mod**
- **Opérateur arithmétique spécifique au type Réel**
  - Division réelle : $/$

– **Sans oublier les opérateurs de comparaison** : $<, >, \leq, \geq, =, \neq$

– **Types dans notre langage algorithmique et dans le langage cible**
  – En mathématique, comme dans notre langage algorithmique, la multiplication des réels est associative : $(x \times y) \times z = x \times (y \times z)$
  – Dans le langage cible, elle ne l'est pas systématiquement du fait d'arrondis

2.6 Base des langages de programmation impératifs

20

- **Les langages impératifs sont caractérisés par** :
  - **la notion de variable**
  - **La modification du contenu des variables par** :
    - **L'affectation**
    - **La définition explicite du flot de contrôle** utilisant les trois schémas algorithmiques (ou structures de contrôle) :
      - Le schéma séquentiel
      - Le schéma alternatif (ou schéma de choix)
      - Le schéma itératif (ou répétitif)

- **Dans le langage cible que nous allons vous présenter il existe un quatrième schéma** :
  - le redoutable « aller à » ou « retourner vers »
  - l'instruction *goto* :
    Edger Dijkstra, mathématicien et informaticien hollandais, dont vous entendrez citer le nom dans les cours d'optimisation combinatoire de seconde année, était connu pour son aversion de l'instruction goto en programmation.
    En 1968, il publia un article "Go To Statement Considered Harmful" qui est considéré comme un élément majeur de la dépréciation de l'instruction goto.
  - l'instruction *goto* est un résidu de l'assembleur, un langage de bas niveau, que vous rencontrerez dans le module AM12
  - Nous éviterons cette instruction

- **Quant à la récursivité**
  - Ne serait-ce pas un schéma algorithmique ?
  - Non, car tout algorithme récursif peut être redéfini avec un schéma itératif

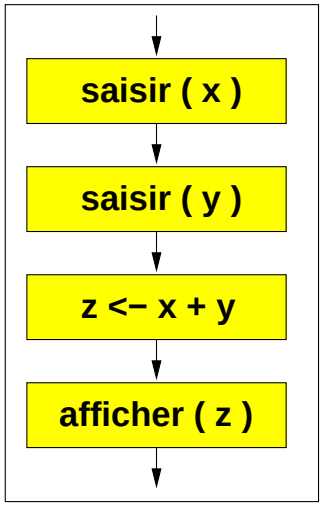
2.7 Le schéma séquentiel

21

■ **Pas grand chose à dire mais c'est essentiel** :

♦ Les actions doivent être exécutées dans l'ordre d'apparition dans le texte de l'algorithme

```
x , y , z : Réel

saisir ( x )
saisir ( y )
z ← x + y
afficher ( z )
```



■ **Utilisé seul, il n'y a pas beaucoup d'exemples intéressants**

– L'*ordinogramme* sur la droite est une autre façon de décrire un algorithme. Mais elle est peu adaptée dès que le problème à résoudre devient tant soit peu complexe. Les rectangles symbolisent les actions et les flèches leur enchaînement.

2.8 Le schéma alternatif (ou schéma de choix)

22

■ **Permet d'exprimer des décisions** :

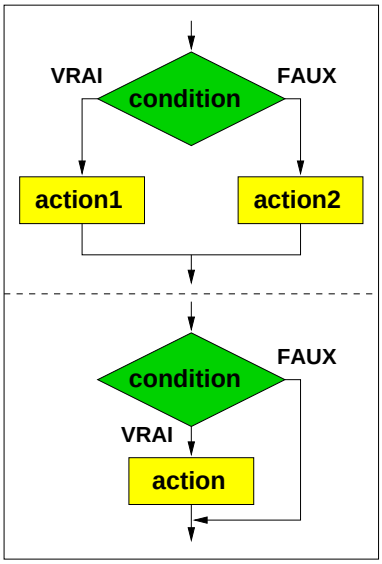
♦ Action(s) conditionnée(s) par le test d'une expression booléenne appelée *condition*

Mots réservés : **si alors sinon fsi**

si condition **alors**
    action1 // condition = VRAI
sinon
    action2 // condition = FAUX
fsi

Forme simplifiée :

si condition **alors**
    action
fsi



– Dans un ordinogramme, une condition est symbolisée par un losange

– **Mots réservés** : il ne faut surtout pas les employer comme identificateur

– L'action à exécuter peut être **simple** ou **composée**

– Une construction algorithmique **maladroite** :

si condition **alors**
    rien
sinon
    action
fsi

– La construction algorithmique **correcte** :

si non condition **alors**
    action
fsi

2.9 Maximum de deux entiers -

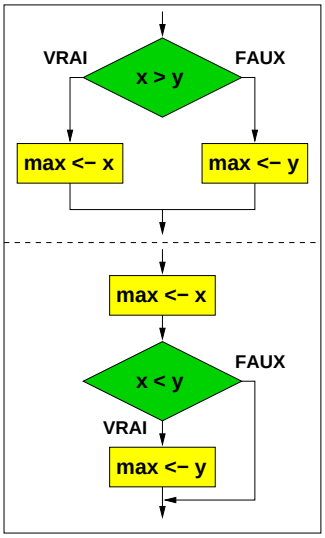
23

■ **Première solution** :

x , y , max : Entier
si x > y **alors**
    max ← x
sinon
    max ← y
fsi

■ **Deuxième solution** :

x , y , max : Entier
max ← x
si x < y **alors**
    max ← y
fsi



2.10 Le schéma de choix généralisé -

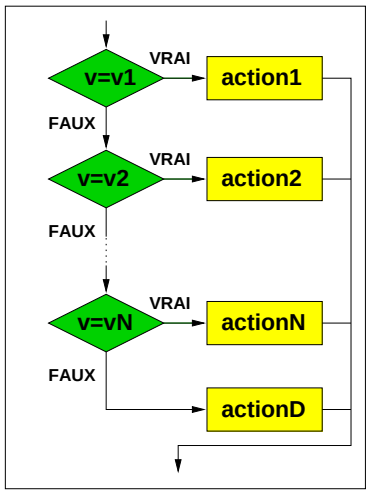
24

■ **Permet d'éviter un cascadage excessif de schémas alternatifs** :

♦ Compare une valeur v «entière» à une série de constantes entières
Mots réservés : **cas où vaut autre fcas**
cas où v **vaut**
v_1 : $action_1$
v_2 : $action_2$
...
v_N : $action_N$
$autre$: $action_D$
fcas

♦ $action_i$ est exécutée si $v = v_i$; on quitte alors le schéma

♦ $action_D$ est exécutée si $v \neq v_i \forall i = 1, ..., N$



– Le terme **généralisé** est à prendre avec circonspection
  – Le schéma n'utilise pas d'expression booléenne
  – Il teste l'égalité entre une valeur de type « entier » et un ensemble de constantes entières

– L'action $action_D$ est l'action par défaut ; elle peut ne pas exister.

– Un exemple d'utilisation : **la gestion d'un menu**

  – En fonction d'une valeur saisie par l'utilisateur, exécution d'une fonctionnalité de l'application

```
choix  : Naturel

afficher( "Saisir votre choix : " )
saisir ( choix)

cas où choix vaut
        1 : afficher ( ... )
        2 : ajouter ( ... )
        3 : retirer ( ... )
        ...
        autre : afficher ( "Erreur saisie" )
fcas
```

2.11 Le schéma itératif canonique

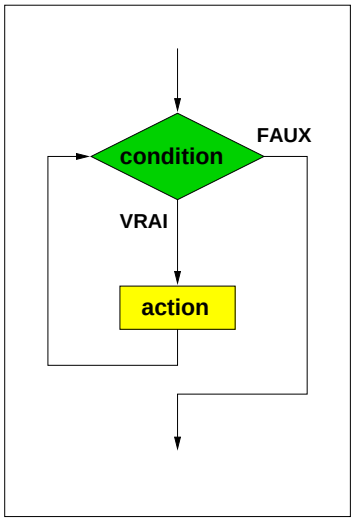
25

■ **Permet de répéter une action simple ou composée un nombre de fois non forcément spécifié à l'avance** :

- ♦ Teste une expression booléenne appelée *condition*
  Mots réservés : **tant que faire ftq**

  tant que condition **faire**
  action
  ftq

- ♦ L'action peut ne jamais être exécutée



– L'action à exécuter peut être **simple** ou **composée**

– **L'action doit, à un moment donné, rendre la condition fausse** → boucle infinie

2.12 Le schéma itératif canonique (variante) -

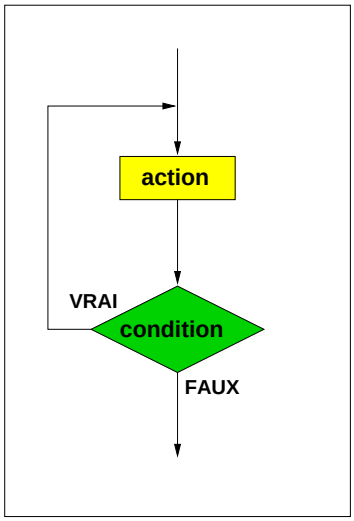
26

■ **Permet de répéter une action simple ou composée un nombre non nul de fois non forcément spécifié à l'avance** :

♦ Teste une expression booléenne appelée *condition*
Mots réservés : **faire tant que**

faire
action
tant que condition

♦ L'action est exécutée au moins une fois



– L'action à exécuter peut être **simple** ou **composée**

– **L'action doit, à un moment donné, rendre la condition fausse** → boucle infinie

– **Autre formulation**
  – Mots réservés : **répétez jusqu'à**

répétez
action
jusqu'à condition

  – issue de langage comme *pascal* ou *ada*
  – il suffit d'**inverser** la condition pour retrouver **notre** construction

2.13 Le schéma itératif « pour »

27

■ **Permet de répéter une action simple ou composée un nombre de fois spécifié**

♦ **Parcours, éventuellement partiel, d'un ensemble indicé**

Mots réservés : **pour à [pas] fpour** (par défaut **pas** = 1)

```
pour i = valeurInitiale : Entier à valeurFinale [ pas h ]
        action
fpour i
```

■ **Somme des Entiers compris entre 1 et n**

```
somme = 0 : Naturel
pour i = 1 : Naturel à n
        somme ← somme + i
fpour i
afficher( somme )
```

■ **Somme des Entiers pairs compris entre 2 et n**

```
somme = 0 : Naturel
pour i = 2 : Naturel à n pas 2
        somme ← somme + i
fpour i
afficher( somme )
```

– Rien n'est exécuté si *valeurInitiale* > *valeurFinale* et *pas* > 0

– Dans l'exemple, on sort de la boucle dès que *i* est strictement supérieur à **n**

– **Utilisation la plus importante** :
  – Le parcours des éléments d'un tableau (défini dans le C3)

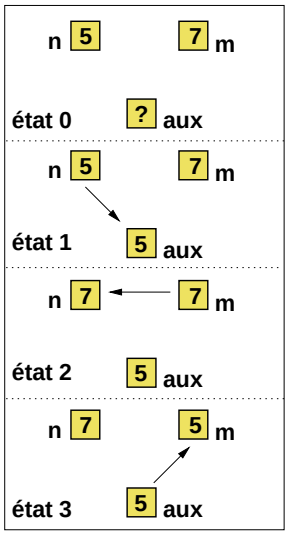
– **Remarque** : la variable de contrôle *i* peut être de type Entier ou Naturel selon la succession des valeurs qu'elle peut prendre

2.14 Echange du contenu de deux variables -

28

■ **Algorithme** :

```
n , m , aux : Entier

saisir ( n )
saisir ( m )   // 0 : les variables n et m
               // contiennent les valeurs 5 et 7
aux ← n        // 1 : sauvegarde du contenu
               // de n dans la variable aux
n ← m          // 2 : on peut transférer le
               // contenu de m dans la variable n
m ← aux        // 3 : il reste à charger m avec
               // le contenu de la variable aux
afficher ( n )
afficher ( m )
```

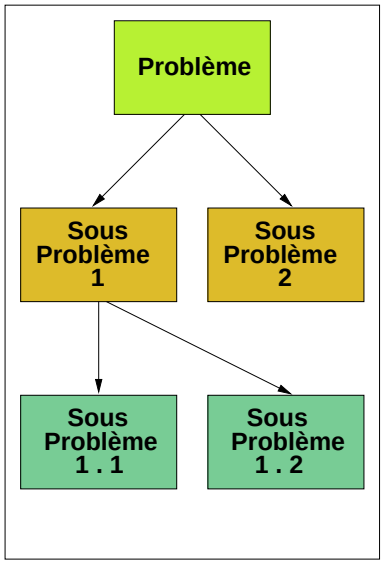


3 La programmation structurée

3.1 Décomposer pour résoudre

30

- **Analyse descendante**
  - Approche incrémentale consistant à analyser la spécification d'un problème selon une arborescence
- **Décomposition/combinaison à chaque niveau de l'arborescence**
  - Décomposer le problème en un certain nombre de sous-problèmes qu'on supposera résolus
  - Combiner les solutions des sous-problèmes pour résoudre le problème courant
  - Résoudre les sous-problèmes par la même approche de décomposition/combinaison



– A chaque niveau de décomposition, la solution du sous-problème correspondra à la définition d'un sous-programme
  – **Fonction**
  – **Procédure**

– Un **algorithme** sera défini comme un ensemble de **fonctions** et de **procédures**

– **Réutilisation** des composants logiciels

31

3.2 Fonction

Action paramétrable qui produit une valeur unique

- **Exemple** :

```
fonction minimum ( n, m : Entier ) : Entier
        si n < m alors
            retourner n
        fsi
        retourner m
ffct minimum
```

- **En-tête** (appelée encore signature ou prototype) :
  - identificateur, précédé du mot reservé **fonction**
  - liste parenthésée de paramètres typés **non modifiables par l'appelé**
  - type du **résultat**
- **Corps** :
  - Définition de variables **locales**
  - Composition d'actions dont **au moins une** permet de renvoyer une valeur à l'appelant : mot réservé **retourner**

– **En-tête**
  – **Signature** dans le langage algorithmique
  – **Prototype** dans le langage cible
  – les paramètres d'une signature sont qualifiés de **formels**

3.3 Désignation/Appel d'une fonction

32

- mots réservés : **fonction retourner ffct**
- **Désignation** : généralement un nom évocateur du résultat
  - ♦ max ← maximum (x,y), **si** estVide (e) **alors**. . .
- **Appelant** :
  - ♦ Fournit des valeurs, les paramètres **effectifs**, qui seront recopiés dans les paramètres formels en respectant l'**ordre** et le **typage**
- **Séquence d'appel** :

```
a = 5 , b = 7 , min : Entier
min ← minimum ( a , b )
afficher ( "Le minimum de " , a , " et de " , b , " vaut " , min )
allerAlaLigne ( )
```

– **Désignation**
  – L'identificateur est en général un substantif, sauf dans le cas où la fonction renvoie un booléen ; on emploie alors un identificateur formé de la concaténation du mot *est* et d'un adjectif.
  Les articulations sont mises en évidence par une majuscule : *estParfait*, *estVide*

– **Attention** à l'ordre et au type des paramètres lors de l'appel d'une fonction
  – Il doit y avoir correspondance **stricte** entre l'appel et la signature

– La valeur **retournée** par une fonction doit être impérativement utilisée, sinon elle est perdue.
  – affectation à une variable de même type dans l'appelant
  – utilisée comme condition dans un schéma alternatif ou itératif quand elle est de type booléen
  – utilisée comme paramètre effectif pour un autre appel de fonction/procédure dans l'appelant

– **Appel avec des expressions comme paramètres effectifs**

```
retourner cnp ( n - 1 , p ) + cnp ( n - 1 , p - 1 )
```

– **Autre séquence d'appel** pour la fonction minimum :

```
a = 5 , b = 7 : Entier

afficher ( "Le minimum de " , a , " et de " , b , " vaut " , minimum ( a , b ) )
allerAlaLigne ( )
```

– La séquence d'appel sera formalisée dans ce que nous appellerons plus loin la **fonction principale**

3.4 Procédure

Action paramétrable

33

■ **Exemple** :

```
constante PI = 3,14 . . .
procédure calculerPérimètreAire ( donnée rayon : Réel
                                  résultat périmètre, aire : Réel )
    périmètre ← 2 × PI × rayon
    aire ← PI × rayon × rayon
fproc calculerPérimètreAire
```

■ **En-tête** (signature, prototype) :

- ♦ identificateur précédé du mot réservé **procédure**
- ♦ liste de paramètres en **donnée** (**non modifiables par l'appelé**)
- ♦ liste de paramètres en **donnée et en résultat** (éventuellement vide)
- ♦ liste de paramètres en **résultat** (éventuellement vide)

■ **Corps** :

- ♦ Définition de variables locales
- ♦ Composition d'actions élémentaires

– **Attention** à l'ordre et au type des paramètres lors de l'appel d'une procédure
  – Il doit y avoir correspondance **stricte** entre l'appel et la signature

3.5 Désignation/Appel d'une procédure

34

- mots réservés : **procédure donnée donnée-résultat résultat fproc**
- **Désignation** : généralement un verbe
  - ♦ afficher (x + 1)
- **Appelant** :
  - ♦ Fournit des paramètres **effectifs** à l'appelé en respectant l'**ordre** et le **typage** pourqu'il utilise :
    - ► leur **valeur** (cas des paramètres formels **donnée**)
    - ► leur **zone mémoire de stockage** (cas des paramètres formels **résultat**)
    - ► **les deux** (cas des paramètres formels **donnée-résultat**)
- **Séquence d'appel** :

```
r = 3.7 , p , a : Réel
calculerPérimètreAire ( r , p , a )
afficher ( "Le périmètre du cercle de rayon " , r , " vaut " , p , " et son aire " , a )
allerAlaLigne ( )
```

– Au cas où il n'existe pas d'opérateur **puissance** dans le langage cible, pour calculer $rayon^2$, multiplions rayon par lui-même.

– La « **transmission** » vers l'appelant se fait par **affectation** explicite de la valeur d'une expression aux paramètres en résultat ou en donnée-résultat.

– Le mot réservé **donnée** n'est jamais employé pour un paramètre formel de fonction ; c'est implicite.

– Quand vous aurez pratiqué suffisamment le langage C, vous vous demanderez peut être à quoi sert la distinction entre fonction et procédure. Ce sont les héritages d'autres langages impératifs comme *Pascal* et *Ada*.
En C, il n'y a que des fonctions et nous aurions pu développer un langage algorithmique basé uniquement sur le concept d'**action**.
Nous pensons néanmoins que cette distinction reste plus « structurante » pour démarrer l'activité de programmation.

35

3.6 La fonction principale

- Il y a toujours un déclencheur extérieur initiant l'exécution de l'algorithme
- Il invoque la fonction **principale**

```
fonction principale ( ) : CodeRetour // synonyme du type Booléen : KO, OK

    rayonDuCercle : Réel
    sonPérimètre, sonAire : Réel // réceptacles pour les paramètres en résultat

    saisir ( rayonDuCercle )
    calculerPérimètreAire ( rayonDuCercle , sonPérimètre , sonAire )
    afficher ( "   Le périmètre du cercle de rayon   " , rayonDuCercle ,
               "   est égal à   " , sonPérimètre , ", et son aire à  " , sonAire )
    allerAlaLigne ( )
    retourner OK

ffct principale
```

– La fonction **principale** correspond au point d'entrée du programme, ce qui va être appelé au départ. C'est elle qui va appeller la ou les fonctions/procédures de l'algorithme selon son arborescence de décomposition. Elle doit être réduite au strict minimum. Elle renvoie à l'appelant, le déclencheur extérieur, une valeur de type **CodeRetour** (synonyme du type Booléen) qui peut prendre deux valeurs :
  – **OK** : exécution sans problème du programme
  – **KO** : un problème a eu lieu à l'exécution

  Un exemple simple :

```
fonction principale ( ) : CodeRetour
    choix : Entier
    afficher ( "Saisir une valeur comprise entre -3 et 5 inclus" )
    saisir ( choix )
    si choix < -3 ou choix > 5 alors
            afficher ( "Valeur incorrecte" )
            retourner KO
    sinon
            // suite du programme
            retourner OK
    fsi
ffct principale
```

– Alors que le problème ne se pose pas pour les fonctions, il est impératif de définir, dans l'appelant d'une procédure, des variables correspondant aux paramètres formels en résultat. La transmission de ces variables en argument lors de l'appel de la procédure permet de les utiliser comme **receptacles** lors des affectations réalisées dans le corps de celle-ci.

4 Un exemple de décomposition

36

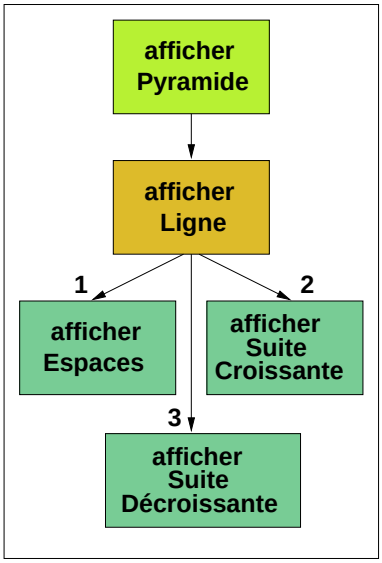
37

4.1 Une pyramide de chiffres -

- **Définir une procédure qui affiche une pyramide de chiffres telle que celle-ci :**

```
          1
        1 2 1
      1 2 3 2 1
    1 2 3 4 3 2 1
```

- On supposera que l'affichage est réalisé en mode texte ; qu'il n'y a pas de possibilité de se placer à un endroit spécifié de la fenêtre
- La procédure sera paramétrée par un naturel non nul *hauteur*, pouvant aller de 1 à 9



– On se limite à une hauteur de 9 maximum pour pouvoir gérer simplement l'alignement.

– Il est en fait possible de se déplacer dans une fenêtre *shell* ; On vous le montrera dans le module AM12.

4.2 afficherPyramide - afficherLigne -

38

■ Définition de la procédure *afficherPyramide*

```
procédure afficherPyramide ( donnée hauteur : NaturelNonNul )
        pour ligne = 1 : NaturelNonNul à hauteur
            afficherLigne ( ligne , hauteur )
        fpour ligne
fproc afficherPyramide
```

■ Définition de la procédure *afficherLigne*

```
procédure afficherLigne ( donnée l, h : NaturelNonNul )
        afficherEspaces ( l , h )
        afficherSuiteCroissante ( l )
        afficherSuiteDécroissante ( l )
        allerAlaLigne ( )
fproc afficherLigne
```

– La procédure *afficherEspaces* a besoin de deux paramètres *l* et *h* car le nombre de caractères espace à écrire en début de la ligne *l* dépend de la hauteur *h* de la pyramide.

– Les procédures *afficherSuiteCroissante* et *afficherSuiteDécroissante* ont besoin du seul paramètre *l*.

– On veillera le plus possible à utiliser des identificateurs différents pour le paramètre formel et pour le paramètre effectif afin de bien distinguer leurs rôles respectifs.

39

4.3 afficherEspaces - afficherSuiteCroissante -

■ Définition de la procédure **afficherEspaces**

```
procédure afficherEspaces ( donnée i, n : NaturelNonNul )
        pour j = 1 : NaturelNonNul à n - i
            afficher ( " " ) // affiche un caractère espace
        fpour j
fproc afficherEspaces
```

■ Définition de la procédure **afficherSuiteCroissante**

```
procédure afficherSuiteCroissante ( donnée i : NaturelNonNul )
        pour j = 1 : NaturelNonNul à i
            afficher ( j )
        fpour j
fproc afficherSuiteCroissante
```

– Dans la procédure *afficherEspaces*, il n'y a pas exécution du schéma **pour** quand le paramètre formel *i* a pour valeur *n*

40

4.4 afficherSuiteDécroissante -

■ Définition de la procédure **afficherSuiteDécroissante**

```
procédure afficherSuiteDécroissante ( donnée i : NaturelNonNul )

        pour j = i - 1 : NaturelNonNul à 1 pas -1
            afficher ( j )
        fpour j

fproc afficherSuiteDécroissante
```

– Là aussi il n'y a pas exécution du schéma **pour** quand le paramètre formel *i* a pour valeur 1

– **Pour vous persuader** : l'algorithme non décomposé

```
procédure afficherPyramide ( donnée hauteur : NaturelNonNul )

    pour ligne = 1 : NaturelNonNul à hauteur

        pour j = 1 : NaturelNonNul à hauteur - ligne
            afficher ( " " )
        fpour j
        pour j = 1 : NaturelNonNul à ligne
            afficher ( j )
        fpour j
        pour j = ligne - 1 : NaturelNonNul à 1 pas -1
            afficher ( j )
        fpour j

        allerAlaLigne ( )

    fpour ligne

fproc afficherPyramide
```

TD1-TD2

LES EXERCICES À PRÉPARER

Exercices validés durant les TD1-TD2

1.1 Un petit calcul de partage

On veut répartir *nbPersonnes* personnes en *nbGroupes* groupes, d'effectifs les plus équilibrés possible (différents d'au plus une unité). Définir l'algorithme qui à partir de *nbPersonnes* et de *nbGroupes* détermine les effectifs *effectifGrandsGroupes* et *effectifPetitsGroupes*, ainsi que les nombres *nbGrandsGroupes* et *nbPetitsGroupes* des respectivement grands et petits groupes.
Spécifier l'énoncé du problème. Définir l'algorithme qui effectue la saisie des données, réalise le calcul et affiche les résultats. Cet algorithme doit contenir la définition de la procédure *partager* et de la fonction *principale*.
Voici, pour ce premier exercice, la présentation que vous devez adopter :

- **Enoncé**
  - **Données** nbPersonnes , nbGroupes : NaturelNonNul
  - **Résultats** effectifGrandsGroupes, effectifPetitsGroupes, nbGrandsGroupes , nbPetitsGroupes : Naturel
  - **Relations entre les données et les résultats** : à vous de les trouver

- **Squelette de l'algorithme**

```
procédure partager ( donnée nbP, nbG : NaturelNonNul,
                     résultat effectifGG, effectifPG, nbGrandsG, nbPetitsG )

  // à vous de jouer

fproc partager

fonction principale ( ) : CodeRetour

  nbPersonnes, nbGroupes : NaturelNonNul
  effectifGrandsGroupes, effectifPetitsGroupes : Naturel
  nbGrandsGroupes, nbPetitsGroupes : Naturel

  afficher ( "Nombre de personnes : " )
  saisir ( nbPersonnes )
  afficher ( "Nombre de groupes : " )
  saisir ( nbGroupes )

  partager ( nbPersonnes, nbGroupes, effectifGrandsGroupes, effectifPetitsGroupes,
             nbGrandsGroupes, nbPetitsGroupes )

  // il vous reste à afficher les résultats en tenant compte
  // que le nombre de grands groupes peut être nul

  retourner OK

ffct principale
```

1.2 L'année bissextile

On veut déterminer si l'année *a* est bissextile ou non.
Si *a* n'est pas divisible par 4 l'année n'est pas bissextile.
Si *a* est divisible par 4, l'année est bissextile sauf si *a* est divisible par 100 et pas par 400.
Spécifier l'énoncé du problème. Définir l'algorithme qui effectue la saisie de la donnée, détermine si l'année est bissextile ou non et affiche le résultat. Cet algorithme doit contenir la définition des fonctions *estBissextile* et *principale*.

1.3 Le temps plus une seconde

Un temps donné est représenté sous la forme : *heure*, *minute* et *seconde* de type naturel. On veut lui ajouter une seconde et afficher avec la même représentation le temps ainsi obtenu.
Spécifier l'énoncé du problème. Définir l'algorithme qui effectue la saisie des données, réalise le calcul et affiche les résultats. Cet algorithme doit contenir la définition de la procédure *ajouterUneSeconde* et de la fonction *principale*.

1.4 Le nombre de jours de congés

Dans une entreprise, le calcul des jours de congés payés s'effectue de la manière suivante :

- Si une personne est rentrée dans l'entreprise depuis moins d'un an, elle a droit à deux jours de congés par mois de présence, sinon à 28 jours au moins.
- Si c'est un cadre, s'il est âgé d'au moins 35 ans et si son ancienneté est supérieure à 3 ans, il lui est accordé 2 jours supplémentaires. S'il âgé d'au moins 45 ans et si son ancienneté est supérieure à 5 ans, il lui est accordé 4 jours supplémentaires, en plus des 2 accordés pour plus de 35 ans.

Spécifier l'énoncé du problème. Définir l'algorithme qui calcule le nombre de jours de congés à partir de l'*âge*, de l'*ancienneté* et de l'appartenance au collège *cadre* d'un employé. Afficher le résultat. Cet algorithme doit contenir la définition des fonctions *nombreJoursCongés* et *principale*.

1.5 Calcul de n !

On veut calculer n !.
Spécifier l'énoncé du problème. Définir l'algorithme qui demande la saisie d'un naturel *n*, qui calcul n ! et qui affiche le résultat. Définissez la fonction *factorielle* de deux façons ; en utilisant le schéma *tant que* puis le schéma *pour*. Définissez également la fonction *principale*.

1.6 Nombres premiers

On veut déterminer si un naturel *nombre* strictement supérieur à 1 est premier (par convention, 1 n'est pas premier).
Spécifier l'énoncé du problème. Définir l'algorithme qui détermine si *nombre* est premier ou non et qui affiche le resultat. Cet algorithme doit contenir la définition des fonctions *estPremier* et *principale*.

1.7 Quotient et reste d'une division entière

On veut réaliser en même temps, à l'aide de soustractions successives, la division entière et le calcul du modulo de deux nombres naturels non nuls *dividende* et *diviseur*.
Spécifier l'énoncé du problème. Définir l'algorithme qui effectue la saisie des données, réalise le calcul et affiche les résultats. Cet algorithme doit contenir la définition de la procédure *calculerQuotientEtReste* et de la fonction *principale*.

1.8 Recherche du zéro d'une fonction par dichotomie

Définir une fonction *zéroFonction*, qui calcule le zéro d'une fonction *f(x)* sur l'intervalle [*a*, *b*], avec une précision *epsilon*. La fonction *f* et les réels *a*, *b* et *epsilon* sont donnés.
Soit *f* une fonction continue monotone sur l'intervalle [*a*, *b*], où elle ne s'annule qu'une seule et unique fois. Pour trouver ce zéro, on procède par dichotomie, c'est à dire que l'on divise l'intervalle de recherche par deux à chaque étape. Soit *m* le milieu de [*a*, *b*]. Si *f(m)* et *f(a)* sont de même signe, le zéro recherché est dans l'intervalle [*m*, *b*], sinon il est dans l'intervalle [*a*, *m*].
Définir également la fonction *principale*.

1.9 Suite de Fibonacci

Définir une procédure, *calculerTermeFibonacci*, qui calcule la valeur *u* et le rang *k* du premier terme de la suite de Fibonacci dépassant une borne entière positive donnée *p*.
La suite de Fibonacci est définie par :
$u_0 = 0\,;\, u_1 = 1$
$u_n = u_{n-1} + u_{n-2}\ \forall n > 1$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13
u_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233

L'histoire :
Léonard de Pise publiait en 1202 sous le nom de Fibonacci (fils de Bonacci) son ouvrage Liber Abaci qui introduit le système décimal en Europe. «Un homme installe un couple de lapins dans un emplacement. Combien de couples de lapins peuvent être produits en un an à partir de ce couple, sachant que la nature de ces lapins est telle que tous les mois chaque couple engendre un nouveau couple qui devient productif au bout du second mois ?»
Définir également la fonction *principale*.

1.10 Intégration par la méthode des trapèzes

Définir une fonction, *intégrale*, qui retourne la valeur de l'intégrale d'une fonction *f* réelle continue sur l'intervalle [*a*, *b*].
Soit *f* une fonction continue sur l'intervalle [*a*, *b*]. L'intégration consiste à découper cet intervalle, en *n* sous-intervalles de longueur *delta*. L'intégrale d'un sous-intervalle [*x*, *x+delta*] est approximée au trapèze de base *delta* et de côtés *f(x)* et f *f(x+delta)*.
Définir également la fonction *principale*.

1.11 Nombres parfaits

Définir une procédure, *afficherNombresParfaits*, qui affiche la suite de tous les nombres parfaits inférieurs ou égaux à un nombre naturel non nul donné noté *n*. Un nombre est dit parfait s'il est égal à la somme de ses diviseurs autres que lui-même.
Exemple : 28 = 1 + 2 + 4 + 7 + 14
Voici la liste des nombres parfaits inférieurs à 10000 : 6, 28, 496, 8128.
la procédure, *afficherNombresParfaits* devra appeler la fonction *estParfait* et sera elle-même appelée par la fonction *principale*.

1.12 Racines d'une équation du second degré

Définir une procédure, *résoudreEquationSecondDegré*, qui affiche, si elles existent, les racines réelles d'une équation du second degré représentée par ses coefficients réels *a*, *b* et *c* donnés.
Vous disposez d'une fonction **racineCarrée** qui prend un paramètre de type réel et qui renvoie un réel ; voici sa signature :

fonction racineCarrée (x : Réel) : Réel

Adoptez une démarche descendante et n'oubliez pas la fonction *principale*.

1.13 Affichage du développement de l'expression $(x + y)^n$

Pour mémoire, les coefficients binomiaux sont définis par :

$C_n^n = C_n^0 = 1$ et $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ si $0 < p < n$, n étant un naturel non nul et p étant un naturel

Définir une procédure *développer* qui prend en donnée un paramètre *n* de type NaturelNonNul et qui affiche le développement de l'expression $(x + y)^n$ dont on rappelle que c'est la somme des termes $C_n^p x^{n-p} y^p$, $0 \leq p \leq n$.
If faut obligatoirement adopter une démarche descendante ; i. e. définir une fonction et plusieurs procédures. On ne doit pas trouver d'appel de l'action *afficher* dans la définition de la procédure *développer* elle-même.
Soignez particulièrement l'affichage. x^n devra être affiché sous la forme **x^n**. La présence d'un caractère espace peut être représenté par le symbole '␣'. L'algorithme, s'il était implémenté, devrait afficher dans la fenêtre *shell* :

- pour *n* = 1 : **x + y**
- pour *n* = 2 : **x^2 + 2 x y + y^2**
- pour *n* = 3 : **x^3 + 3 x^2 y + 3 x y^2 + y^3**

Définir également la fonction *principale*.

TD1-TD2

C3

LES TYPES COMPOSITES ET LES TRIS ITÉRATIFS

Objectifs du C3

2

- Savoir définir et utiliser des types « énumération »
- Savoir définir et utiliser des types composites
  - Tableaux
  - Structures
- Connaître les principales méthodes de tri itératifs par comparaison
  - tri par sélection
  - tri par insertion
  - tri à bulles

Contenu du C3

3

1 Les types définis explicitement

4

1.1 Le type énumération

5

- **Définition d'un type**
  - mot réservé **type**
  - **Identificateur** commencant par une majuscule
  - **=** Description du type
- **Type énumération** : spécifie la liste des valeurs du type
  - **Définition du type** :
    - **type** Jour = { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
  - **Utilisation** :

```
jour  : Jour // Définition d'une variable de type Jour
si jour = VENDREDI alors
    afficher ( "Ce soir, c'est le week-end" )
    allerAlaLigne ( )
fsi
```

  - **Intérêt** : lisibilité

– On pourrait associer à chaque jour de la semaine une valeur « entière »
  – *Lundi* → 1
  – *Mardi* → 2
  – ...
  – *Dimanche* → 7

– L'algorithme serait alors :

```
jour  : Jour // Définition d'une variable de type Jour

si jour = 5 alors // 5 ⟺ Vendredi

    afficher ( "Ce soir, c'est le week-end" )
    allerAlaLigne ( )

fsi
```

6

1.2 Les types composites

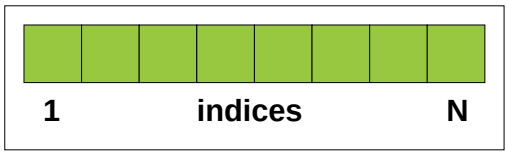
- **Motivation**
  - Composer des types élémentaires pour décrire des structures de données
- **Type composé**
  - Construit à partir d'autres types
  - Notion de composant
    - Type des composants
    - Organisation de ces composants
    - accès à ces composants
  - **Tableau** et **structure**
- **Opérations**
  - Opérations permises sur le composé
  - Opérations permises sur les composants du composé

– **Tableau** → ensemble **fini** de valeurs

– **Structure** → objet du monde réel

7

1.3 Les tableaux

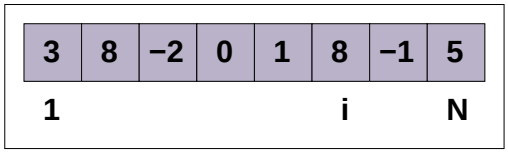


- **Collection indicée d'éléments de même type**
- **Taille fixe** : constante *N*
- **Opérations** :
  - Sur le composé : **uniquement** l'accès aux composants
  - Sur les composants : celles **autorisées** sur leur **type**
- **Accès aux composants** :
  - Par leur indice : expression d'un type ordonné
    les indices vont de 1 à *N* inclus
  - Accès direct (en temps constant)
- **Algorithmes** :
  - Saisie et Affichage
  - Recherche d'un élément
  - Tri

– **Accès aux composants** :
  – indice : expression de type « entière »
  – Accès en temps constant : les temps nécessaires pour accéder au 1er et au N^{ème} éléments sont identiques.

– **Opérations sur le composé** : aucune
  – Il n'est pas possible, par exemple, de transférer **globalement** le contenu d'un tableau dans un autre tableau (de même type).
  – Le transfert doit s'opérer composant par composant → schéma itératif

1.4 Tableau à une dimension



8

■ **Définition de type**
- ♦ **constante** N = 8
- ♦ **type** Vecteur = **tableau**[1 .. N] **de** Entier

■ **Définition de variables**
- ♦ t1 : Vecteur
- ♦ t2 : **tableau**[1 .. N] **de** Entier

■ **Accès au i^{ème} élément**
- ♦ x ← t1 [i]
- ♦ t2 [i+1] ← t2 [i]

■ **Algorithme de base**

pour i = 1 : NaturelNonNul **à** N
    traiterElément (t [i])
fpour i

– **Définition de constante**
  – **constante** N = 8 // constante entière
  – **constante** π = 3,14159265358979323846 // constante réelle

– **Définition de variables** : deux possibilités
  – Définition d'un type explicite tableau et utilisation de ce type pour définir des variables.
    → définition de plusieurs tableaux de même type (même type de base et même nombre d'éléments)
  – Définition directe d'une variable de type tableau.

– **Attention à la rigueur des constructions algorithmiques**
  – Une construction incorrecte :

```
si
    pour i = 1 : NaturelNonNul à N
        t [i] > valeur alors
            t [i] ← valeur
    fpour i
fsi
```

  – La construction correcte :

```
pour i = 1 : NaturelNonNul à N
    si t [i] > valeur alors
        t [i] ← valeur
    fsi
fpour i
```

9

1.5 Tableau à deux dimensions -

- **Définition de type**
  - **constante** N = 3, M = 4
  - **type** Matrice = **tableau**[1 .. N][1 .. M] **de** Entier
- **Définition de variables**
  - m1 : Matrice
  - m2 : **tableau**[1 .. N][1 .. M] **de** Entier
- **Accès aux composants**
  - c [i][j] ← c [i][j] + a [i][k] × b [k][j]

	1		j	M
1	3	2	5	7
i	9	4	1	4
N	7	8	5	9

- **Algorithme de base**

```
pour ligne = 1 : NaturelNonNul à N // première dimension
        pour colonne = 1 : NaturelNonNul à M // deuxième dimension
                traiterElément ( t [ligne][colonne] )
        fpour colonne
fpour ligne
```

– **Opérations sur le composé** : aucune

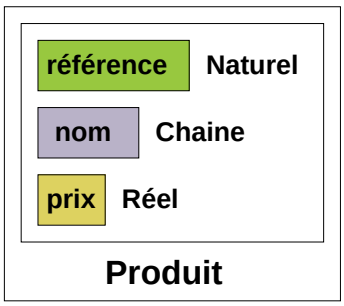
– Il est possible de définir des tableaux de dimension supérieure à deux.

– Il n'est bien sûr pas possible de définir un tableau de dimension infini.

1.6 Les structures

10

- **Regroupement d'informations de différents types**
  - **Représente un objet du monde réel**
  - Les composants sont appelés **champs**
- **Opérations** :
  - Sur le composé : **l'affectation** et **l'accès** aux composants
  - Sur les composants : celles **autorisées** sur leurs **types**
- **Définition de type**
  - **type** Produit = **Structure**
    référence : Naturel
    nom : Chaîne
    prix : Réel
    fstruct



- **Définition de variables**
  - produit1, produit2 : Produit
- **Affectation** (objets de même type)
  - produit1 ← produit2
- **Accès aux champs**
  - afficher (produit1.référence)
  - produit2.prix ← 3,5

– Le type structure est principalement destiné à regrouper des informations de types différents, mais il y a des exceptions.

– Définition du type *Point* à coordonnées réelles dans le plan

type Point = **Structure** // préférable à l'utilisation d'un tableau de deux réels
abscisse : Réel
ordonnée : Réel
fstruct

– L'intérêt est ici d'utiliser des identificateurs signifiants plutôt que des indices.

2 Les tris itératif

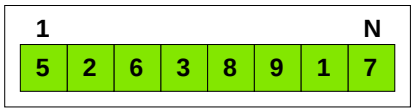
11

2.1 Spécification du problème

12

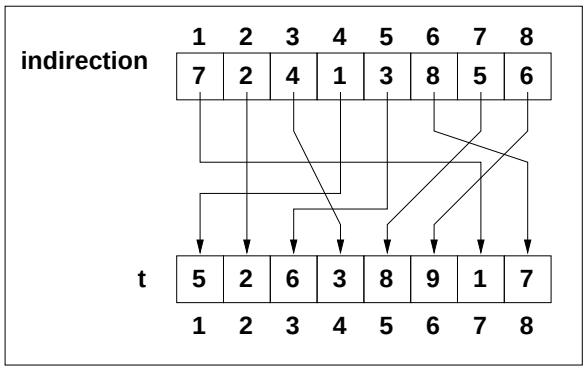
- **Données**
  - une séquence de *N* éléments comparables $< a_1, a_2, \ldots, a_N >$
- **Résultats**
  - une permutation $< a'_1, a'_2, \ldots, a'_N >$ de ces *N* éléments telle que : $a'_1 \leq a'_2 \leq \ldots \leq a'_N$
- **Eléments à trier**
  - **clé** : valeur de référence pour le tri
    les clés appartiennent à un ensemble totalement ordonné
  - **données satellites** : a priori permutées en même temps que la clé
- **Hypothèses simplificatrices**
  - élément = clé = entier (pas de données satellites)
  - séquence = t : **tableau** [1. . .N] **de**

Entier



- **Contrainte** : éviter d'utiliser un tableau auxiliaire

– **Données satellites**
  – en pratique, les éléments à trier sont d'un type structure dont un des champs est la **clé**
  – pensez par exemple à un fichier de la *Sécurité Sociale* avec, pour chaque individu, le numéro *INSEE*, les nom et prénoms, l'age, . . .
  – lors du tri, les données satellites peuvent être permutées mais, pour plus d'efficacité, on peut créer un tableau d'**indirection** comme le montre l'exemple ci-dessous :



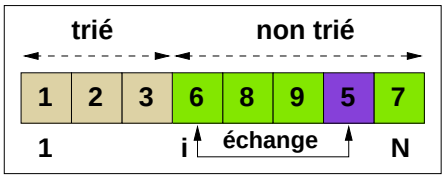
  – Dans la présentation des tris, on rangera systématiquement les éléments du tableau dans l'**ordre croissant**.

13

2.2 Tri par sélection

■ **Principe**

♦ à chaque étape i, on place le minimum de t [i. . .N] en t [i]



■ **Méthode**

♦ i ← 1

♦ recherche du minimum dans t [i. . .N]

♦ échange de t [i] avec le minimum

♦ i ← i+1

♦ arrêt pour i = N-1 (le maximum est forcément en t [N])

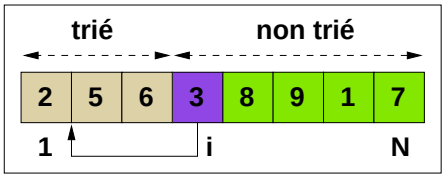
– **Principe général**
  – Un algorithme (itératif ou récursif) doit progressivement **rapprocher** l'organisation du tableau de celle de **la séquence triée**.
  – L'objectif d'une étape peut être :
    – mettre un élément à sa position **finale** dans le tableau
    – mettre un élément à sa position **relative** dans un sous-tableau **déjà trié**
  – Une étape du tri par sélection considère une **nouvelle position finale** dans le tableau et recherche **l'élément** qui doit y être stocké.
  – La 1ère étape place le minimum en 1ère position, etc. . .

14

2.3 Tri par insertion

■ **Relation de récurrence**

- t [1. . .1] est trié
- **si** t [1. . .(i-1)] est trié **alors** trier (t [1. . .i]) = insérer t [i] à sa place dans t [1. . .i]



■ **Insertion**

- recherche de la **position d'insertion**
  - recherche séquentielle
  - recherche dichotomique (t [1. . .(i-1)] est trié)
- «**faire une place**» pour l'élément à insérer → **décalages**
  - en même temps que la recherche **séquentielle** de la position d'insertion

– **Principe**
  – Une étape du tri par insertion considère un nouvel élément à classer et recherche sa **position relative** dans une sous-suite **déjà triée** pour l'y **insérer**.
  – La 1ère étape place le 1er élément dans une sous-suite vide, etc. . .

– **Recherche dichotomique**
  – Elle améliore la complexité de la méthode en terme de nombre de **comparaisons**.
  – Elle accélère la détermination de la position de *t[i]* mais il faudra néanmoins effectuer des **décalages** dans le sous-tableau trié.
  – → **dégradation de performances**.

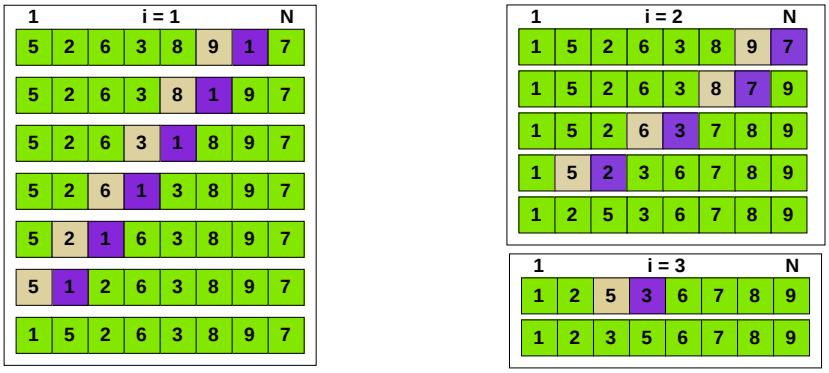
– **Recherche séquentielle**
  – si on s'y prend bien, les décalages peuvent être effectués **en même temps** que la recherche de la position d'insertion.

15

2.4 Tri à bulles

■ **Principe**

- ♦ à chaque étape i
  - ► parcourir t [i. . .N] de N vers i
  - ► échange de deux éléments consécutifs s'ils ne sont pas dans le bon ordre
- ♦ les éléments «légers» remontent vers le début du tableau



– **Principe**
  – Une étape du tri à bulle **réorganise** les éléments du tableau et place correctement **au moins** un élément en lui assurant un **positionnement définitif** par rapport à tous les autres.

– **Pour i = 1**
  – On est sûr que l'élément en **début** du tableau après la 1ère itération est un élément de **valeur minimum**.

– **Boucle englobante**
  – normalement, on pousuit jusqu'à **i = N - 1**
  – mais comme l'exemple le montre, il est possible que le tableau soit **déjà trié avant**.
  – on peut améliorer les performances de la méthode en gérant une variable booléenne qui va détecter s'il y a eu **au moins un échange** à une itération donnée.
  – si ce n'est pas le cas, on peut s'arrêter.

TD3-TD4

LES EXERCICES À PRÉPARER

Exercices validés durant les TD3-TD4

2.1 Plus petit élément d'un tableau d'entiers

Définir une fonction *minTableau*, qui à partir d'un tableau *t* d'entiers contenant *N* (*N* > 0) éléments retourne le plus petit élément du tableau. S'il y en a plusieurs, le premier rencontré sera renvoyé.
Remarque : dans cet exercice comme dans la suite on considère que toutes les cases du tableau contiennent une valeur.

2.2 Plus petit et plus grand éléments d'un tableau

Définir une procédure, *calculerMinMax*, qui à partir d'un tableau d'entiers, noté *t*, de *N* éléments donnés, retourne le plus petit et le plus grand éléments du tableau, en faisant attention à ne le parcourir qu'une seule fois.

2.3 Inversion d'un tableau

Soit un tableau *t* d'entiers. Définir une procédure, *inverserTableau*, qui change de place les éléments de ce tableau de telle façon que le nouveau tableau soit une sorte de « miroir » de l'ancien.
Exemple : 1 2 4 6 → 6 4 2 1

2.4 Nombre d'occurrences d'un élément

Définir une fonction *nombreOccurrences*, qui détermine le nombre de fois où un élément noté *élément* apparaît dans un tableau *t* d'entiers donnés.

2.5 Elément le plus fréquent d'un tableau

Soit *t* un tableau d'entiers. On suppose qu'il n'est pas trié. Définir une procédure *déterminerEltPlus-Frequent*, qui détermine l'élément qui apparaît le plus souvent dans le tableau *t*, ainsi que son nombre d'occurrences. Si plusieurs éléments différents répondent au problème, votre algorithme doit en fournir un, quel qu'il soit. Vous ne devez utiliser aucun autre tableau que celui sur lequel vous travaillez.
Un exemple de réutilisation : nombre d'occurrences

2.6 Recherche dichotomique

Définir une fonction, *rechercheDichotomique*, qui détermine par dichotomie le plus petit indice d'un élément *x* (dont on est sûr de l'existence) dans un tableau *t* trié par valeurs croissantes.

2.7 Sous-séquences croissantes

Définir une procédure *déterminerSéquences*, qui à partir d'un tableau d'entiers *t*, fournit le nombre de sous-séquences strictement croissantes de ce tableau, ainsi que les indices de début et de fin de la plus grande sous-séquence.
Soit *t* un tableau de 15 éléments : 1, 2, 5, 3, 12, 25, 13, 8, 4, 7, 24, 28, 32, 1 1, 14.
Les sous-séquences strictement croissantes sont : <1, 2, 5>, <3, 12, 25>, <13>, <8>, < 4, 7, 24, 28, 32>, <11, 14>.
Le nombre de sous-séquences est : 6 et la plus grande sous-séquence est : <4, 7, 24, 28, 32>.

2.8 Transposée d'une matrice carrée

Définir une procédure, *transposer* qui réalise la transposition sur place (en utilisant une seule matrice) d'une matrice $N \times N$ notée *matrice*.

2.9 Produit de deux matrices

Définir une procédure *effectuerProduitMatrice*, qui réalise le produit *c* de la matrice d'entiers $N \times P$ *a* par la matrice d'entiers $P \times M$ *b*.

2.10 Recherche d'un élément dans un tableau de structures

Dans un magasin, à chaque produit sont associés une référence et un prix. Tous les produits sont mémorisés dans un tableau, *produits*, donné. Définir une procédure, *rechercherProduit*, qui pour une référence d'un produit, *référence*, en donnée, retourne le prix du produit ainsi que sa position dans le tableau si le produit est effectivement dans le tableau, et sinon retourne une information booléenne sur l'absence du produit.
Bien entendu, on arrête la recherche dès que le produit est rencontré.

2.11 Moyenne des notes

On désire mémoriser les notes du domaine informatique pour les élèves de première année de **Télécom INT**. Proposez une structure de données qui réponde aux critères suivants :

1. Un module est représenté par un nom et un coefficient.
2. Il faut, pour chaque élève, mémoriser les notes obtenues dans tous les modules du domaine, c'est à dire UX11, AP11 et AM12.
3. Le nombre d'élèves d'une promotion est de 200 au maximum, mais il peut être inférieur.
4. La direction des études souhaite surveiller de manière approfondie les résultats des redoublants.

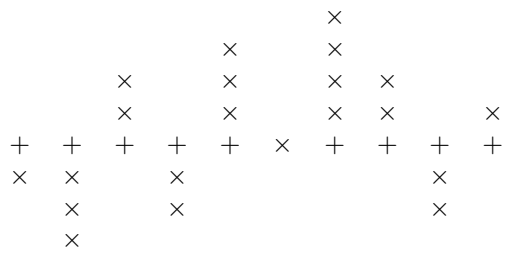
2.12 Histogramme

Définir une procédure *afficherHistogramme*, permettant de représenter un histogramme sur un écran (cf. la figure ci-dessous) à partir d'un tableau *h* de *N* entiers.

i	1	2	3	4	5	6	7	8	9	10
h[i]	-1	-3	2	-2	3	0	4	2	-2	1

Présentation :

L'histogramme sera cadré en haut à gauche de l'écran. Chaque colonne *i* contient un nombre de croix égal à la valeur de*h[i]*. L'axe des abscisses sera représenté comme une ligne de caractères '+' mais si *h[i]* = 0, on affichera un caractère '×'



Remarque :

On fait l'hypothèse que le tableau *h* est compatible avec la taille de l'écran. Par exemple si on a un écran de 25 lignes x 80 colonnes : N $\leq$ 80 et | h[i]-h[j] | $\leq$ 24, $1 \leq i, j \leq N$.

On suppose que l'on dispose des procédures de base suivantes :

1. **afficherCroix ( )** : affiche un caractère $\times$ à l'emplacement du curseur puis avance le curseur
2. **afficherEspace ( )** : affiche un espace à l'emplacement du curseur puis avance le curseur
3. **afficherPlus ( )** : affiche un caractère $+$ à l'emplacement du curseur puis avance le curseur
4. **allerAlaLigne ( )** : place le curseur en début de ligne suivante
5. **effacerEcran ( )** : efface l'écran puis met le curseur en haut à gauche

TD5

LES EXERCICES À PRÉPARER

Exercices validés durant le TD5

3.1 Tri par sélection

Définir une procédure *triSélection* qui effectue le tri par ordre croissant d'un tableau *t* d'entiers donné par la méthode de sélection.
On rappelle que le principe du tri par sélection consiste à échanger le premier et le plus petit élément de la partie non triée du tableau.

3.2 Tri par insertion

Définir une procédure *triInsertion* qui effectue le tri par ordre croissant d'un tableau *t* d'entiers donné par la méthode d'insertion.
On rappelle que le principe du tri par insertion consiste à placer correctement l'élément courant dans la partie déjà triée du tableau.

3.3 Tri à bulles

Définir une procédure *triAbulles* qui effectue le tri par ordre croissant d'un tableau *t* d'entiers donné.
On rappelle que le principe du tri à bulles consiste à comparer deux éléments consécutifs et à les échanger s'ils ne sont pas dans le bon ordre. A l'étape *i*, on parcourt le tableau à partir de la fin de manière à « faire remonter » le plus petit élément en *t[i]*.
On rappelle également qu'on peut améliorer les performances de la méthode en gérant une variable booléenne qui va détecter s'il y a eu au moins un échange à une itération donnée. Intégrez cette amélioration dans votre algorithme.

3.4 Tri de trois couleurs

On considère *N* boules alignées. Elles sont réparties en un nombre quelconque de boules de couleurs vertes, jaunes et rouges et sont disposées dans un ordre quelconque.
En langage algorithmique, on représentera les couleurs par une énumération de constantes :

type **Couleur** = **énumération** VERT, JAUNE, ROUGE

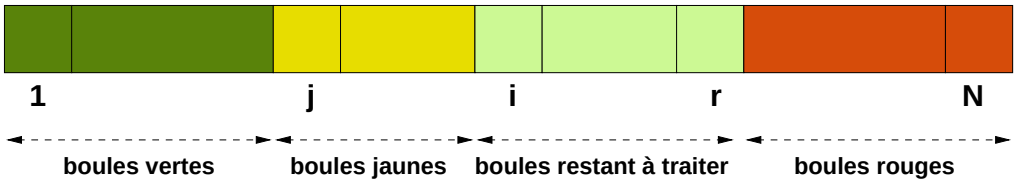
et la ligne de boules par un tableau d'objets de type Couleur :

constante N = 12 // par exemple

type Ligne = **tableau**[1..N] **de** Couleur

On veut trier les boules de telle façon que les boules vertes apparaissent au début du tableau, suivies des boules jaunes et finalement des boules rouges. On veut également que ce tri s'opère en ne parcourant qu'une fois le tableau.
Définir la procédure *trierLigne* qui prend en donnée-résultat un paramètre *ligne* de type *Ligne* et qui réalise le tri spécifié ci-dessus.

Eléments de réponse :
Supposons que la procédure ait déjà réalisé une partie du travail. Nous nous trouvons dans la configuration suivante :



Les boules vertes sont regroupées au début du tableau entre les indice 1 et *j*-1 (*j* comme jaune). Ensuite viennent les boules jaunes entre les indices *j* et *i*-1. Les boules rouges sont, quant à elles, regroupées en fin de tableau entre les indices *r*+1 et *N* (*r* comme rouge).
Il reste à traiter les boules entre les indices *i* et *r* compris et l'on examine *ligne[i]*. On peut distinguer deux cas :

- *i* = *r*+1 : l'algorithme est terminé.
- $i \leq r$: il reste au moins une boule à traiter. Selon la couleur de celle-ci, il faut modifier le contenu de certaines des variables *j*,*i* et *r*. Dans certains cas, il faut échanger deux éléments du tableau.

Enfin vous disposez d'une procédure prédéfinie *échangerCouleurs* qui prend deux paramètres en donnée-résultat de type *Couleur* et qui réalise l'échange. La signature de cette procédure est :

procédure échangerCouleurs (**donnée-résultat** c1, c2 : Couleur)

TD5

C4

LA RÉCURSIVITÉ, LES TRIS RÉCURSIFS

Objectifs du C4

2

- Connaître les principes de construction d'un algorithme récursif
  - Savoir paramétrer le problème
  - Savoir gérer la fin de la descente en récursion
- Connaître les principales méthodes de tri récursifs par comparaison
  - tri rapide
  - tri par fusion

Contenu du C4

3

1 La récursivité

4

1.1 La notion de récursivité

■ **Fonction ou procédure partiellement définie à partir d'elle-même**

fonction = **si** condition **alors**
action1 // action n'exécutant pas d'appel récursif → **terminaison**
sinon
action2 (**fonction**) // action contenant **au moins**
fsi // un appel récursif de la fonction

5

■ **Applications**

- ♦ **Implémentation de relations de récurrence**
- ♦ **Méthodes de tri récursives**
- ♦ **Parcours de structures naturellement récursives** :
  - ► liste, arbre, graphe

■ **Exécution généralement coûteuse : temps, espace mémoire**

– **Procédures récursives**
  – souvent dans la définition d'une procédure récursive, quand on atteint la fin de la descente en récursion, il n'y a rien à faire
  – la structure de ces procédures sera alors :

procédure = **si** condition **alors**
action (**procédure**) // action contenant **au moins**
fsi // un appel récursif de la procédure

– **Applications**
  – On vous proposera de définir des procédures récursives sur des tableaux comme la recherche du plus grand et du plus petit élément d'un tableau
  – l'objectif est purement pédagogique ; le codage de ces algorithmes vous permettra de mieux maîtriser le passage de paramètre
  – mais dans un contexte opérationnel, l'utilisation de la récursivité est à prohiber pour ce genre de problème

1.2 Méthode de conception

6

- **Paramétrer le problème**
  - **Selon sa taille**
- **Gérer la fin de la *descente* en récursion**
  - **Rechercher le ou les cas triviaux** qui se résolvent sans appel récursif
- **Décomposer le cas général**
- **Exemple** : calcul de n !
  - **Taille du problème** : n
  - **Cas triviaux** : $0! = 1$, $1! = 1$
  - **Décomposition** : $n! = n \times (n-1)!$

- **Définition de la fonction** *factorielle*

```
fonction factorielle ( n : Naturel ) : Naturel

    si n ≤ 1 alors
        retourner 1
    sinon
        retourner n × factorielle ( n-1 )
    fsi

ffct factorielle
```

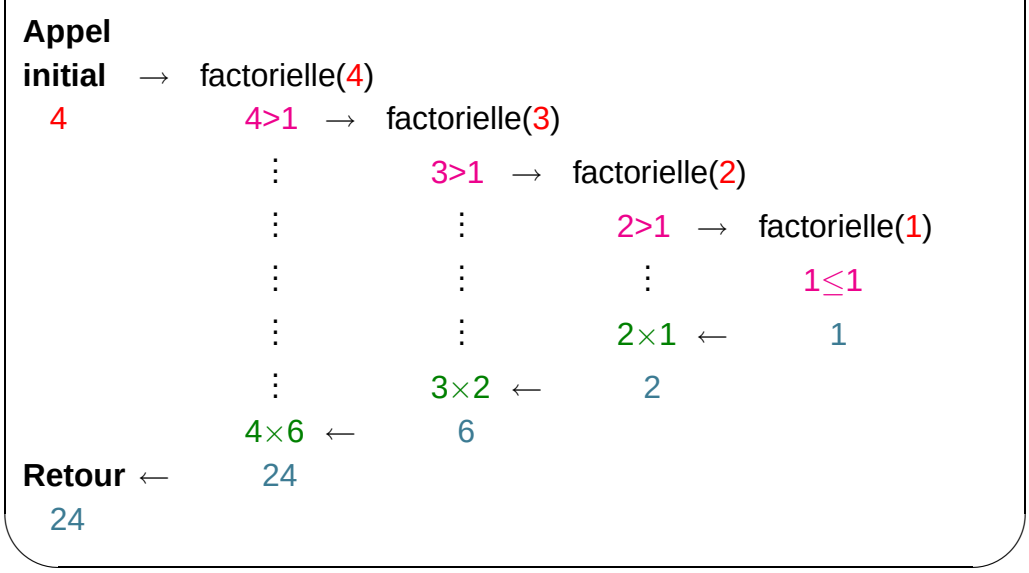
– **Paramétrage du problème**
  – concernant l'implémentation de relations de récurrence, il n'y a pas de difficulté
  – pour les procédures récursives opérant sur un tableau, la taille du problème correspond au nombre d'éléments du tableau
    – à chaque appel récursif, le tableau sera « divisé » en deux sous-tableaux
    – ceci est réalisé « logiquement » grâce à deux paramètres *début* et *fin*
    – la fin de la descente en récursion se produira quand les sous-tableaux seront réduits à un élément (*début* $=$ *fin*), voir « vides » (*début* $>$ *fin*)

– **Appel initial de la fonction** *factorielle*

```
fonction principale ( ) : CodeRetour

    nb : Naturel

    saisir ( nb )
    afficher ( nb, "! = ", factorielle ( nb ) )
    allerAlaligne ( )

    retourner OK

ffct principale
```

7

1.3 Enchaînement des appels récursifs -



- **Récursivité terminale**
  - une **fonction** *f* (on ne parle pas ici de procédures) est *récursive terminale* quand chaque appel récursif est de la forme **retourner f (. . .)**
  - la valeur retournée est directement issue d'un appel récursive sans qu'il n'y ait aucune opération sur cette valeur
  - certains compilateurs C détectent les fonctions récursives terminales et les transforment en schéma itératif pour optimiser l'utilisation de l'espace mémoire
  - la fonction *factorielle* telle que présentée ici n'est pas récursive terminale puisque les multiplications sont réalisées lors de la « remontée » des appels récursifs
  - voici une autre définition de la fonction *factorielle* qui est récursive terminale :

```
fonctionfactorielle ( n , fact : Naturel ) : Naturel
        si n ≤ 1 alors
            retourner fact
        sinon
            retourner factorielle ( n-1, n × fact )
        fsi
    ffct factorielle

    fonction principale ( ) : CodeRetour
        nb : Naturel
        saisir ( nb )
        afficher ( nb, "! = ", factorielle ( nb, 1 ) )
        allerAlaligne ( )
        retourner OK
    ffct principale
```

2 Les tris récursifs

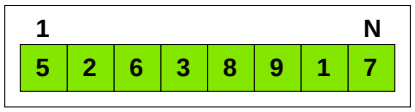
8

2.1 Spécification du problème

9

- **Données**
  - une séquence de *N* éléments comparables $< a_1, a_2, \ldots, a_N >$
- **Résultats**
  - une permutation $< a'_1, a'_2, \ldots, a'_N >$ de ces *N* éléments telle que : $a'_1 \leq a'_2 \leq \ldots \leq a'_N$
- **Eléments à trier**
  - **clé** : valeur de référence pour le tri
    les clés appartiennent à un ensemble totalement ordonné
  - **données satellites** : a priori permutées en même temps que la clé
- **Hypothèses simplificatrices**
  - élément $=$ clé $=$ entier (pas de données satellites)
  - séquence $=$ t : **tableau** [1. . .N] **de**

Entier



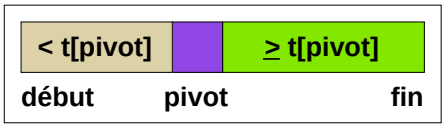
- **Contrainte** : éviter d'utiliser un tableau auxiliaire

10

2.2 Tri rapide

■ **Diviser t [début. . .fin]**

♦ partitionner t [début. . .fin] en trois parties : l'élément « **pivot** » t [pivot] et deux **sous-tableaux** non vides t [début. . .pivot-1] et t [pivot+1. . .fin] tels que :
$\forall x \in t[debut \ldots pivot-1], \forall y \in t[pivot+1 \ldots fin] \quad : \quad x < t[pivot] \leq y$



■ **Résoudre**

♦ les sous-tableaux t [début. . .pivot-1] et t [pivot+1. . .fin] sont triés par des **appels récursifs** à la procédure *triRapide*

■ **Combiner**

♦ il n'y a rien à faire car les sous-tableaux sont déjà triés

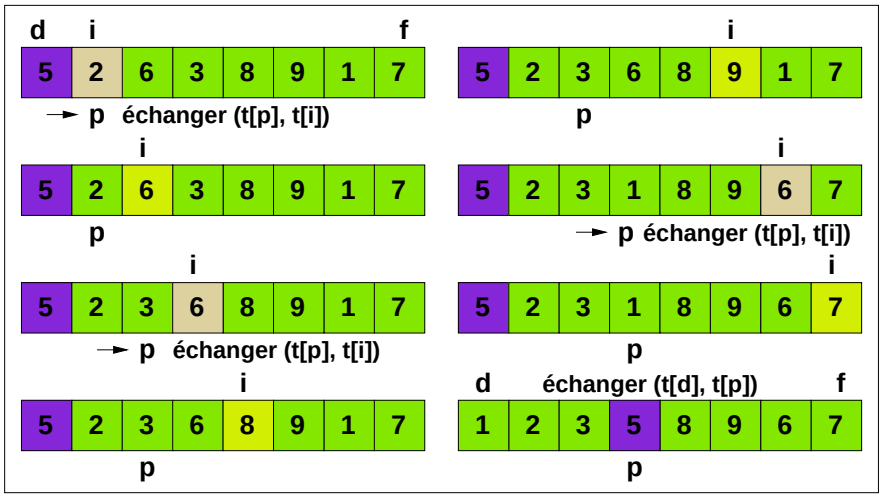
– **Principe**
  – Une étape du tri rapide place un élément (le *pivot*) à sa **position finale** dans le tableau tout en **répartissant** les autres éléments dans la **bonne « moitié »** du tableau par rapport à ce *pivot*.

– **Choix du pivot**
  – dans notre implémentation, nous choisirons le **premier** élément du tableau.
  – **dégénérescence** si l'élément pivot est égal à la **plus petite** ou la **plus grande** valeur du tableau.
  – choix de l'élément **médian** → complexité accrue

– **Procédure** *partitionner*
  – c'est le composant essentiel de l'algorithme puisque tout le travail se fait là.

11

2.3 Partitionnement (tri rapide)

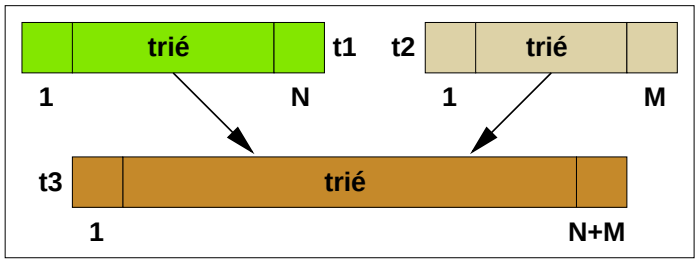


– **Eléments de compréhension de la procédure** *partitionner*
  – l'élément *pivot* a pour valeur 5 dans l'exemple ; il ne sera déplacé qu'à la fin de la procédure *partitionner*, et le sous-tableau de gauche se construit donc provisoirement décalé d'une position à droite.
  – la variable *i* a pour valeur initiale *d*+1 ; elle est incrémentée à chaque itération.
  – la variable *p (pivot)* a pour valeur initiale *d* ; elle mémorise la position du pivot et doit donc être incrémentée chaque fois qu'un élément inférieur au pivot est rencontré.
  – à chaque itération :
    – si *t[i]* est strictement inférieur à la valeur du *pivot*, le sous-tableau de gauche doit contenir cet élément supplémentaire.
    – la variable *p* est donc incrémentée d'une unité. A la suite de quoi *t[p]* est maintenant un élément qui a priori n'appartient pas au sous-tableau de gauche et doit donc laisser sa place à *t[i]*.
    – on échange ensuite *t[i]* et *t[p]* pour placer *t[i]* dans le sous-tableau de gauche.
  – la procédure se termine en échangeant *t[d]* et *t[p]*

12

2.4 Tri par fusion

- **Diviser**
  - **diviser** le tableau de N éléments en **deux** sous-tableaux de longueur N/2 (à un près)
- **Résoudre**
  - trier les sous-tableaux par des **appels récursifs** à la procédure *triFusion* jusqu'à obtenir des sous-tableaux réduits à un élément (donc forcément triés)
- **Combiner**
  - **fusionner** les sous-tableaux triés



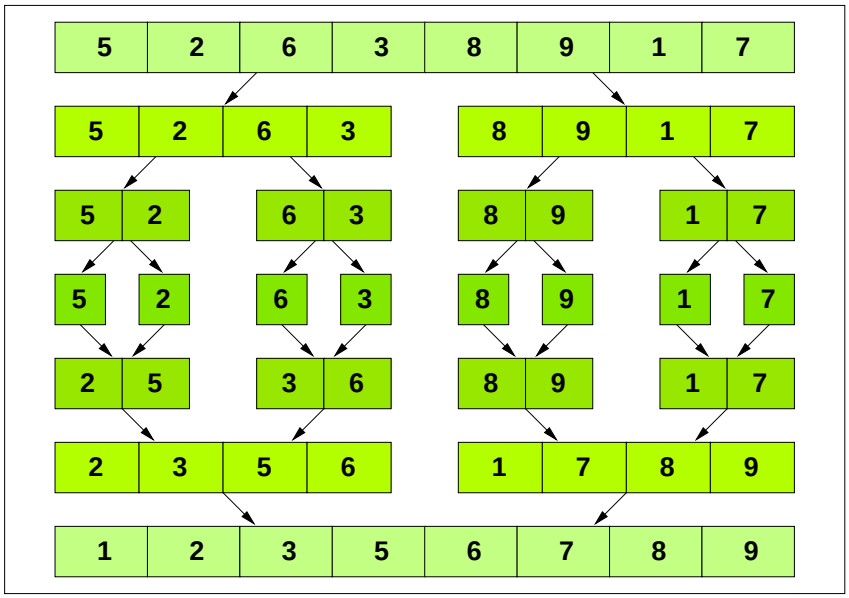
– **Procédure** *fusionner*
  – tout le travail de l'algorithme se situe **ici**.
  – le principe est très simple :
    – on compare le premier élément du tableau *t1* avec le premier élément du tableau *t2*
    – on place le minimum des deux en début du tableau *t3*
    – on passe au suivant dans le tableau où l'on a sélectionné le minimum
    – on passe également au suivant dans le tableau *t3*
    – on recommence jusqu'à ce qu'un des tableaux *t1* ou *t2* soit « épuisé »
  – il reste à « vider » le tableau non épuisé en plaçant les éléments restant à la fin du tableau *t3*.

– **Adaptation à la procédure** *triFusion* **triant le tableau** *t*
  – *t1* correspond à la partie de *t* comprise entre deux indices *début* et *milieu*.
  – *t2* correspond à la partie de *t* comprise entre *milieu*+1 et l'indice *fin*.
  – *t3* est un tableau temporaire de longueur *fin-début*+1.
  – à la fin de la procédure *fusionner*, il faut recopier les éléments du tableau *t3* dans *t* entre les indices *début* et *fin*.
  – pour cette raison cet algorithme, bien qu'étant de complexité optimale, est en pratique peu performant.

– Avec cette méthode de tri, on ne peut éviter le recours à un **tableau auxiliaire**.

13

2.5 Déroulement du tri par fusion -



2.6 Complexité des algorithmes de tri

- **Evaluée en nombre de comparaisons** (tris comparatifs)
- **Rappel** borne inférieure de complexité : $\Omega(n \log_2 n)$ dans le pire des cas

14

nombre de comparaisons	au minimum	en moyenne	au maximum
tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$
tri par insertion	$O(n)$	$O(n^2)$	$O(n^2)$
tri à bulles optimisé	$O(n)$	$O(n^2)$	$O(n^2)$
tri rapide	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
tri par fusion	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

– **Certaines précisions données ci-dessous dépendent de choix algorithmiques**

– **tri par sélection** : dans tous les cas, il y a $N - i$ comparaisons à faire pour trouver le minimum dans le sous-tableau *t[i...N]* ; ce qui donne : $\sum_{i=1}^{N-1} N - i = N(N-1)/2$ comparaisons.

– **tri par insertion**
  – **au minimum** (le tableau initial est trié par ordre croissant) : 1 comparaison à faire à chaque itération ; ce qui donne $N - 1$ comparaisons.
  – **au maximum** (le tableau initial est trié par ordre décroissant) : $i - 1$ comparaisons à faire à chaque itération ; ce qui donne : $\sum_{i=2}^{N} i - 1 = N(N-1)/2$ comparaisons.

– **tri à bulles** (version optimisée)
  – **au minimum** (le tableau initial est trié par ordre croissant) : $N - 1$ comparaisons suffisent à le vérifier.
  – **au maximum** (le tableau initial est trié par ordre décroissant) : $N - 1$ comparaisons placent le plus petit élément au début du tableau, $N - 2$ comparaisons placent le suivant en deuxième position, et ainsi de suite. d'où $N(N-1)/2$ comparaisons.

– **tri rapide**
  – comme nous l'avons déjà mentionné, l'algorithme de tri rapide voit ses performances s'effondrer si la procédure *partitionner* ne réalise pas une partition relativement équilibrée du tableau initial du fait d'un trop grand éloignement du *pivot* par rapport à le médiane.

– **tri par fusion**
  – sa complexité est optimale, mais dans la réalité, il a des performances médiocres.

TD6-TD7

LES EXERCICES À PRÉPARER

Exercices validés durant les TD6-TD7

4.1 Calcul de C(n,p)

Définir une fonction récursive *cnp*, qui en fonction des entiers positifs *n* et *p*, retourne le nombre de combinaisons de *p* éléments parmi *n*.

4.2 Terme de la suite de Fibonacci

Définir une fonction récursive *termeFibonacci*, qui calcule la valeur du terme de rang *n* de la suite de Fibonacci (→ exercice 1.9 des TD1-TD2).

4.3 PGCD : Algorithme d'Euclide

L'algorithme d'*Euclide* détermine le plus grand diviseur commun de deux entiers positifs *a* et *b*. On supposera que *a* > *b*. L'algorithme dans sa version itérative fonctionne de la manière suivante :
A chaque itération, on calcule le reste *r* de la division entière de *a* par *b*.

$$a = b\,q + r, r < b$$

Tout diviseur commun de *a* et *b* divise aussi *r* = *a* - *bq*, et réciproquement tout diviseur commun de *b* et *r* divise aussi *a* = *bq* + *r*. Donc le calcul du PGCD de *a* et *b* se ramène à celui du PGCD de *b* et *r* ; et on peut recommencer sans craindre une boucle sans fin, car les restes successifs sont strictement décroissants. Le dernier reste non nul obtenu est le PGCD cherché.
Voici un petit exemple avec *a* = 84 et *b* = 30 :

a		*b*		*r*
80	$= 2 \times$	30	$+$	24
30	$= 1 \times$	24	$+$	6
24	$= 4 \times$	6	$+$	0

Le PGCD vaut 6.

- Définissez la fonction **itérative** *pgcd* qui prend en paramètres *a* et *b* de type Naturel et qui renvoie un naturel égal au PGCD de ces deux nombres.
- Définissez la fonction **récursive** *pgcd* de même signature.

4.4 Zéro d'une fonction par dichotomie

Définir une fonction récursive *zéroFonction*, qui calcule le zéro d'une fonction *f(x)* sur l'intervalle [*a*, *b*], avec une précision *epsilon*. La fonction *f* et les valeurs réelles *a*, *b* et *epsilon* sont données.
Soit *f* une fonction continue monotone sur l'intervalle [*a*, *b*], où elle ne s'annule qu'une seule et unique fois. Pour trouver ce zéro, on procède par dichotomie, c'est-à-dire que l'on divise l'intervalle de recherche par deux à chaque étape. Soit *m* le milieu de [*a*, *b*]. Si *f(m)* et *f(a)* sont de même signe, le zéro recherché est dans l'intervalle [*m*, *b*], sinon il est dans l'intervalle [*a*, *m*].

4.5 Inversion d'un tableau

Soit un tableau *t* d'entier. Définir une procédure récursive, *inverserTableau*, qui change de place les éléments de ce tableau de telle façon que le nouveau tableau *t* soit une sorte de « miroir » de l'ancien.

4.6 Recherche d'un élément dans un tableau

Définir une fonction récursive *estPrésent*, qui retourne VRAI si un élément donné, noté *élément* est présent dans un tableau, *t*, d'entiers et FAUX sinon.
Etudier successivement les cas où *t* n'est pas un tableau trié et où *t* est un tableau trié par valeurs croissantes.

4.7 Plus petit et plus grand éléments d'un tableau

Définir de deux façons différentes une procédure récursive, *calculerMinMax*, qui détermine le plus petit et le plus grand éléments d'un tableau *t* d'entiers. On pourra utiliser les principes suivants :

- Le plus grand élément est le maximum entre le premier élément et le maximum du reste du tableau.
- Le plus grand élément est le maximum entre le plus grand de chaque moitié du tableau (méthode dichotomique).

4.8 Somme des éléments d'une matrice d'entiers

Définition du type *Matrice* :

constante NB_LIGNES = 5
constante NB_COLONNES = 7 // par exemple

type Matrice = **tableau**[1..NB_LIGNES][1..NB_COLONNES] **de** Entier

Définir la fonction récursive *sommeMatrice* qui prend comme paramètre *matrice* de type *Matrice* et qui renvoie un entier égale à la somme des éléments de cette matrice.

4.9 Déterminant d'une matrice carrée d'entiers

Soit la constante *NB_MAX* égale par exemple à 10 :

constante NB_MAX = 10 // par exemple

Nous allons maintenant définir le type *MatriceCarrée* comme une structure composée de deux champs. Le premier champ *t2D* est un tableau à deux dimensions de *NB_MAX* lignes et *NB_MAX* colonnes d'entiers. Le deuxième champ est un naturel *nb* qui indique le nombre de lignes et de colonnes réellement utilisées dans le tableau *t2D*. Les indices « utiles » vont donc de 1 à *nb* inclus et le champ *nb* peut varier de 1 à *NB_MAX* inclus.

type MatriceCarrée = **structure**
t2D : **tableau**[1..NB_MAX][1..NB_MAX] **de** Entier
nb : Naturel
fstruct

Définir la fonction récursive *déterminant* qui prend un paramètre *matrice* de type *MatriceCarrée* et qui renvoie un entier, le déterminant de cette matrice.
La méthode algorithmique à utiliser est le développement suivant une ligne ou une colonne. Cette méthode est loin d'être efficace, mais il s'agit ici simplement d'un exercice d'algorithmique. Vous prendrez impérativement la première colonne.
Il faudra définir une fonction *matriceRéduite* qui prend en paramètre un objet *mat* de type *MatriceCarrée* et un naturel *n* et dont la « valeur » de retour est un objet de type *MatriceCarrée* dont le champ *nb* a été décrémenté de une unité par rapport à la matrice carrée passée en paramètre. Son champ *t2D* est obtenue

en éliminant du champ *t2D* de la matrice passée en paramètre la première colonne et la ligne d'indice *n*. Voici la signature de cette fonction :

fonction matriceRéduite (mat : MatriceCarrée, n : Naturel) : MatriceCarrée

TD8

LES EXERCICES À PRÉPARER

Exercices validés durant le TD8

5.1 Tri rapide

On rappelle que le principe du tri rapide est basé sur le modèle « diviser pour résoudre ». Pour trier un sous-tableau *t*[*début*. . .*fin*] le principe est le suivant :

1. **Diviser** : on choisit au hasard un élément du tableau *t*[*début*. . .*fin*] que l'on nommera *pivot*. Pour remplacer le hasard, on se contente généralement de prendre le premier élément *t*[*début*]. On partitionne (en le réarrangeant) le tableau *t*[*début*. . .*fin*] en deux sous-tableaux non vides *t*[*début*. . .*milieu*] et *t*[*milieu+1*. . .*fin*], de telle façon que, le *pivot* étant positionné à sa place définitive, tous les éléments strictement inférieurs au *pivot* soient placés avant lui (dans les indices inférieurs), et tous les éléments supérieurs ou égaux au *pivot* soient placés après lui (dans les indices supérieurs).
2. **Résoudre** : le *pivot* étant à sa place définitive, il reste à trier (de la même façon) les deux sous-tableaux placés avant et après lui.
3. **Combiner** : les sous-tableaux étant triés, il n'y a rien à faire.

Définir les procédures *triRapide* et *partitionner*.

5.2 Tri par fusion

On rappelle que l'algorithme du tri par fusion peut se décrire dans le paradigme « diviser pour résoudre » de la manière suivante :

1. **Diviser** : diviser la séquence de n éléments à trier en deux sous-séquences de n/2 éléments.
2. **Résoudre** : trier les deux sous-séquences récursivement.
3. **Combiner** : fusionner les deux sous-séquences triées.

Définir la procédure *triFusion*.

C5-C6

LE LANGAGE C ET LE PASSAGE DE PARAMÈTRES

Objectifs des C5 et C6

2

- Les premiers pas dans la programmation en C
  - Savoir définir et intialiser des variables
  - Connaître les principales constructions syntaxiques du langage
  - Etre capable de créer un fichier source en C, de le compiler et de l'exécuter
- Aborder la problématique du passage de paramètres
  - Passage par valeur
  - Passage par adresse
- Remarque : les transparents qui suivent vous apporteront les éléments les plus importants du langage. Vous trouverez en annexe de ce polycopié une présentation plus détaillée du langage. Vous trouverez également sur le site AP11 des supports de référence du langage.

3

Contenu des C5 et C6

1 Le langage C

4

1.1 Structure d'un programme élémentaire -

5

monPremierProgramme.c

```
/* Votre premier programme
   Il affiche... Bonjour */

#include <stdlib.h>  // pour la constante
                     // symbolique EXIT_SUCCESS
#include <stdio.h>   // pour le prototype de
                     // la fonction printf
int main ( ) {
   printf ( "Bonjour\n" );
   return EXIT_SUCCESS;
}
```

– **L'inclusion de fichiers d'en-tête** (directive **`#include`**) :
Le langage C se réduit au strict minimum. Il nécessite un environnement de programmation : un ensemble de collections de « sous-programmes » pré-compilés et appelés *bibliothèques* ou *librairies*. Les fichiers d'en-tête réalisent l'interface entre les bibliothèques et le programme utilisateur. Le fichier *stdio.h*, par exemple, contient la **déclaration** (type de la valeur de retour, paramètres et leur type) de la fonction *printf* qui permet l'affichage dans la fenêtre *shell* dans laquelle est lancée l'exécution.
Le compilateur peut ainsi vérifier si l'appel de la fonction est cohérent avec sa déclaration.

– **La fonction main** : Elle correspond à ce que nous avons appelé fonction *principale* dans notre langage algorithmique.
Elle renvoie une valeur entière à l'appelant, le *shell* à partir duquel a été lancée l'exécution (valeur affichable immédiatement par la commande **`echo $?`**). Ainsi, l'instruction **`return EXIT_SUCCESS;`** correspond à l'action « **retourner** OK ».
Le corps de la fonction main (comme de toute autre fonction) est délimité par les caractères **`{`** et **`}`**.

– **Marque de fin d'instruction** : chaque instruction se termine par le caractère **`;`**.

6

1.2 Types prédéfinis du langage -

- **Les types « entiers »**
  - **`short`**, **`int`**, **`long`**, **`long long`** : dépendants de la plate-forme
  - **Généralement** : **`short`** sur 2 octets, **`int`** sur 4 octets, **`long`** sur 4 ou 8 octets, **`long long`** sur 8 octets
  - Préfixe **`unsigned`** : valeur positive ou nulle
- **Les types « réels »**
  - **`float`** sur 4 octets, **`double`** sur 8 octets : normalisés IEEE 754
- **Le type Caractère**
  - **`char`** sur 1 octet
- **Pas de type Booléen** (sauf utilisation de la norme C99 $\rightarrow$ notes)
  - Utilisation de valeurs entières ou de constantes énumérées
  - En C, une valeur nulle $\Longleftrightarrow$ FAUX, une valeur non nulle $\Longleftrightarrow$ VRAI
- **Pas de type Chaîne**
  - Utilisation de tableaux de caractères

– **Les types « entiers »** : Le langage C ne spécifie pas le nombre d'octets nécessaires pour les stocker en mémoire
  – L'opérateur **`sizeof`** retourne le nombre d'octets utilisés pour stocker une variable d'un type donné.
  – Ce qu'assure le langage :
  `sizeof(short)` $\leq$ **`sizeof(int)`** $\leq$ **`sizeof(long)`** $\leq$ **`sizeof(long long)`**
  – Le type **`unsigned int`** peut correspondre au type *Naturel* de notre langage algorithmique.

– **Les types normalisés `float`** et **`double`** sont représentés en *virgule flottante* (mantisse et exposant)
Sauf si vous avez à manipuler un tableau de plusieurs dizaines de milliers de réels, utilisez de préférence le type **`double`** pour deux raisons :
  – l'évaluation d'une expression invoquant le contenu de variables d'un type réel se fait toujours dans le type **`double`** pour améliorer la précision.
  – Les fonctions de la bibiothèque mathématique prennent systématiquement un ou deux arguments de type **`double`** et renvoient une valeur de type **`double`**.

– **Le type `char`** est signé, la plage de valeurs va de -128 à +127. Les variables de ce type stockent la plupart du temps des codes *ASCII* (sur 7 bits) ou *ISO-8859-1* (sur 8 bits) qui représentent les caractères affichables. Mais on peut utiliser ce type à d'autres fins.
Par exemple, pour stocker une valeur de pixel en niveaux de gris allant de 0 à 255, nous utiliserons le type **`unsigned char`**.

7

1.3 Variables, affectation -

- **Définition et initialisation de variables**
  - **Définition** (en début de bloc)
    - `int i;`
    - `double x, y;`
  - **Initialisation** (facultative)
    - `int n = 3;`
    - `double v = 5, w = 7.9;`
  - **Identificateur**
    - Alphabet : « **`a-z`**, **`A-Z`**, **`0-9`**, _ », distinction majuscules/minuscules
    - Commence par une lettre ou le caractère '_'
  - **Visibilité**
    - limitée au bloc où la variable a été définie
- **L'affectation**
  - **L'opérateur =**
    - `i = n - 2;`
    - `x = ( v + w ) / 3.;`

– **La définition d'une variable** correspond à :
  – Une association indentificateur/objet
  – Une réservation mémoire pour stocker cet objet

  L'initialisation est facultative, mais prudence car une variable non initialisée contient en général une valeur indéterminée.
  Quant à la visiblité, nous nous contenterons de définir des variable *locales* aux fonctions ; elles ne seront donc visibles qu'à l'intérieur de ces fonctions.
  La norme *C99* du langage est plus laxiste ; les variables ne sont pas forcément définies en début de bloc.

– **L'affectation** : ce n'est pas un opérateur au sens strict du mot. Le langage C en a fait un. Ce qui permet d'écrire des choses comme :
  `x = y = 1;` // `x = ( y = 1);` (la valeur d'une expression d'affectation est la valeur affectée)
  mais d'autres moins lisibles comme :
  `x = 2 * ( y = z + 1);`
  N'utilisez pas ces possibilités pour démarrer l'apprentissage du langage.

– **L'opérateur de succession** permet d'enchaîner l'évaluation de plusieurs expressions, donc plusieurs affectations dans une même instruction :
  `x = 1, y = 2;` ⟺ `x = 1; y = 2;`

1.4 Opérateurs et expressions -

8

- **Opérateurs**
  - **Opérateurs arithmétiques**
    - ordinaires : **+**, **-**, *****, **/**, **%**
    - combinés avec l'opérateur d'affectation : **+=**, **-=**, ***=**, **/=**
    - incrémentation/décrémentation : **++**, **--**
  - **Opérateurs de comparaison**
    - **<**, **<=**, **>**, **>=**, **==**, **!=**
  - **Opérateurs logiques**
    - **!**, **&&**, **||**
  - **Mais aussi**
    - l'opérateur **sizeof**
    - les opérateurs de bas niveau : masquage, décalage
- **Expression**
  - **Séquence, éventuellement parenthésée et syntaxiquement correcte, d'opérateurs et d'opérandes**
    - **(v + w) / 3.;**
    - Pour le parenthésage, voir la précédence des opérateurs

– **Opérateurs arithmétiques**
  – **Division** : Il n'existe pas en C d'opérateur spécifique pour la division entière comme dans d'autres langages.
  Si l'expression n'invoque que des opérandes « entiers », la division sera entière. S'il existe au moins un opérande de type « réel » (variable de type **float** ou **double**, constante suffixée par un point), l'expression sera évaluée dans le type **double**. En effet, si les deux opérandes d'un opérateur ne sont pas de même type, l'opérande du type le plus pauvre est promu dans le type le plus riche. Mais si le résultat est affecté à une variable « entière », il y aura troncature pour stocker un entier.
  – **Ces trois instructions sont équivalentes** : **x = x + 1;**, **x += 1;**, **x++;**
  – **Précisions sur l'incrémentation/décrémentation** :
  Dans une expression, on distingue pré-incrémentation et post-incrémentation (idem pour la décrémentation). Un petit exemple :

```
int i = 1, j, k;
j = i++; // affectation du contenu de i à j, puis incrémentation de i
k = ++i; // incrémentation de i, puis affectation du contenu de i à k
```

  Finalement la variable *j* contient la valeur 1, les variables *i* et *k* la valeur 3.

– **Opérateurs de comparaison** : la confusion entre l'opérateur d'affectation **=** et l'opérateur d'égalité **==** est de mise chez les débutants.

– **Opérateurs de bas niveau** :
  – **négation** bit à bit : **~**, **et** bit à bit : **&**, **ou** bit à bit : **|**, **ou exclusif** bit à bit : **^**
  – **décalage** à gauche : **<<**, **décalage** à droite : **>>**

1.5 Définition de bloc -

9

```
{  int i = 2;    // un premier bloc
   int j;

   j = i + 2;

   {  int j;     // un deuxième bloc ; j est local
                 // au bloc et masque le j extérieur
      j = i + 1; // mais i est visible dans le bloc
   }             // imbriqué ; le j local vaut 3

   j++;          // j contient la valeur 5
}                // l'autre n'existe plus
```

– **Action composée du langage algorithmique → bloc en langage C**
  – commence par le caractère **{**
  – se termine par le caractère **}**

  – en général derrière un schéma alternatif ou itératif

– Les variables définies dans un bloc ne sont **visibles** qu'à l'intérieur de ce bloc.

– Elles doivent impérativement être définies en **début de bloc**, i.e. après la première instruction du bloc ne doit plus figurer de définition de variable.
  Remarque : cette restriction est levée en langage C++ ainsi que dans la norme C99. Bien que les dernières versions de *gcc* reposant sur le compilateur C++ tolèrent ces constructions, elles sont très fortement déconseillées pour leur manque de lisibilité.

1.6 Schéma alternatif : instruction if -

10

```
if ( a == 0 ) { // équation du premier degré
  if ( b != 0 )
    racine = -c/b;
  else {
    // ...
  }
}
else {          // équation du second degré
  delta = b*b - 4*a*c;
  // ...
}
```

– **Encore une fois** : attention à la confusion entre l'opérateur d'affectation **=** et l'opérateur d'égalité **==** : l'expression d'affectation (a = 0) a une valeur nulle et serait interprétée comme la valeur booléenne FAUX par le compilateur. Le bloc correspondant ne sera **jamais** exécuté.
De plus le contenu de la variable *a* est modifié, ce qui n'était certainement pas voulu.
L'écriture équivalente **`if ( 0 = a )`** évite à coup sûr le problème.

– Remarquez que la définition d'un bloc **`{...}`** n'est pas systématique.

```
if( b != 0)
  racine = -c/b;
```

– **Respectez la syntaxe** :
  – Le mot réservé ne s'écrit pas **`IF`**, ni **`If`** mais **`if`**.
  – La condition est **parenthésée**.

– **L'instruction `switch`** qui correspond au schéma **cas où** de notre langage algorithmique : à étudier par vous-même dans l'annexe « L'essentiel du langage C ».

1.7 Schéma itératif : instructions while, do/while et for -

11

```
#define N 10

int i;

i = 1;
while( i <= N ) {
 printf("%d",i);
 i++;
}
```

```
#define N 10

int i;

i = 1;
do {
 printf("%d",i);
 i++;
} while ( i<=N );
```

```
#define N 10

int i;

for(i=1;i<=N;i++)
 printf("%d",i);
```

- **Définition de constantes symboliques**
  - directive **`#define N 10`**
  - toutes les occurrences de la chaîne "N" seront remplacées par la valeur associée dans le reste du fichier source : le compilateur lira alors **`while( i <= 10 )`**
  - une directive ne se termine pas par le caractère "**`;`**" ; **`#define N 10;`** entraînera dans le source l'instruction **`while( i <= 10; )`**, ce qui provoquerait une erreur à la compilation
  - à ne pas confondre avec les variables « **constantes** » : **`const int N = 10;`**
    → même effet mais avec utilisation d'une zone mémoire

- **Une écriture plus concise du premier programme**

```
while( i <= N )
  printf( "%d", i++ );
```

  la définition d'un bloc n'est plus obligatoire

- **L'instruction `for`**
  - elle est bien plus riche que le schéma **pour** de notre langage algorithmique
  - exemple : l'affichage des puissances de deux

```
#define MAX 1024
int x = 2;)
for ( ; x <= MAX; x*=2 )
  printf( "%d", x );
```

  le champ initialisation a été omis car la variable x a été initialisée à la définition

12

1.8 Tableau à une dimension -

- **Définition**
  - `#define NB_ELEMENTS 5`
  - `int tab[NB_ELEMENTS]; // valeurs a priori indéterminées`
- **Définition et initialisation**
  - `int tab[NB_ELEMENTS] = {3, 8, -9, 0, 5};`
  - `int tab[] = {3, 8, -9, 0, 5};`
  - `char message[] = "Bonjour"; // 8 caractères dont '\0'`
  - `char message[] = {'B','o','n','j','o','u','r','\0'};`
- **Accès aux éléments**
  - Dans le langage C, les indices d'un tableau débutent en 0
  - `tab[i]` = i+1ème élément du tableau
  - `tab[3] = 2; // il n'y a plus d'élément nul dans tab`

– **Le nombre d'éléments d'un tableau**
  – utilisez systématiquement une constante symbolique
  – elle sera certainement invoquée plusieurs fois dans le programme
  – si vous voulez changer la taille du tableau, il suffit de modifier une seule ligne dans le programme : `#define NB_ELEMENTS 10`

– **Tableaux de caractères**
  – C'est la convention du langage C ; une « chaîne » de caractères se termine par le caractère `'\0'` qui correspond à une valeur nulle.
  – à noter que la première syntaxe d'initialisation est bien plus sympathique que la seconde.
  – le langage C ne gère rien d'autre ; heureusement, l'environnement de programmation nous propose des fonctions qui sont définies dans la bibliothèque standard et déclarées dans le fichier d'en-tête *string.h* :
    – **`strlen`** renvoie le nombre de caractères d'une chaîne (non-compris le caractère conventionnel de terminaison)
    – **`strcmp`** permet de comparer deux chaînes de caractères
    – **`strcpy`** permet de copier le contenu d'une chaîne dans une autre chaîne . . .
  – consultez le manuel en ligne : `$ man string`

– **Accès aux éléments d'un tableau**
  – là aussi, une source d'erreurs dans la transcription d'un algorithme : les indices vont de `0` à `NB_ELEMENTS-1` et non de `1` à `NB_ELEMENTS`

```
int i;
for ( i=0; i < NB_ELEMENTS; i++ )
      printf( "%2d", tab[i] );
```

1.9 Tableau à deux dimensions -

13

- **Définition**
  - `#define NB_LIGNES 3`
  - `#define NB_COLONNES 4`
  - `double matrice[NB_LIGNES][NB_COLONNES];`
- **Définition et initialisation**
  - `double matrice[NB_LIGNES][NB_COLONNES] =`

```
{{3.5, 4.2, 8.4, 7.3},
 {6.3, 2.1, 5.9, 9.3},
 {8.9, 7.3, 1.5, 6.4}};
```

- **Accès aux éléments**
  - `matrice[ligne][colonne] = 10.;`

– **Affichage des éléments de la matrice**

```
int ligne, colonne;

for ( ligne=0; ligne < NB_LIGNES; ligne++ ){
  for ( colonne=0; colonne < NB_COLONNES; colonne++ )
    printf( "%3.1f ", matrice[ligne][colonne] );
  printf( "\n" );
}
printf( "\n" );
```

14

1.10 Types scalaires et tableau -

- **Définition de types scalaires**
  - `typedef unsigned int Naturel; // type synonyme`
  - `typedef enum {FAUX, VRAI} Booleen;`
  - **Utilisation**
    - `Naturel nombre;`
    - `Booleen encore = VRAI;`
- **Définition de types tableau**
  - `#define N 10`
  - `#define M 20`
  - `typedef int Vecteur[N];`
  - `typedef int Matrice[N][M];`
  - **Utilisation**
    - `Vecteur unVecteur;`
    - `Matrice uneMatrice;`

– **Rappel** : afin de les distinguer facilement des noms de variables, utilisez un identificateur commençant par une majuscule pour les types.

– **Utilisation de constantes énumérées pour définir le type `Booleen`**
  – **`typedef enum {FAUX=0, VRAI=1} Booleen;`**
  ou plus simplement
  `typedef enum {FAUX, VRAI} Booleen;`
  mais surtout pas
  `typedef enum {VRAI, FAUX} Booleen;`
  – les valeurs attribuées par le compilateur aux constantes énumérées sont, par défaut, des entiers successifs à partir de 0, ce qui met bien ici **`FAUX`** en correspondance avec 0 (faux en C) et **`VRAI`** avec 1 (non nul, donc vrai en C). Inverser les deux constantes dans l'énumération **`{VRAI, FAUX}`** serait catastrophique puisque l'interprétation C serait inversée (sauf si on spécifie explicitement les valeurs associées ; i.e. **`{VRAI=1, FAUX=0}`**).
  – les variables de ce type sont stockées en mémoire comme le type **`int`** : sur 32 bits sous *Linux*
  – si vous avez quelques variables booléennes à définir, ça n'est pas un problème.

1.11 Types structure -

15

■ **Définition**

♦ `#define LONGUEUR ( 80 + 1 )`

```
typedef struct produit { int      reference;
                         char nom[LONGUEUR];
                         double        prix; } Produit;
```

■ **Le nommage de la structure est souvent inutile**

```
typedef struct { int      reference;
                 char nom[LONGUEUR];
                 double        prix; } Produit;
```

■ **Accès aux champs**

♦ `Produit produit;`

♦ `produit.prix = 15.8;`

– **Définition et initialisation**

```
#define NB_POINTS 3

typedef struct { double x;
                 double y; } Point;

// définition de deux points, initialisation du deuxième
Point point1, point2 = {1.,-2.};

// définition et initialisation d'un tableau de points
Point tabPoints[NB_POINTS] = { {1.5,2.}, {1.,-2.}, {0.,3.} };
```

– **La seule opération globale sur l'objet de type structure est l'affectation**

```
point1 = tabPoints[1];
```

– **Les comparaisons se font champ à champ**

```
if ( point1.x == point2.x && point1.y == point2.y )
  printf ( "les points sont confondus\n" );
```

1.12 Fonctions -

Une fonction du langage algorithmique

```
int minimum ( int x, int y ) {
    if ( x < y ) return x;
    else return y;
}
```

16

Une procédure du langage algorithmique

```
void afficherNombresParfaits ( int n ) {
    int i;
    for ( i=1 ; i<=n ; i++ )
        if ( estParfait ( i ) == VRAI )
           printf ( "%d est parfait\n", i );
}
```

- **Il n'y a que des fonctions en C**
- **Syntaxe**
  - type de la valeur de retour
  - identificateur
  - liste parenthésée de paramètres typés
  - bloc de définition entre accolades

- **Fonction du langage algorithmique**
  - fonction C qui renvoie une valeur à l'appelant
  - mot réservé **`return`** qui correspond à l'action **retourner** de notre langage algorithmique
  - le type de retour peut être un type scalaire : **`int`**, **`double`**, . . .
  - Cela peut être également un type structure
  - Dans tous les cas, il doit être identique au type de retour annoncé.

- **Procédure du langage algorithmique**
  - fonction C typée **`void`** (vide)
  - la fonction C ne renvoie rien à l'appelant
  - nous verrons plus loin comment gérer en C les paramètres en résultat ou en donnée-résultat du langage algorithmique

1.13 Affichage et saisie -

17

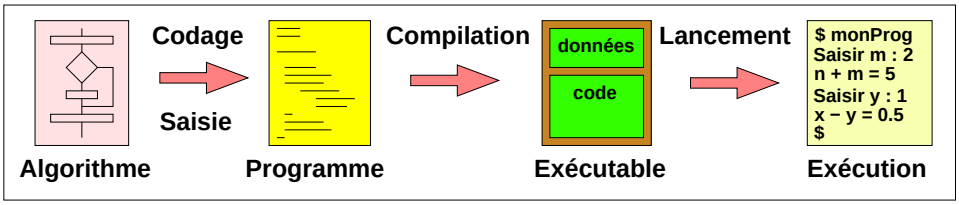
```
#include <stdio.h>

int n = 3, m;
double x = 1.5, y;

printf ("Saisir m : ");
scanf ("%d", &m );
printf ("n + m = %d\n",n+m);

printf ("Saisir y : ");
scanf ("%lf", &y );
printf ("x - y = %f\n",x-y);
```

- **Les fonctions** *printf* **et** *scanf*
  - **Premier argument : chaîne** *format*
    - Spécifie le type de conversion
      %d : int, **%ld : long**
      %f : float et **double** (*printf*)
      %f : float (*scanf*)
      %lf : double (*scanf*)
      %s : tableau de caractères
    - Caractères spéciaux
      \n : retour à la ligne
      \t : tabulation
  - **Arguments en nombre variable**
    - *printf* : expressions
    - *scanf* : adresses de variables

– **Affichage :** *printf*
  – le premier argument obligatoire est la chaîne *format*.
  – elle peut contenir des spécifications de conversion de format qui sont introduites par le caractère '**%**'.
    Ces spécifications indiquent à la fonction *printf* comment générer la séquence de caractères à afficher dans la fenêtre *shell* ; ainsi dans le deuxième appel de *printf*, la spécification **%d** correspond au type **int** de l'expression **n+m**
  – le nombre d'arguments autres que la chaîne *format* est variable et doit correspondre au nombre de spécifications de conversion de la chaîne *format*.

– **Saisie :** *scanf*
  – considérez que la chaîne *format* ne doit contenir que des spécifications de conversion de format
  – on vous expliquera juste après pourquoi il faut passer comme argument les expressions **&m** ou **&y** qui correspondent aux adresses des variables **m** et **y**

1.14 Chaîne de production -



18

- **Codage/Saisie**
  - `$ xemacs monProg.c &`
- **Compilation**
  - `$ gcc -Wall monProg.c -o monProg`
- **Lancement de l'exécution**
  - `$ monProg`
- **Mais aussi** :
  - Interpréter les messages d'erreurs du compilateur
  - Corriger le fichier source
  - Recompiler
  - . . .

– **Saisie**
  – utiliser un éditeur syntaxique comme *emacs* ou *xemacs* ; il vous apportera une aide précieuse pour l'indentation du source, la gestion des blocs, des parenthésages, . . .
  – lancez-le en arrière-plan car il est vraisemblable que votre fichier source ne « passera » pas du premier coup à la compilation ; il vous faudra alors corriger le source, recompiler, . . .
  – attention aux changements de répertoire dans la fenêtre *shell* après le lancement de l'éditeur.

– **Compilation**
  – la commande *gcc* prend bien sûr votre fichier source comme argument ainsi qu'un certain nombre d'options.
  – elle vous affichera une liste de messages d'erreur ; il y a deux catégories d'erreurs :
    – les erreurs « fatales » bloquent le processus de compilation ; il faudra bien les corriger
    – les erreurs de type *warning* ne bloquent pas le processus de compilation ; certaines d'entres-elles sont bénignes, d'autres plus suspectes qui risquent de provoquer des erreurs à l'exécution.
  – l'option **`-Wall`** (*warnings all*) permet d'afficher toutes les erreurs ; votre objectif est que la compilation ne provoque aucun message d'erreur.
  – l'option **`-o nomFichier`** permet de produire un exécutable avec le nom spécifié ; par défaut, le compilateur générera un fichier **`a.out`**

– **Lancement de l'exécution**
  – suivant le contenu de votre variable d'environnement *PATH*, il vous faudra taper **`nomProg`** ou **`./nomProg`**

2 Le passage de paramètres

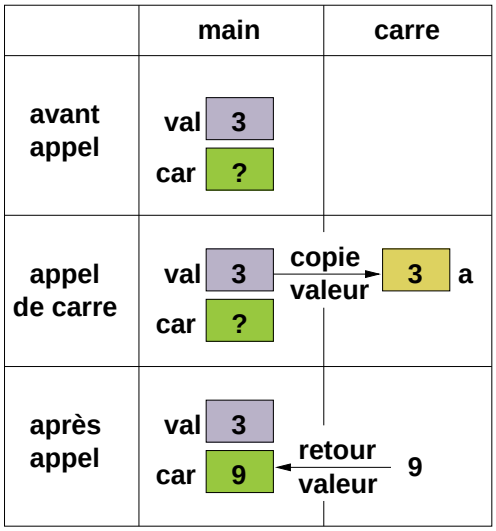
19

2.1 Passage de paramètres "donnée"

20

carre.c

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

int carre ( int a ) {
  return a * a;
}

int main ( ) {
  int val = 3, car;
  car = carre (val);
  printf ("carré=%d\n",car);
  return EXIT_SUCCES;
}
```



– **Pas de difficulté**
  – vous verrez, de manière plus technique, comment s'opère le retour d'une valeur vers l'appelant dans le module AM12.

– **La valeur de retour est ici de type `int`**
  – on aurait pu utiliser directement cette valeur à l'appel de *printf* sans passer par la variable **`car`** :
  `printf ( "carré = %d\n", carre ( val ) );`

– **Les arguments d'une fonction sont plus généralement des expressions**
  `car = carre ( val + 1 );`

2.2 Passage de paramètres "résultat"

21

- **Recette**
  - **Prototype** : si le prototype d'une procédure dans le langage algorithmique fait apparaître un paramètre formel « résultat » (ou « donnée-résultat »), alors l'identificateur de ce paramètre doit en C être précédé du caractère *****, aussi bien dans le prototype que dans le corps de la fonction C correspondante.
  - **Appel** : au niveau de l'**appel**, l'objet utilisé comme paramètre effectif doit être précédé du caractère **&**, sauf si ce paramètre formel est de type *tableau*.
- **Exemple** :

procédure verbe (**donnée** x : TypeX, **donnée-résultat** y :TypeY, **résultat** z : TypeZ)
    // action sur x, y et z
fproc verbe

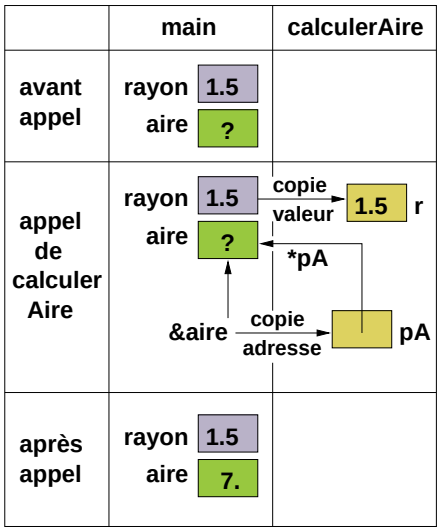
```
void verbe ( TypeX x, TypeY *y, TypeZ *z ) {
  // action sur x, *y et *z
}
```

22

2.3 Passage de paramètres "résultat"

calculerAire.c

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

void calculerAire(double r,
                  double *pA) {
  *pA = M_PI * r * r;
}
int main ( ) {
  double rayon = 1.5, aire;
  calculerAire (rayon, &aire);
  printf ("Aire = %f\n", aire);
  return EXIT_SUCCESS;
}
```



– **Le transparent présente une version allégée de la fonction pour un problème de place**
  `void calculerPerimetreAire (double r, double *pP, double *pA);`
  – pour calculer l'aire, il aurait suffit de la fonction :
    `double aire ( double r );`
  – voici la définition complète de la fonction et du *main* :

```
void calculerPerimetreAire (double r, double *pP, double *pA) {

  *pP = 2 * M_PI * r;
  *pA = M_PI * r * r;
}

int main ( ) {

  double rayon = 1.5, perimetre, aire;

  calculerPerimetreAire (rayon, &perimetre, &aire);
  printf ("Périmètre = %f, aire = %f \n", perimetre, aire);
  return EXIT_SUCCESS;
}
```

  – ce qui est illusté sur le transparent pour le paramètre formel **pA** est bien sûr valable pour le paramètre **pP**.
  – le « retour » de valeur vers l'appelant se fait par affectation directe d'expression aux objets dont l'adresse a été transmise lors de l'appel.

23

2.4 Adresse mémoire

- **Définition de variable : `int nb ;`**
  - **D'un point de vue conceptuel**
    - *nb* est un symbole désignant une variable qui prend des valeurs entières
    - *nb* | 5 |
  - **Vue du compilateur**
    - *nb* est une zone mémoire pour stocker des valeurs entières
    - Le compilateur attribue une **adresse** à *nb* en mémoire
    - **adresse** 104 | 5 |
- **L'adresse d'un objet est une donnée en elle-même**
- **Le langage C autorise** :
  - L'utilisation de l'adresse d'un objet
  - La manipulation d'adresses mémoire
  - La définition de variables qui contiennent des adresses mémoire
- **L'opérateur &**
  - l'expression **&nb** dénote l'adresse de *nb*
  - **&nb** vaut 104 → | 5 |

– **Les aspects plus « techniques » du langage C apparaissent rapidement**

– il faudra vous habituer à manipuler les adresses des variables ; sur ce sujet, des compléments seront apportés lors du C10

– dans l'immédiat,on peut se contenter d'utiliser la « recette » du transparent 2.2 pour transcrire l'appel d'une fonction C implémentant une procédure avec des paramètres en résultat ou en donnée-résultat de notre langage algorithmique

– le compilateur dispose d'un espace de stockage dans lequel il choisit la position qu'il attribue à la zone correspondant à chaque variable, repérée par une adresse. On peut se représenter cet espace comme un tableau d'octets et l'adresse comme l'indice dans ce tableau du premier octet de la zone associée

2.5 Du langage algorithmique au langage C

24

- **Paramètres en donnée**
  - **Passage par valeur**
- **Paramètres en résultat ou en donnée-résultat**
  - **Nécessité de disposer d'une zone de stockage pour chaque résultat**
  - **Possibilité pour l'appelé de modifier le contenu d'objets de l'appelant**
    Or les objets de l'appelant ne sont pas visibles par l'appelé
- **Copie des valeurs des adresses de ces objets pour l'appelé qui y accède en utilisant ces adresses**
  - **Paramètres « adresse de Type »**
  - `void calculerPerimetreAire (double r, double *pP, double *pA);`
- **Accès pour l'appelé aux objets désignés**
  - **Utilisation de l'opérateur de déréférencement : `*`**
  - `*pA = M_PI * r * r;`

– **Référencement/déréférencement**
  – l'opérateur de référencement ou « adresse » : **&**
  – l'opérateur de déréférencement d'une adresse ou d'indirection : *****

2.6 Synthèse sur le passage de paramètres

25

- **Passage de paramètres « donnée »**
  - Le **paramètre effectif** correspond à la valeur d'une **expression**
  - La **valeur** du paramètre effectif est recopiée dans le paramètre formel
  - Si le paramètre effectif est un **identificateur de variable**, celle-ci **n'est jamais mofifiée** par la fonction appelée
- **Passage de paramètres « résultat »**
  - Le **paramètre effectif** est l'**adresse** d'un objet
  - Cette **adresse** est recopiée dans le paramètre formel
  - L'objet de l'appelant dont l'adresse est contenue dans le paramètre effectif **est modifiable** par la fonction appelée : par exemple la fonction *scanf*

– **La fonction C** *scanf* :
  – correspond à une procédure avec des paramètres en résultat du langage algorithmique
  – les paramètres effectifs, hormis la chaîne format, sont des « expressions-adresses »
  – exemple :

```
#define LONGUEUR 20

char nom[LONGUEUR];
int age;

printf ( "Saisissez un nom et un âge : " );
scanf ( "%s%d", nom, &age ); // nom est une << expression-adresse >>
// ...
```

3 Transcription algorithme vers programme C

26

3.1 La procédure afficherNombresParfaits -

27

- **La procédure en langage algorithmique**

```
procédure afficherNombresParfaits (donnée nb : Naturel)
    pour i = 1 : Naturel à nb
                si estParfait ( i ) = VRAI alors
                    afficher ( i )
                    allerAlaLigne ( )
                fsi
    fpour i
fproc afficherNombresParfaits
```

- **La fonction en langage C**

```
void afficherNombresParfaits(int nb) {
  int i;
  for(i=1;i<=nb;i++)
    if(estParfait(i)==VRAI) printf("%d est parfait\n",i);
}
```

3.2 La fonction estParfait -

28

■ **La fonction en langage algorithmique**

```
fonction estParfait (nombre : Naturel) : Booléen

    somme = 0 : Naturel // somme des diviseurs

    pour diviseur = 1 : Naturel à nombre div 2
            si nombre mod diviseur = 0 alors
                somme ← somme + diviseur
            fsi
    fpour diviseur
    si somme = nombre alors
            retourner VRAI
    sinon
            retourner FAUX
    fsi
ffct estParfait
```

■ **La fonction en langage C**

```
Booleen estParfait(int nombre) {

  int somme=0, diviseur;

  for(diviseur=1;
        diviseur<=nombre/2;
          diviseur++)
    if (nombre%diviseur==0)
       somme+=diviseur;

  if(somme==nombre)
    return VRAI;
  return FAUX;
}
```

3.3 La fonction principale

29

■ **La fonction en langage algorithmique**

```
fonction principale ( ) : CodeRetour

    n : Naturel

    afficher ( "Saisir un entier > 1 : " )
    saisir ( n )
    si n ≤ 1 alors
            afficher ( "Valeur incorrecte" )
            allerAlaLigne ( )
            retourner KO
    fsi
    afficherNombresParfaits ( n )
    retourner OK

ffct principale
```

■ **La fonction en langage C**

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

typedef enum{FAUX,VRAI} Booleen;

int main() {

  int n;
  printf("Saisir un entier > 1 : ");
  scanf("%d",&n);
  if( n <= 1 ) {
    printf("Valeur incorrecte\n");
    return EXIT_FAILURE;
  }
  afficherNombresParfaits(n);
  return EXIT_SUCCESS;
}
```

Le fichier nombresParfaits.c

```
#include <stdlib.h> // pour la constante symbolique  EXIT_SUCCESS
#include <stdio.h>  // pour les prototypes de printf et scanf

// définition du type Booleen
typedef enum{FAUX,VRAI} Booleen;

// déclaration des fonctions afficherNombresParfaits et estParfait

void afficherNombresParfaits(int nb);
Booleen estParfait(int nombre);

/* le programme principal qui demande la saisie d'un
nombre et qui appelle la fonction afficherNombresParfaits */

int main() {

  int n;

  printf("Saisir un entier > 1 : ");
  scanf("%d",&n);
  if( n <= 1 ) { printf("Valeur incorrecte\n");
                 exit(EXIT_FAILURE); }
  afficherNombresParfaits(n);
  return EXIT_SUCCESS;
}

/* La fonction C (procédure en algorithmique) qui affiche
tous les nombres parfaits <= nb */

void afficherNombresParfaits(int nb) {

  int i;

  for(i=1;i<=nb;i++)
    if( estParfait(i) ) printf("%d est parfait\n",i);
}

/* La fonction C (fonction en algorithmique) qui
détermine si le paramètre nombre est parfait ou non */

Booleen estParfait(int nombre) {

  int somme=0, diviseur;

  for(diviseur=1;diviseur<=nombre/2;diviseur++)
    if (nombre%diviseur==0) somme+=diviseur;
  if(somme==nombre) return VRAI; // ou plus simplement
  return FAUX;                   // return (somme==nombre);
}
```

TP1-TP2

LES EXERCICES À CODER

Algorithmes à coder

1. **Ce que vous devez coder avant la séance de TP**
   (a) Les nombres parfaits [1.11] (recopie)
   (b) L'année bissextile [1.2]
   (c) L'échange du contenu de deux variables [transparent 2.14 du C2]
   (d) L'inversion d'un tableau en itératif [2.3]
   (e) Le calcul de n ! en itératif [1.5]

2. **Ce que vous devez coder durant la séance de TP**
   (a) Les nombres premiers en itératif [1.6]
   (b) Le plus petit et le plus grand éléments d'un tableau en itératif [2.2]
   (c) La recherche d'un élément dans un tableau de structures [2.10]
   (d) Le nombre d'occurrences d'un élément [2.4]
   (e) L'élément le plus fréquent [2.5]

3. **Pour aller plus loin**
   (a) Un petit calcul de partage [1.1]
   (b) Le temps plus une seconde [1.3]
   (c) Le nombre de jours de congés [1.4]
   (d) Affichage du développement de l'expression $(x + y)^n$ [1.13]
   (e) L'histogramme [2.12]

TP3-TP4

LES EXERCICES À CODER

AOÛT 2006

Algorithmes à coder

1. **Ce que vous devez coder avant la séance de TP**
   (a) La recherche du zéro d'une fonction par dichotomie en itératif [1.8]
   (b) L'intégration par la méthode des trapèzes en itératif [1.10]
   (c) Les sous-séquences croissantes [2.7]
   (d) La transposée d'une matrice carrée [2.8]
   (e) n ! en récursif [transparent 2.2 du C3]

2. **Ce que vous devez coder durant la séance de TP**
   (a) Le calcul de C(n,p) en récursif [3.1]
   (b) Le terme de la suite de Fibonacci en récursif [3.2]
   (c) L'inversion d'un tableau en récursif [3.5]
   (d) Le plus petit et plus grand éléments d'un tableau en récursif [3.7]
   (e) Le tri à bulles [4.3]
   (f) Calcul du PGCD : Algorithme d'Euclide en récursif [3.3]

3. **Pour aller plus loin**
   (a) Le déterminant d'une matrice carrée d'entiers en récursif [3.9]
   (b) La suite de Fibonacci en itératif [1.9]
   (c) Les racines d'une équation du second degré [1.12]
   (d) La recherche dichotomique en itératif [2.6]
   (e) Le produit de deux matrices [2.9]
   (f) La somme des éléments d'une matrice d'entiers en récursif [3.8]

C7

L'ABSTRACTION DE DONNÉES, LA MODULARITÉ EN C, LA CHAÎNE DE PRODUCTION

2

Objectifs du C7

- Aborder l'abstraction de données
- Savoir créer et utiliser un module en C
- Faire la distinction entre compilation et édition des liens
- Savoir créer un fichier *makefile* et utiliser la commande *make*

3

Contenu du C7

1 L’abstraction de données

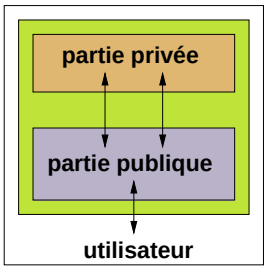
4

5

1.1 Définir des données de manière abstraite

■ **L'encapsulation des données**

- ♦ fournir à l'utilisateur de nouvelles structures de données et des opérations pour les manipuler
- ♦ **partie publique** : les services offerts
- ♦ **partie privée** : réalisation des services



■ **Type abstrait de données**

- ♦ **cacher** les détails de l'implémentation
- ♦ **convention** entre le réalisateur et l'utilisateur
  - ► Type «Nom»
  - ► Utilise «types»
  - ► Opérations
  - ► Sémantique

■ **Correspond à un module en C**

- ♦ partie publique : **interface** (fichier **`.h`**)
- ♦ partie privée : **définition** (fichier **`.c`**)
- ♦ dépendances : **directive `#include`**

– **L'abstraction de données**
  – c'est ce qu'on peut faire de mieux avec un langage **impératif procédural** comme le langage C.
  – vous aborderez en deuxième année un langage **impératif orienté objet**, le langage *java* qui offre de toutes autres possibilités.

– **Type abstrait et module en C**
  – le module C implémente le type abstrait ; l'utilisateur n'a pas besoin d'avoir accès directement aux structures de données, il n'utilise que les fonctions qui sont *déclarées* dans l'interface.
  – la réutilisation est très aisée ; il suffit de **dupliquer** les deux fichiers du module dans le répertoire où vous développez votre application.
  – fini les opérations de « **copier-coller** » entre fichiers sources avec toutes les possibilités d'erreurs qui en découlent.

6

1.2 Exemple : nombres complexes -

- **Type Complexe**
- **Utilise Réel**
- **Opérations**
  - nouveauComplexe : Réel $\times$ Réel $\rightarrow$ Complexe
  - partieRéelle : Complexe $\rightarrow$ Réel
  - partieImaginaire : Complexe $\rightarrow$ Réel
  - ... déclarer toutes les opérations de base sur les complexes
- **Sémantique** : décrire l'effet de ces opérations

- **Implémentation** : décrire comment sont réalisées ces opérations

```
fonction plus ( z1, z2 : Complexe ) : Complexe
        r,i : Réel
        r ← partieRéelle ( z1 ) + partieRéelle ( z2 )
        i ← partieImaginaire ( z1 ) + partieImaginaire ( z2 )
        retourner nouveauComplexe ( r, i )
ffct plus
```

– **Organisation**
  – les définitions de types se trouvent normalement dans le fichier d'interface, elles peuvent être partiellement **cachées** (utilisation de pointeurs).
  – les déclarations des fonctions C implémentant les opérations du type abstrait sont précédées d'un commentaire apportant une **sémantique** (un sens) à ces opérations.
  – les définitions des fonctions C peuvent bien sûr accéder aux structures de données, elles peuvent néanmoins appeler d'autres fonctions du module, en particulier des fonctions non visibles au niveau de l'interface.
  – la définition du type *Complexe* ne figure pas sur le transparent ; la définition de la fonction *plus* utilise les fonctions déjà définies. Redéfinissons-la en accédant directement à la structure de données :

```
type Complexe = structure
    re : Réel
    im : Réel
fstruct

fonction plus ( z1, z2 : Complexe ) : Complexe

    z : Complexe

    z.re ← z1.re + z2.re
    z.im ← z1.im + z2.im
    retourner z

ffct plus
```

2 La modularité en C

7

2.1 Un programme mono-module

8

inversion.c

```
#include <stdlib.h>
#include <stdio.h>
#define N 10

// déclaration des fonctions
void saisirTableau(int t[]);
void inverserTableau(int t[]);
void afficherTableau(int t[]);

int main() {
  int tab[N];
  saisirTableau ( tab );
  inverserTableau ( tab );
  afficherTableau ( tab );
  return EXIT_SUCCESS;
}
// définition des fonctions
void saisirTableau(int t[]) { ... }
void inverserTableau(int t[]) { ... }
void afficherTableau(int t[]){ ... }
```

- **Programme mono-module**
  - un seul fichier source
  - un ensemble de fonctions dont la fonction *main*
- **Limites**
  - **difficile à gérer** pour une application conséquente
  - Une **seule unité** de décomposition
    - la **fonction**, abstraction d'une **action**
    - pas d'unité pour abstraire un **type** de données et ses opérations associées
  - pas de **réutilisation** possible entre programmes
  - **mise au point** délicate

– **Limites du développement mono-fichier**
  – l'exemple du transparent n'est pas à la hauteur de ce qu'on voudrait vous faire sentir.
  – imaginez un fichier source de plusieurs centaines ou milliers de lignes.

2.2 Module

9

- **Définition**
  unité de programme composée de constantes, de types, de variables et de fonctions
- **Interface d'un module** : fichier **.h**
  partie **publique** accessible par les unités utilisatrices
  - ♦ définition de constantes et de types
  - ♦ prototypes des fonctions
- **Corps d'un module** : fichier **.c**
  partie **privée** non accessible par les unités utilisatrices
  - ♦ définition et initialisation éventuelle des variables
  - ♦ définition des fonctions
- **Principe d'encapsulation**
  les objets définis dans le module ne sont **manipulables** qu'à travers les fonctions **proposées** par l'interface
  - ♦ utilisation en conformité avec cette interface

2.3 Un programme utilisant un module

10

tabio.h

```
// déclaration des fonctions

void saisirTableau(int t[], int n);
void afficherTableau(int t[], int n);
```

tabio.c

```
#include <stdio.h>
#include "tabio.h"

// définition des fonctions

void saisirTableau(int t[], int n) {
  // ...
}
void afficherTableau(int t[], int n ){
  // ...
}
```

module tabio

inversion.c

```
#include <stdlib.h>
#include "tabio.h"
#define N 10

// déclaration de la fonction
void inverserTableau(int t[]);

int main() {
  int tab[N];
  saisirTableau( tab, N);
  inverserTableau( tab );
  afficherTableau( tab, N);
  return EXIT_SUCCESS;
}
// définition de la fonction
void inverserTableau(int t[]) {
  // ...
}
```

programme principal

- **Module tabio** (tableau *input/output*)
  - le constante symbolique *N* n'est pas connue du module *tabio*.
  - le paramètre *n* (nombre d'éléments du tableau) a été rajouté aux en-têtes des fonctions *saisirTableau* et *afficherTableau*.
  - le module peut être utilisé quelque soit la longueur du tableau dans le programme principal.
  - il est néanmoins limité à la saisie et à l'affichage d'éléments de type **`int`**.

- **La directive `#include "tabio.h"`**
  - dans le fichier *tabio.c*, quoique dans cet exemple ça n'est pas obligatoire, le fichier *tabio.h* ne contenant que des déclarations de fonction (en revanche, dès qu'il y a définition de constantes et de types, cela devient indispensable et c'est généralement le cas).
    Cela permet au compilateur de contrôler la cohérence entre les prototypes annoncés dans l'interface et ceux utilisés pour la réalisation. Par conséquent, incluez systématiquement le fichier **`.h`** dans le fichier **`.c`** correspondant.
  - dans le fichier *inversion.c* pour que le compilateur puisse vérifier que l'appel des fonctions du module *tabio* est conforme à leur définition.

2.4 Intérêt des modules

11

- **Réutilisation**
  un module peut être réutilisé par un autre programme ou un autre module
- **Extensibilité-Flexibilité**
  - modification du corps d'un module sans toucher à l'interface
  - ajout de nouveaux services à un module en complétant l'interface
  - création d'un module supplémentaire, remplacement d'un module
- **Sécurité**
  une unité utilisatrice n'a pas accès au corps du module
  - elle ne peut qu'appeler les fonctions spécifiées par l'interface du module
  - elle ne peut donc pas modifier directement les objets définis dans le module
- **Validation**
  chaque module est testé séparément
  - facilite la vérification du bon fonctionnement de l'application
- **Utilisation**
  - les bibliothèques : standard, mathématique, graphiques (Xlib, GTK), . . .
  - création de nouveaux types de données : rationnels, complexes, . . .

– **Réutilisation**
  – quand vous implémenterez les algorithmes de tri, il faudra les valider. Pour chacun d'eux, vous devrez saisir les éléments d'un tableau et l'afficher trié.
  Avec le module *tabio*, vous n'aurez plus besoin de définir les fonctions *saisirTableau* et *afficherTableau* dans chacun des programmes de tri.

– **Validation**
  – c'est un point essentiel : un module doit être validé avant d'être proposé aux utilisateurs.
  Pour cela, il faut écrire une fonction *main* qui permettra de vérifier le module avec un *jeu de tests* représentatif.

– **Utilisation**
  – dans le parcours AP11, vous aurez, à plusieurs reprises, l'occasion de définir et/ou d'utiliser des modules :
    – le petit module *tabio* pour les méthodes de tri
    – dans l'application image, on vous fournira trois modules prêts à l'emploi, vous aurez à compléter le quatrième
    – dans un des TP AP11, on vous fournira un module gérant le type *Personne* afin de pouvoir développer rapidement une application gérant un répertoire de personnes à l'aide d'une liste chainée
  – enfin dans le projet informatique PI12, vous aurez à découper votre application en différents modules

2.5 Module et compilation

12

- **Unité de compilation**
  - un fichier **`.h`** n'est pas une unité de compilation
  - un fichier **`.c`** est une unité de compilation qui inclut des fichiers **`.h`**
- **Commandes**
  - compilation :
    - **`$ gcc -Wall -c tabio.c`** → production de **`tabio.o`**
    - **`$ gcc -Wall -c inversion.c`** → production de **`inversion.o`**
  - édition des liens : **`$ gcc -Wall inversion.o tabio.o -o inversion`**
- **Règles de compilation**
  - compiler le fichier **`.c`** d'un module **avant** toute unité utilisatrice
  - modification du **corps** d'un module
    - recompiler le fichier **`.c`** du module
    - édition des liens **sans** recompiler les unités utilisatrices
  - modification de l'**interface** d'un module
    - recompiler le fichier **`.c`** du module
    - recompiler **toutes** les unités utilisatrices
  - recompiler n'**importe quelle** unité nécéssite de **refaire** l'édition des liens

– **fichier `.h`**
  – il ne doit pas contenir de **définition** de fonctions.

– **Compilation séparée**
  – chaque module ainsi que le programme principal doit être **compilé** indépendamment du reste.
  – à la compilation du fichier *inversion.c*, il y aura des *références non résolues* ; l'appel des fonctions *saisirTableau* et *afficherTableau* qui sont définies ailleurs.
  – ces *références* seront résolues à la phase d'**édition des liens** car les définitions des fonctions *saisirTableau* et *afficherTableau* sont contenues dans le fichier *tabio.o*.
  – à ce stade, considérez que les fichiers objet (**`.o`**) sont des «morceaux» d'exécutable et que ces morceaux seront rassemblés lors de l'édition des liens

2.6 Le module complexe -

13

complexe.h

```
// définition du type Complexe

typdef struct { double re;
                double im; } Complexe;


/* déclaration des fonctions manipulant
   le type Complexe */


Complexe nouveauComplexe ( double r,
                           double i );

Complexe plus (Complexe z1,Complexe z2);

Complexe moins(Complexe z1,Complexe z2);

double module ( Complexe z );

double argument ( Complexe z );
```

complexe.c

```
#include <math.h> // pour sqrt et atan
#include "complexe.h"
Complexe nouveauComplexe ( double r,
                           double i ) {
  Complexe z;
  z.re = r, z.im = i;
  return z;
}
Complexe plus (Complexe z1,Complexe z2){
  return nouveauComplexe (z1.re+z2.re,
                          z1.im+z2.im);
}
Complexe moins(Complexe z1,Complexe z2){
  return nouveauComplexe (z1.re-z2.re,
                          z1.im-z2.im);
}
double module ( Complexe z ) {
  return sqrt (z.re*z.re + z.im*z.im);
}
double argument ( Complexe z ) {
  return atan ( z.im / z.re );
```

- *complexe.h*
  - définition du type *Complexe*
  - début de la liste des déclarations des fonctions implémentant les opérations sur le type *Complexe*.

- *complexe.c*
  - dépendance : **`#include "complexe.h"`**.
  - début de la liste des définitions des fonctions implémentant les opérations sur le type *Complexe*.
  - le module utilise deux fonctions mathématiques *sqrt* (racine carrée) et *atan* (arc tangente).
    ces fonctions sont déclarées dans le fichier d'en-tête *math.h* et définies dans la librairie mathématique.
    → option supplémentaire lors de l'édition des liens (voir transparent 3.2).

2.7 Un programme utilisant le module complexe -

useComplexe.c

```
#include <stdlib.h>
#include <stdio.h>
#include "complexe.h"

int main ( ) {

  Complexe z1, z2, z3;

  z1 = nouveauComplexe ( 1., 0.); // appel du constructeur
  z2 = nouveauComplexe ( 0., 1.);
  z3 = plus ( z1, z2);
  printf ( "Module de z3   : %f\n", module ( z3 ) );
  printf ( "Argument de z3 : %f\n", argument ( z3 ) );
  return EXIT_SUCCESS;
}
```

14

Un exemple minimal d'utilisation

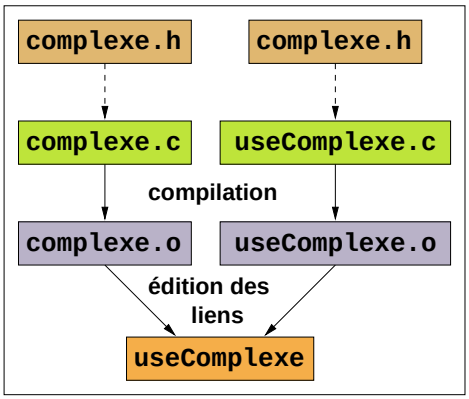
3 La chaîne de production/le fichier makefile

15

16

3.1 Présentation simplifiée de la chaîne de production

- **Compilation**
  - **Analyse lexicale et syntaxique**
  - **Production d'un fichier *objet* (`.o`)**
  - **Options**
    - `-c` → compilation seule
- **Edition des liens**
  - **Regroupement de fichiers objet**
  - **Production d'un fichier exécutable**
  - **Options**
    - `-o` → nom de l'exécutable
    - `-l` → ajout de bibliothèque



- **Etude détaillée → module AM12**

- `$ gcc -Wall -c complexe.c`
- `$ gcc -Wall -c useComplexe.c`
- `$ gcc -Wall -lm useComplexe.o complexe.o -o useComplexe`

– **Options de compilation**
  – `-c` : l'enchaîneur de passes *gcc* s'arrête à la production d'un fichier objet ayant le même nom que le source mais suffixé par `.o`.

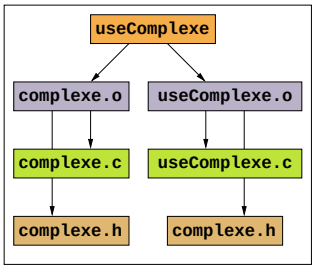
– **Edition des liens**
  – comme nous l'avons déjà vu, à la compilation du fichier *useComplexe.c* les appels des fonctions du module *complexe* sont des références non résolues ; à l'édition des liens elles le seront.
  – option `-l` : ici, il est nécessaire d'utiliser l'option `-lm` (comme librairie mathématique) sinon les appels des fonctions *sqrt* et *atan* resteraient des références non résolues. On en reparlera lors du prochain cours.
  – en ce qui concerne la bibliothèque standard (*libC*), elle est prise en compte d'office par *gcc*.

17

3.2 Fichier makefile

■ **Mémoriser dans un fichier** *makefile*

♦ **l'arbre des dépendances**



♦ **les actions de reconstructions**

makefile

```
cible principale : dépendances
<TAB>   action
cible : dépendances
<TAB>   action
```

makefile

```
useComplexe : useComplexe.o\
              complexe.o
<TAB>   gcc -Wall -lm useComplexe.o\
        complexe.o -o useComplexe

useComplexe.o : useComplexe.c\
                complexe.h
<TAB>   gcc -Wall -c useComplexe.c

complexe.o : complexe.c complexe.h
<TAB>   gcc -Wall -c complexe.c

# commentaires
```

■ **Commande** *make*

♦ **`$ make`** → cible principale

♦ **`$ make complexe.o`**

♦ parcours récursif de l'arbre de dépendances

– **Arbre des dépendances** issu :
  – de l'architecture de l'application pour les fichiers **`.o`** et **`.c`**.
  – des directives **`#include`** pour les fichiers d'en-tête.

– **Le fichier** *makefile* **ou** *Makefile*
  – constitué d'une série de déclarations sur 2 lignes de type :
    – cible : liste de dépendances
    – **<TAB>**action

    <TAB> veut dire tapez sur la touche tabulation en début de ligne et non saisir "TAB" ni une suite de caractères espace qui ne serait en aucun cas équivalent.
  – la cible principale est la première apparaissant dans le fichier.
  – les cibles et les dépendances sont généralement des noms de fichier, mais il peut y avoir des pseudo-cibles.
  – le caractère '\' permet de poursuivre sur la ligne suivante.
  – possibilité d'introduire des commentaires ; ils débutent par le caractère '#' et courent jusqu'à la fin de la ligne.

– **La commande** *make*
  – lit le fichier *makefile* présent dans le répertoire et tente de produire la cible principale (appel : **`make`**) ou la cible spécifiée (appel : **`make nomDeCible`**) en exécutant l'action correspondante.
  – elle s'arrête à la première action renvoyant un statut d'erreur.
  – elle n'exécute que les actions strictement nécessaires ; pour cela, elle se base sur la date et l'heure de dernière modification des fichiers.

18

3.3 Les variables du fichier makefile

■ Définition et utilisation dans le *makefile*

makefile

```
OBJETS = inversion.o tabio.o

CC = gcc

CFLAGS = -Wall -g

inversion : $(OBJETS)
      $(CC) $(CFLAGS) $(OBJETS)\
      -o inversion

inversion.o : inversion.c tabio.h
      $(CC) $(CFLAGS) -c inversion.c

tabio.o : tabio.c tabio.h
      $(CC) $(CFLAGS) -c tabio.c
```

$ make

■ Définition à l'appel de la commande *make*

makefile

```
all : $(APPLI) clean

$(APPLI) : $(APPLI).o tabio.o
      gcc -Wall $(APPLI).o\
      tabio.o -o $(APPLI)

$(APPLI).o : $(APPLI).c tabio.h
      gcc -Wall -c $(APPLI).c

tabio.o : tabio.c tabio.h
      gcc -Wall -c tabio.c

# suppression des fichiers objet
clean : $(APPLI)
      rm -f *.o
```

$ make APPLI=inversion

- **Définition de variables** dans le fichier *makefile* ou dans l'appel de *make*
  - nomVariable = valeur
- **Utilisation**
  - $(nomVariable) ou ${nomVariable}
- **Intérêt**
  - spécifier une fois pour toutes la liste des fichiers objet, les options de compilation . . .
- **Intérêt de la définition d'une variable à l'appel de la commande** *make*
  - dans certains cas, cela permet de rendre le fichier *makefile* plus générique.
  - lorsque vous implémentez les algorithmes de tri, vous avez dans votre répertoire :
    - le module *tabio*
    - le fichier *makefile* générique (celui de droite sur le transparent)
    - vos programmes de tri : makefile
      - *triSelection.c*
      - *triInsertion.c*
      - *triAbulles.c*
      - *triRapide.c*
      - *triFusion.c*
  - pour compiler un de ces programmes, il vous suffit de lancer la commande *make* en spécifiant le source à prendre en compte :
    $ make APPLI=triSelection
  - vous pouvez utiliser à chaque fois le même *makefile* sans avoir à l'éditer pour passer d'un programme de tri à un autre.

TP5

LES EXERCICES À CODER

Exercices à réaliser

1. **Ce que vous devez faire avant la séance de TP**
   (a) **Application modulaire : le tri à bulles**

      i. Vous disposez du programme mono-fichier *triAbulles.c*

triAbulles.c (début)

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

#define LONGUEUR 10

typedef int Tableau[LONGUEUR];
typedef enum { FAUX, VRAI } Booleen;

void triAbulles ( Tableau t );
void saisirTableau ( Tableau t );
void afficherTableau ( Tableau t );
void echanger ( int *px, int *py );

int main () {

  Tableau tab;

  saisirTableau ( tab );
  triAbulles ( tab );
  printf ( "Voici le tableau trié : " );
  afficherTableau ( tab );
  return EXIT_SUCCESS;
}

void triAbulles ( Tableau t ) {

  int i = 0, j;
  Booleen aucunEchange = FAUX;

  while ( ( i<LONGUEUR-1 ) && !aucunEchange ) {

    aucunEchange = VRAI;

    for ( j=LONGUEUR-1; j>i; j-- )
      if ( t[j]<t[j-1] ) {
        echanger ( &t[j], &t[j-1] );
        aucunEchange = FAUX;
      }
    i++;
  }
}
```

triAbulles.c (fin)

```
void saisirTableau ( Tableau t ) {

  int i;

  for ( i=0; i<LONGUEUR; i++ ) {
    printf ( "t[%02d] : ", i );
    scanf ( "%d", &t[i] );
  }
}

void afficherTableau ( Tableau t ) {

  int i;

  for ( i=0; i<LONGUEUR; i++ )
    printf ( "%d ", t[i] );
  printf ( "\n" );
}

void echanger ( int *px, int *py ) {

  int aux = *px;

  *px = *py;
  *py = aux;
}
```

ii. vous devez découpez ce fichier pour créer :
   – le module *tabio* :
     – *tabio.h*
     – *tabio.c*
   – le fichier *triAbulles.c* comprenant uniquement les définitions des fonctions *main*, *triAbulles* et *echanger*

iii. Vous devez ensuite écrire le fichier *makefile* et taper la commande *make* pour créer l'exécutable *triAbulles*

(b) **Application** *secondDegre* utilisant le module *complexe*

Vous sont fournis :

– le fichier *complexe.h* qui contient la définition du type *Complexe* et la déclaration des fonctions C le gérant

complexe.h (début)

```
// Définition du type Complexe

typedef struct { double re;
                 double im; } Complexe;
```

complexe.h (fin)

```
/* Déclaration des fonctions implémentant les opérations
   sur le type Complexe */

Complexe nouveauComplexe(double x, double y);
double partieReelle(Complexe z);
double partieImaginaire(Complexe z);
double argument(Complexe z);
double module(Complexe z);
Complexe oppose(Complexe z);
Complexe inverse(Complexe z);
Complexe somme(Complexe z1, Complexe z2);
Complexe difference(Complexe z1, Complexe z2);
Complexe produit(Complexe z1, Complexe z2);
Complexe quotient(Complexe z1, Complexe z2);
Complexe racineCarree(Complexe z);
void afficherComplexe(Complexe z);
```

– le fichier objet *complexe.o* qui contient les définitions compilées des fonctions déclarées dans *complexe.h*

Vous devez :

– créer sous votre répertoire de connexion un répertoire *include* et y placer le fichier *complexe.h*

– placer le fichier *complexe.o* dans votre répertoire de travail (celui ou vous aller stocker le fichier *secondDegre.c*

– développer l'application *secondDegre.c* qui détermine les racines d'une équation du second degré sur $\mathbb{C}$

– Ecrire le fichier *makefile* et taper la commande *make* pour obtenir l'application *secondDegre*

2. **Ce que vous devez faire durant la séance de TP**

   Finir l'application *secondDegre* utilisant le module *complexe*

3. **Pour aller plus loin**

   (a) Le zéro d'une fonction par dichotomie en récursif [4.4]

   (b) Le tri de trois couleurs [3.4]

C8

LES BIBLIOTHÈQUES ET LA MISE AU POINT

2

Objectifs du C8

- Savoir utiliser des bibliothèques
- Maîtriser l'utilisation d'un metteur au point

Contenu du C8

3

C8

1 Les bibliothèques

4

5

1.1 les bibliothèques

- **Bibliothèque** : regroupement de fichiers *objet*
- **Deux sortes** :
  - statique : fichier d'extension **.a** (archive)
  - dynamique : fichier d'extension **.so** (*shared object*)
  - sujet traité dans le module *AM12*
- **Localisation** :
  - standard : répertoire */usr/lib*
  - non standard → variable *LD_LIBRARY_PATH* (bibliothèque dynamique)
    exemple :
    `$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib`

– **Bibliothèques statiques ou dynamiques**
Durant le module AP11, vous n'utiliserez que des bibliothèques dynamiques. Vous comprendrez la différence entre les deux lors du module AM12.

1.2 Options de la commande gcc

6

- **Préprocesseur**
  - Option **`-I`** : permet de spécifier où trouver les fichiers d'en-tête quand ils ne sont pas dans le répertoire « standard » */usr/include*.
- **Edition des liens**
  - Option **`-l`** : permet de prendre en compte une bibliothèque
  - Option **`-L`** : permet de spécifier où trouver les fichiers de bibliothèque quand elles ne sont pas dans le répertoire « standard » */usr/lib*.
- **Un exemple** : la bibliothèque *Xwindow* de base (la *Xlib*)

  Les fichiers d'en-tête et de bibliothèque se trouvent dans les répertoires :
  - */usr/include/X11* → fichier *Xlib.h*
  - */usr/X11R6/lib* → fichier *libX11.so*

  Pour produire l'exécutable *appliX11* à partir du fichier source *appliX11.c*, il faudra saisir la commande :
  - **`gcc -Wall -I/usr/include/X11 -L/usr/X11R6/lib -lX11 appliX11.c -o appliX11`**

– **Option -I**
Il est également possible de spécifier le chemin d'accès vers le fichier d'en-tête dans la directive **`#include`** :
– **`#include "/usr/include/X11/X11.h"`**
– ou **`#include <X11/X11.h>`** : puisque le répertoire */usr/include/X11* est sous le répertoire */usr/include*

C8

2 La mise au point

7

8

2.1 Les méthodes

■ **La méthode manuelle**

- ajout d'appels de *printf* dans le source
  - pour afficher le contenu des variables
    ```
    printf ( "nb = %d\n", nb );
    ```
  - pour suivre les sections exécutées
    ```
    printf ( "étape 2 : OK\n"
    );
    ```
  - il faut recompiler
  - il faudra retirer ces appels
- la compilation **conditionnelle**

```
#ifdef DEBUG
   printf ( "nb = %d\n", nb );
#endif
```

■ **Le metteur au point**

- il permet
  - de lancer l'exécution du programme
  - de poser des points d'arrêt, ce qui provoque la suspension de l'exécution
  - d'afficher le contenu des variables
  - de modifier le contenu des variables
  - d'exécuter le programme pas à pas
  - de visualiser l'enchaînement des appels de fonction à la réception d'un signal d'erreur : violation de segmentation
  - de lister le source du programme
- Il dispose d'une aide en ligne
- Pour l'utiliser pleinement **compilation** avec l'option `-g`

– **Ajout de *printf* dans le source**
  – il faut terminer la chaîne *format* de la fonction *printf* par le caractère '\n'.
  – les entrées-sorties sont « bufferisées »; les caractères à afficher dans la fenêtre *shell* sont temporairement stockées dans des zones mémoire de l'application et du système.
  Si votre message ne se termine pas par le caractère '\n', il n'y a pas de garantie d'affichage au cas où votre programme génère une erreur système (violation de segmentation par exemple).

– **Le metteur au point**
  – il vous permettra de gagner du temps pour corriger les erreurs d'exécution de votre application.

2.2 Metteurs au point

9

■ *gdb* **en mode texte**

- ♦ appel : **`gdb nomExecutable`**
- ♦ principales requêtes :
  - ► **`help`** pour l'aide en ligne
  - ► **`r`** (**`run`**) lance l'exécution du programme
  - ► **`b`** (**`breakpoint`**) pose un point d'arrêt
  - ► **`c`** (**`continue`**) poursuit l'exécution
  - ► **`n`** (**`next`**) exécute l'instruction suivante ou le corps de la fonction appelée
  - ► **`s`** (**`step`**) exécute l'instruction suivante
  - ► **`k`** (**`kill`**) arrête l'exécution
  - ► **`p`** (**`print`**) affiche le contenu d'une variable
  - ► **`l`** (**`liste`**) affiche le fichier source
  - ► **`bt`** (**`back track`**) affiche la pile
  - ► **`q`** (**`quit`**) pour quitter

■ *ddd* **interface graphique de** *gdb*

- ♦ appel : **`ddd nomExecutable`**
- ♦ utilisation de la souris :
  - ► bouton *Run* lance l'exécution
  - ► clic droit → *Break* pose un point d'arrêt
  - ► bouton *Cont* poursuit l'exécution
  - ► bouton *Next* exécute l'instruction suivante ou la fonction en entier
  - ► bouton *Step* exécute l'instruction suivante
  - ► bouton *Interrupt* arrête l'exécution
  - ► clic droit sur la variable → *Print* affiche le contenu de la variable
  - ► menu *Status* → *Backtrace* affiche la pile des appels

– **On ne peut pas lister ici toutes les possibilités de ces outils**
  – initiation lors du **TP6**

TP6

LES EXERCICES À CODER

Exercices à réaliser

1. **Ce que vous devez faire avant la séance de TP**

   (a) Implémenter l'algorithme du tri rapide en utilisant le module *tabio*. Vous disposez des fichiers *tabio.h* et *tabio.c*. Vous utiliserez le fichier *makefile* générique créé lors du TP5 pour produire l'application *triRapide*.

   (b) Implémenter l'algorithme du tri par insertion en utilisant le module *tabio*. Pour cet exercice, vous disposez du fichier d'en-tête *tabio.h* et du fichier de bibiothèque *libtabio.so*. Vous devrez :
   - créer sous votre répertoire de connexion un répertoire *include* et un répertoire *lib*
   - placer le fichier *tabio.h* dans le répertoire *include*
   - placer le fichier *libtabio.so* dans le répertoire *lib*
   - écrire dans votre répertoire de travail le fichier *triInsertion.c*
   - écrire le fichier *makefile* correspondant en utilisant les options **`-I`**, **`-l`** et **`-L`** de la commande *gcc*

2. **Ce que vous devez faire durant la séance de TP** : **Mise au point**

   Le programmme ci-dessous est censé déterminer le nombre de singletons dans un tableau d'entiers.

   Soit ***t[i]*** l'élément courant : $si\ \exists\ j,\ i < j < N,\ tel\ que\ t[j] = t[i]\ alors\ t[i]\ n'\ est\ pas\ un\ singleton$

   Le tableau de booléens *dejaVu* permet de « marquer » les cases du tableau contenant une valeur déjà rencontrée.
   Hélas, le programme répond désespérément zéro alors qu'il y a cinq singletons dans le tableau *tab* défini et initialisé dans la fonction *main*.

   Le concepteur de ce programme aurait-il la moyenne dans le module AP11 s'il avait à le valider ?
   Pouvez-vous l'aider afin que son programme fonctionne correctement ?

nbSingletons.c

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

#define LONGUEUR 10

typedef enum{FAUX,VRAI} Booleen;

int nbSingletons(int t[LONGUEUR]);

int main() {

  int tab[LONGUEUR]={3,2,8,7,5,4,2,7,2,6};

  printf("Il y a \%d singleton[s] dans ce tableau\n", nbSingletons(tab) );
  return EXIT_SUCCESS;
}
```

nbSingletons.c (suite)

```
int nbSingletons(int t[LONGUEUR]) {

  int i=0, nombre=0;
  Booleen dejaVu[LONGUEUR]; // L'élément a-t-il été déjà rencontré ?

  for(i=0;i<LONGUEUR;i++)
    dejaVu[i]=FAUX; // non au début de l'algorithme

  while( i<LONGUEUR-1 ) {
    if( dejaVu[i]==FAUX ) { // L'élément n'a pas encore été rencontré

      int j=i+1;
      Booleen unAutre=FAUX; // pour voir si on trouve la même valeur
                            // plus loin dans le tableau
      dejaVu[i]=VRAI;       // On marque l'élément rencontré
      while( j<LONGUEUR ) {
        if( t[j]==t[i++] ) {
          unAutre=VRAI;     // On a trouvé la même valeur
          dejaVu[j]=VRAI;   // On marque l'élément rencontré
        }
        j++;
      }
      if( unAutre==FAUX )   // On a trouvé un singleton
        nombre++;
    }
  }
  return nombre;
}
```

Vous devez :

(a) compiler ce fichier avec l'option **-g**
$ gcc -Wall -g nbSingletons.c -o nbSingletons

(b) lancer la commande *ddd* (interface graphique de la commande *gdb* d'*Unix*)
$ ddd nbSingletons &

(c) commencer par poser un point d'arrêt au début de la fonction *nbSingletons*

(d) cliquer sur le bouton *Run*

(e) afficher de manière permanente le contenu des variables *i*, *nombre* et *dejaVu* en cliquant dessus avec le bouton droit et en sélectionnant l'item *Display*

(f) exécuter le programme pas à pas en cliquant sur le bouton *Step* pour trouver et corriger les erreurs

Voici un complément d'information sur l'utilisation de la commande *ddd* :

(a) **Gestion des points d'arrêt**

i. **création d'un point d'arrêt** : clic droit en début de ligne et sélection de l'item *Set Breakpoint*

ii. **suppression d'un point d'arrêt** : clic droit sur l'icône *STOP* et sélection de l'item *Delete Breakpoint*

(b) **Exécution**

i. **lancement du programme** : clic gauche sur le bouton *Run* ; s'il n'y a pas de point d'arrêt, le programme sera exécuté intégralement

ii. **continuer l'exécution** : clic gauche sur le bouton *Cont*

iii. **exécution pas à pas** :

A. clic gauche sur le bouton *Step* (si l'instruction à exécuter est un appel de fonction, celle-ci sera également exécutée pas à pas)

B. clic gauche sur le bouton *Next* (si l'instruction à exécuter est un appel de fonction, celle-ci sera exécutée intégralement)

iv. **arrêter l'exécution** : clic gauche sur le bouton *Interrupt*

(c) **Visualisation du contenu des variables scalaires et tableaux**

i. **temporaire** : clic sur l'identificateur de la variable (définition ou utilisation) et sélection de l'item *Print* → affichage dans la sous-fenêtre *gdb* du bas

ii. **permanente** : clic sur l'identificateur de la variable (définition ou utilisation) et sélection de l'item *Display* → affichage dans la sous-fenêtre du haut (il faut parfois agrandir celle-ci et déplacer avec la souris les objets qui s'y trouvent)

(d) **Visualisation de la pile d'appel des fonctions**

i. sélection du menu *Status* et de l'item *Backtrace*

3. **Pour aller plus loin**

(a) La recherche d'un élément dans un tableau en récursif [4.6]

(b) La moyenne des notes [2.11]

C9

LES FICHIERS

2

Objectifs du C9

- Comprendre l'organisation des fichiers
  - la relation entre l'application et le système d'exploitation
  - les opérations d'ouverture et de fermeture
  - les opérations de lecture et d'écriture
  - l'accès séquentiel et l'accès direct
- Savoir utiliser les fonctions d'entrées-sorties proposées par la bibliothèque standard

Contenu du C9

3

1 Généralités sur les fichiers

4

1.1 Fichiers et langages de programmation -

5

- **Nécessité des fichiers**
  - un programme est capable de manipuler des données en mémoire centrale
  - il doit pouvoir également lire et écrire des données sur ou à partir des supports de stockage
- **Solution**
  - le langage de programmation offre un **type** spécial que nous nommerons **Fichier**
  - les **opérations** associées à ce type permettront de manipuler les fichiers
  - dans un programme, un **fichier** sera une **variable** de type **Fichier**
- **Interface avec le système de gestion de fichiers (SGF)**
  - les opérations offertes par le langage de programmation sont implémentées en utilisant les services du SGF du système d'exploitation
  - une opération d'**assignation** permet de faire le lien entre le **nom du fichier** sur le disque et une **variable** de type **Fichier** dans le programme

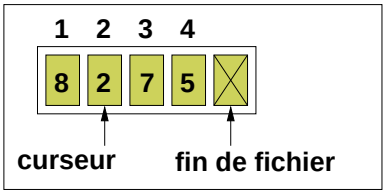
6

1.2 Organisation et ouverture en lecture -

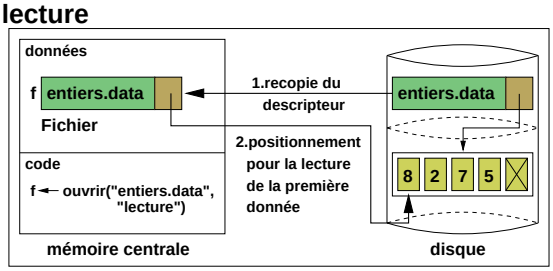
■ **Organisation et accès**

- ♦ les données sont rangées en séquence
- ♦ un curseur repère la donnée courante
- ♦ accès séquentiel et accès direct
- ♦ opérations de lecture et d'écriture
- ♦ une marque de fin de fichier

■ **Exemple** : un fichier d'entiers



■ **Ouverture en lecture**



– l'ouverture du fichier est l'opération préalable avant toute tentative de lecture ou écriture dans ce fichier.

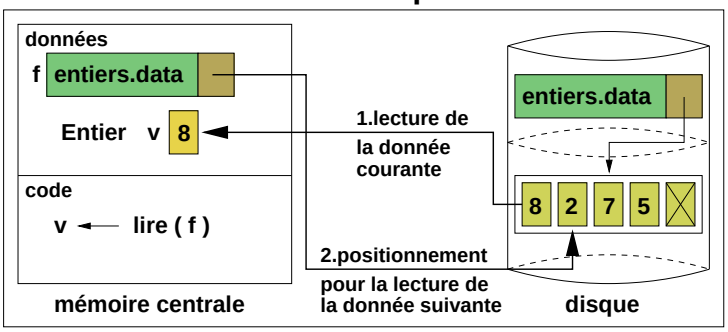
– **Organisation et accès**
  – la notion de **curseur** (i.e. position courante) est intégrée dans la variable de type **Fichier**. Le curseur est automatiquement positionné sur la donnée suivante par une opération de lecture.
  – accès séquentiel : lecture de la donnée courante.
  – accès direct : le curseur est positionné directement à un emplacement spécifié du fichier avant d'opérer la lecture.
  – marque de fin de fichier : c'est généralement une marque « logique » (la valeur du curseur et la taille du fichier suffisent à déterminer si la fin du fichier est atteinte).

– **Ouverture en lecture**
  – vérification de l'existence du fichier et des droits d'accès.
  – recopie partielle du descripteur de fichier dans la variable de type **Fichier** de l'application.

– **Echecs possibles** : erreur de nom, de droits,. . .

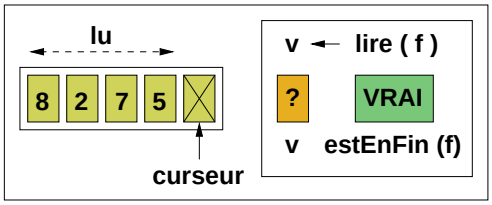
7

1.3 Lecture séquentielle -



■ **Gestion de la fin de fichier**

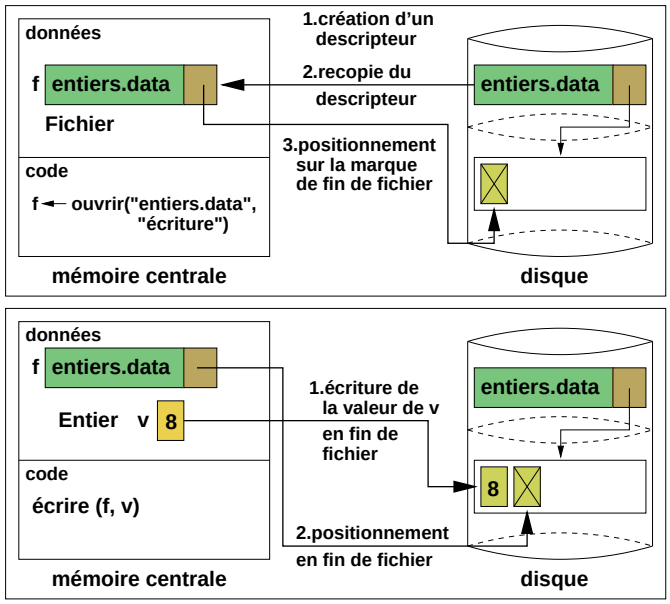
- ♦ **v ← lire (f)**
- ♦ la dernière tentative de lecture échoue
- ♦ le contenu de *v* est indéterminé
- ♦ l'opération **estEnFin (f)** renvoie la valeur booléenne VRAI



– **Fin de fichier**
  – L'opération *estEnFin* n'accède pas elle-même au fichier mais s'appuie sur le résultat de la dernière opération de lecture.
  – une tentative de lecture avec échec est donc nécessaire pour que l'opération *estEnFin* renvoie la valeur VRAI.

8

1.4 Ouverture en création et écriture -

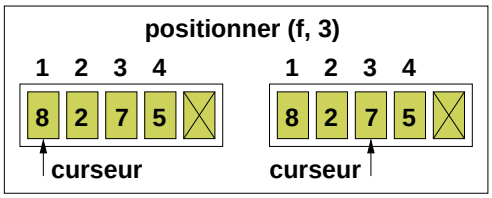


1.5 Accès direct et fermeture d'un fichier -

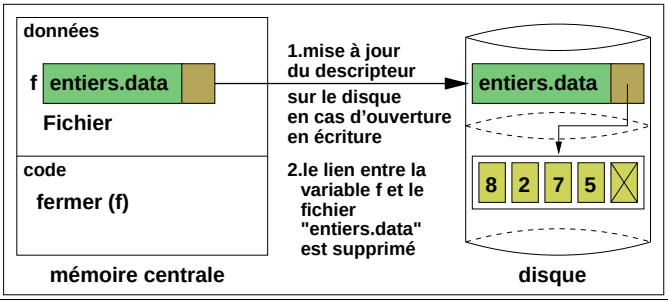
9

■ **Accès direct**

- ♦ **positionner (f, p)**
- ♦ positionne le curseur sur l'élément de rang *p*
- ♦ *p* doit être une position valide du fichier



■ **Fermeture**



– **Mise à jour du descripteur à la fermeture du fichier**
  – taille du fichier
  – heure et date du dernier accès et/ou de la dernière modification

1.6 Spécification du type Fichier -

10

- **type** Fichier **de** TypeValeur
- **ouvrir : Chaîne** $\times$ **Chaîne** $\rightarrow$ **Fichier**
  - premier argument de type Chaîne : nom du fichier
  - deuxième argument de type Chaîne : mode d'ouverture
  - les modes d'ouvertures :
    - "**lecture**" : le fichier est ouvert en lecture s'il existe et s'il est accessible, sinon erreur ! ! ! Positionnement du curseur en début de fichier
    - "**écriture**" : le fichier est ouvert en écriture et le curseur est positionné au début de celui-ci. S'il n'existe pas il y aura création, sinon risque d'écrasement des données
    - "**ajout**" : le fichier est ouvert en écriture et le curseur est positionné en fin de celui-ci
- **fermer : Fichier** $\rightarrow$
  - rompt le lien entre la variable de type Fichier et le fichier sur le disque

– **Mode** *ajout*
  – possibilité d'écrire à la fin d'un fichier existant

1.7 Spécification (fin) et algorithme de lecture -

11

- **lire : Fichier → Fichier × TypeValeur**
  - ♦ lit l'élément à la position du curseur
  - ♦ erreur si celui-ci est sur la marque de fin de fichier
- **écrire : Fichier × TypeValeur → Fichier**
  - ♦ écrit l'élément à la position du curseur
- **estEnFin : Fichier → Booléen**
  - ♦ renvoie VRAI après une lecture en échec, FAUX sinon
- **positionner : Fichier × Naturel → Fichier**
  - ♦ modifie la position du curseur
  - ♦ l'argument de type Naturel doit être une position autorisée

- **Algorithme de lecture**

```
f : Fichier // fichier d'entiers
v : Entier

f ← ouvrir ( "entiers.data", "lecture" )
v ← lire ( f )

tant que non estEnFin ( f ) faire
    afficher ( v )
    v ← lire ( f )
ftq

fermer ( f )
```

– **Algorithme de lecture**
  – remarquez le premier appel de l'opération *lire* avant de pouvoir tester la fin de fichier.

2 Les fichiers en C

12

13

2.1 Le type FILE

- **Le fichier d'en-tête** *stdio.h* **contient** :
  - la définition du type **FILE**
  - la définition de la constante symbolique **EOF**
    - `#define EOF (-1)`
  - la définition de la constante symbolique **NULL**
    - `#define NULL 0`
  - les prototypes des fonctions associées au type **FILE**
- **Définition en C d'une variable de type Fichier**

```
#include <stdio.h>

FILE *f; // f est une adresse d'un objet de type FILE
```

- **Type FILE**
  - type structure dont l'un des champs est un « pointeur » sur l'élément courant dans le fichier.
  - L'utilisateur n'a pas à connaître les détails de l'implémentation mais seulement l'interface (prototype des fonctions et sémantique décrite dans le manuel).
- **Constante symbolique NULL**
  - plus précisément , elle est définie dans le fichier */usr/include/linux/stddef.h* qui est lui-même inclus dans le fichier *stdio.h*

2.2 Fichier texte : ouverture et fermeture

- Fichier texte
  - ♦ séquence de caractères affichables
  - ♦ accès séquentiel
  - ♦ les fonctions d'accès renvoient **`EOF`** ou **`NULL`** en cas d'échec
- Entrées-sorties
  - ♦ caractère par caractère
  - ♦ ligne par ligne
  - ♦ formatées

14

```
FILE *f; // FILE *stream; (flux d'entrée-sortie)
```

- **Ouverture d'un fichier en lecture**

```
f=fopen("entiers.txt","r"); // f contient la valeur NULL si échec
```

- **Ouverture d'un fichier en écriture**

```
f=fopen("entiers.txt","w"); // s'il existe, il est écrasé
```

- **Autres modes d'ouverture**
  - ♦ **`"a"`** : écriture à la fin du fichier existant
  - ♦ **`"r+"`** ou **`"a+"`** : lecture et écriture sur un fichier existant
- **Fermeture d'un fichier ouvert**

```
fclose( f );
```

– **Ouverture d'un fichier**
  – quel que soit le mode d'ouverture, il faut absolument tester la valeur de retour de la fonction *fopen* :

```
f = fopen( "entiers.txt", "r" );
if ( f == NULL ) {
   printf ( "Erreur ouverture de fichier\n" );
   exit ( EXIT_FAILURE );
}
```

ou, en parenthésant l'affectation,

```
if ( ( f = fopen( "entiers.txt", "r" ) ) == NULL ) {
   printf ( "Erreur ouverture de fichier\n" );
   exit ( EXIT_FAILURE );
}
```

– **Problème d'accès aux fichiers et fin de fichier**
  – **`perror ( "Problème fichier" );`** → affiche sur le flux des erreurs la chaîne "Problème fichier" ainsi que la cause du problème.
  – **`if ( feof ( f) )`** → *feof* renvoie une valeur non nulle en fin de fichier

2.3 Entrées-sorties orientées caractère

15

■ **Lecture d'un caractère** : fonction *fgetc*

- prototype : **`int fgetc (FILE *stream);`**
- *fgetc* renvoie un caractère lu comme un **`unsigned char`** (ce qui permet de prendre en compte les codages des caractères sur 8 bits) et « **casté** » en **`int`** (pour **`EOF`**)
- *fgetc* renvoie **`EOF`** à la fin du fichier ou en cas d'erreur
- usage courant (la variable réceptacle est de type **`char`**)

```
char c;
c = fgetc (f);
```

■ **Ecriture d'un caractère** : fonction *fputc*

- prototype : **`int fputc (int c, FILE *stream);`**
- *fputc* écrit la valeur **`c`** « **casté** » en **`unsigned char`** sur le flux **`stream`**
- *fputc* renvoie **`EOF`** en cas d'erreur
- usage courant (on teste rarement la valeur de retour)

```
char c;
fputc (c, f);
```

– Le « **cast** » (ou conversion de type) consiste à modifier l'interprétation ou l'organisation de la représentation mémoire d'une valeur d'un certain type pour l'utiliser comme valeur d'un autre type.
Par exemple, un « cast » du type **`char`** vers le type **`int`** élargit simplement la représentation sans modifier la valeur. Ce qui permet de distinguer de cas d'erreur **`EOF`** (valeur hexadécimale FFFFFFFF) du caractère dont le code hexadécimal est égal à FF.
En revanche, un « cast » du type **`int`** vers le type **`char`** se traduit par une troncature ne conservant que la partie « basse » de la valeur initiale : la valeur obtenue n'est donc pas nécessairement la même.

– **Utilisation courante de la fonction *fgetc* pour lire un fichier**

```
char c;

while ( ( c = fgetc(f) )!=EOF ) {
       // ...
}
```

– **Pour être plus précis**
Si dans votre fichier le caractère dont le code hexadécimal est égal à FF est présent, l'agorithme de lecture se terminera prématurément. D'où :

```
char c;
int retour;

while ( ( retour = fgetc(f) )!=EOF ) {
       c = (char) retour;
       // ...
}
```

2.4 Entrées-sorties orientées ligne

16

- Une **ligne** est une séquence de caractères terminée par le caractère '**\n**'
- **Lecture d'une ligne** : fonction *fgets*
  - prototype : **`char *fgets (char *s, int size, FILE *stream);`**
  - *fgets* lit au plus **`size`**-1 caractères à partir du flux **`stream`** et les place dans le tampon **`s`** ; elle s'arrête dès qu'un caractère '**\n**' est rencontré ou à la fin du fichier
  - un caractère de terminaison '**\0**' est placé après les caractères lus
  - *fgets* renvoie la valeur **`NULL`** à la fin du fichier ou en cas d'erreur
  - usage courant (la valeur de retour est directement testée)

```
#define TAILLE_TAMPON (80+1) // 80 caractères utiles
char tampon[TAILLE_TAMPON]; // réservation mémoire
while ( (fgets(tampon, TAILLE_TAMPON, f))!=NULL ) { ...}
```

- **Ecriture d'une ligne** : fonction *fputs*
  - prototype : **`int fputs (const char *s, FILE *stream);`**
  - *fputs* écrit la chaîne de caractères **`s`** sur le flux **`stream`** ; le caractère '**\0**' n'est pas écrit
  - *fputs* renvoie **`EOF`** en cas d'erreur
  - usage courant (l'écriture s'arrête dès que le caractère '**\0**' est rencontré)

```
fputs(tampon, f);
```

– **Fonction** *fgets*
  – elle est « **sécurisée** » ; le paramètre **`size`** permet d'éviter ce qu'on appelle un « **débordement de tampon** » (buffer overflow) → écrire en dehors du tableau **`tampon`**.
  – à noter qu'il faut toujours penser à réserver un octet supplémentaire pour le caractère de terminaison '\0' :
  `#define TAILLE_TAMPON (80+1)`
  – c'est ici l'occasion de mentionner que l'expression associée à la constante symbolique doit être systématiquement parenthésée.

– **Fonction** *fputs*
  – remarquez la présence du modificateur **`const`** devant le type du paramètre **`s`** ; le contenu de **`s`** ne peut ici être modifié (contrairement à *fgets* bien sûr).
  – s'il n'y a pas de caractère '\0' dans le tableau **`tampon`**, la fonction *fputs* va continuer de parcourir la zone mémoire au delà du tableau. Dans le meilleur des cas, elle finira par rencontrer un caractère '\0', sinon elle provoquera une erreur d'accès mémoire.
  Dans les deux cas, vous aurez n'importe quoi dans votre fichier.

17

2.5 Entrées-sorties formatées

- Le format du fichier doit être **connu** (lecture) ou **choisi** (écriture) par le programmeur
- Exemple : un entier (l'âge) et une chaîne d'au plus 20 caractères (le nom) d'une personne

```
#define TAILLE_NOM (20+1) // +1 pour le caractère '\0'
int age;
char nom[TAILLE_NOM];
```

- **Lecture formatée** : fonction *fscanf*
  - prototype : `int fscanf ( FILE *stream, const char *format, ...);`
  - *fscanf* lit à partir du flux `stream` et réalise les conversions spécifiées par la chaîne `format` ;
  - les autres arguments, en nombre variable, sont des expressions-adresses
  - *fscanf* renvoie la valeur `EOF` à la fin du fichier ou en cas d'erreur
  - usage courant : `fscanf( f, "%d%s", &age, nom );`
- **Ecriture formatée** : fonction *fprintf*
  - prototype : `int fprintf ( FILE *stream, const char *format, ...);`
  - *fprintf* écrit sur le flux `stream` en réalisant les conversions spécifiées par la chaîne `format`
  - les autres arguments, en nombre variable, sont des expressions
  - *fprintf* renvoie le nombre de caractères écrits
  - usage courant : `fprintf( f, "%d %s\n", age, nom );`

– **Utilisation des fonctions** *fscanf* **et** *fprintf*
  – mise à part le premier paramètre de type `FILE *`, elle est identique à celle des fonctions *scanf* et *printf*.

– **Valeur de retour de la fonction** *fscanf*
  – pour être précis, si la fin de fichier n'est pas atteinte, *fscanf* renvoie le nombre de « **correspondances** » réussies, le nombre demandé étant le nombre de spécifications de conversion de format.
  – dans l'exemple du transparent, si l'on voulait vérifier la lecture, il faudrait comparer la valeur de retour à 2 :

```
if ( ( fscanf( f, "%d%s", &age, nom )!= 2 ) {
    printf ( "Erreur lecture\n" );
    exit ( EXIT_FAILURE );
}
```

  – ceci est également valable pour les fonctions *scanf* et *sscanf*.

– **Lecture d'une chaîne de caractères** avec les fonctions *scanf*, *fscanf* et *sscanf*
  – pour ces fonctions, une chaîne de caractères ne comporte pas de caractère « **blanc** » et est délimitée par deux blancs, ceux-ci étant l'espace, la tabulation ('\t') et le saut à la ligne ('\n').

2.6 Les entrées-sorties standards

18

- **L' entrée standard**
  - correspond matériellement au clavier
  - vu comme un fichier texte
  - variable prédéfinie **`stdin`** de type **`FILE *`**
  - fonctions de lecture : *scanf*, *getchar*, . . .
- **La sortie standard**
  - correspond à la fenêtre *shell* d'où est lancé le processus
  - vu comme un fichier texte
  - variable prédéfinie **`stdout`** de type **`FILE *`**
  - fonctions d'écriture : *printf*, *putchar*, *puts*, . . .
- **La sortie des erreurs standard**
  - correspond à la fenêtre *shell* d'où est lancé le processus
  - vu comme un fichier texte
  - variable prédéfinie **`stderr`** de type **`FILE *`**
  - permet de différencier l'affichage du résultat d'une commande des messages d'erreurs qu'elle peut produire (cf les redirections du *shell*)

– **stdin**
  – **`scanf ( "%d", &nb );`** $\Longleftrightarrow$ **`fscanf ( stdin, "%d", &nb );`**
  – **`c = getchar ( );`** $\Longleftrightarrow$ **`c = fgetc ( stdin );`**
  – il existe également la fonction *gets* mais elle ne gére pas les débordements de buffer (elle n'a pas de paramètre **`size`**).
  → lire la section « **BOGUES** » du manuel de *gets*
  pour réaliser la lecture d'une ligne à partir du clavier, utilisez *fgets* :
  `fgets( tampon, TAILLE_TAMPON, stdin );`
  et non
  `gets( tampon );`

– **stdout**
  – **`printf ( "Bonjour\n" );`** $\Longleftrightarrow$ **`fprintf ( stdout, "Bonjour\n" );`**
  – **`putchar ( c );`** $\Longleftrightarrow$ **`fputc ( c, stdout );`**
  – **`puts ( tampon );`** $\Longleftrightarrow$ **`fputs ( tampon, stdout );`** (ajout d'un '\n' pour *puts*)

– **stderr**
  – maintenant qu'on connaît son existence, on écrira plutôt :

```
if ( ( fscanf( f, "%d%s", &age, nom )!= 2 ) {
   fprintf ( stderr, "Erreur lecture\n" );
   exit ( EXIT_FAILURE );
}
```

2.7 Fichier binaire : ouverture et fermeture

19

- Fichier binaire
  - ♦ séquence d'octets non interprétés
  - ♦ accès séquentiel et direct
- Interprétation des données binaires
  - ♦ à la charge du programmeur
  - ♦ une séquence de 8 octets peut correspondre à :
    - ► deux valeurs de type **`int`**
    - ► une valeur de type **`double`**
    - ► un tableau de 8 caractères . . .
- **Ouverture d'un fichier en lecture**

```
f=fopen("entiers.bin","rb"); //f contient la valeur NULL si échec
```

- **Ouverture d'un fichier en écriture**

```
f=fopen("entiers.bin","wb"); // s'il existe, il est écrasé
```

- **Autres modes d'ouverture**
  - ♦ **`"ab"`** : écriture à la fin du fichier existant
  - ♦ **`"r+b"`** ou **`"a+b"`** : lecture et écriture sur un fichier existant
- **Fermeture d'un fichier ouvert**

```
fclose( f );
```

– **Accès direct dans un fichier binaire**
  – un fichier binaire est plus ou moins **structuré** :
    – une séquence de valeurs de type **`int`**
    – une séquence d'objets de type structure
    – une organisation plus complexe avec une **en-tête** et différentes **sections**
  – l'accès **direct** est alors utilisable

– **Ouverture d'un fichier binaire**
  – le caractère '**b**' comme binaire que l'on rajoute au deuxième paramètre (la chaîne spécifiant le mode d'ouverture) de la fonction *fopen* n'est utile que pour la compatibilité avec des systèmes comme *DOS/Windows*.
  – sous *Unix*, il ne sert à rien car il n'existe pas de **code de fin de fichier texte**.
  – sous *DOS/Windows*, il y en a un. Si on ouvre un fichier binaire en « mode texte », il y a toutes les chances pour qu'un octet du fichier corresponde à ce caractère de fin de fichier ; la lecture du fichier sera alors prématurément interrompue.
  – rajoutez-le quand même pour assurer la portabilité de vos programmes.

2.8 Fichier binaire : lecture

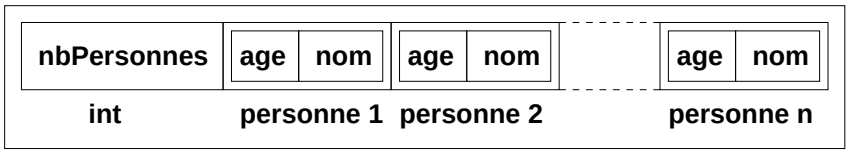
20

- Le format du fichier doit être **connu** (lecture) ou **choisi** (écriture) par le programmeur
- Exemple : un fichier de personnes

```
#define TAILLE_NOM (20+1) // +1 pour le caractère '\0'
typedef struct { int age;
                 char nom[TAILLE_NOM]; } Personne;
Personne personne;
```

- **Lecture** : fonction *fread*
  - prototype :
    `size_t fread (void *ptr, size_t size, size_t nmemb, FILE *stream);`
    `size_t` est un renommage (ANSI) du type `unsigned int`
  - *fread* lit à partir du flux `stream` un nombre `nmemb` d'objets de taille `size` octets chacun et les stocke à l'emplacement `ptr`
  - *fread* renvoie le nombre d'objets correctement lus
  - usage courant : `fread( &personne, sizeof(Personne), 1, f);`

– **Structuration d'un fichier de personnes**
  – le nombre de personnes *nbPersonnes* de type `int`
  – *nbPersonnes* éléments de type *Personne*



– **Lecture du fichier de personnes**

```
FILE *f;
int nbPersonnes;
Personne *tabPersonnes;

// ouverture du fichier avec test de la valeur de retour de la fonction fopen
// lecture du nombre de personnes
fread ( &nbPersonnes, sizeof(int), 1, f);
// allocation dynamique pour le tableau de personnes avec test
// de la valeur de retour de la fonction malloc (voir C10)
// lecture globale du tableau de personnes
fread ( tabPersonnes, sizeof(Personne), nbPersonnes, f);
```

2.9 Fichier binaire : écriture et accès direct

21

- **Ecriture** : fonction *fwrite*
  - prototype :
    ```
    size_t fwrite (const void *ptr,size_t size,size_t nmemb,FILE *stream);
    ```
  - *fwrite* écrit sur le flux **stream** un nombre **nmemb** d'objets de taille **size** octets chacun, ces objets étant stockés à l'emplacement **ptr**
  - *fwrite* renvoie le nombre d'objets correctement écrits
  - usage courant : `fwrite( &personne, sizeof(Personne), 1, f);`
- **Accès direct** : fonction *fseek*
  - prototype : `int fseek (FILE *stream, long offset, int whence);`
  - *fseek* modifie la position courante sur le flux **stream** ; la nouvelle position est obtenue en additionnant **offset** octets au point de départ **whence**, **whence** indiquant le début (**SEEK_SET**), la position courante (**SEEK_CUR**) ou la fin (**SEEK_END**) du fichier
  - *fseek* renvoie -1 en cas d'erreur, sinon 0
  - usage courant :
    ```
    long position = 3; // troisième élément du fichier
    fseek ( f, (position-1) * sizeof(Personne), SEEK_SET);
    ```

– **Ecriture du fichier de personnes**

```
// écriture du nombre de personnes
fwrite ( &nbPersonnes, sizeof(int), 1, f);
// écriture globale du tableau de personnes
fwrite ( tabPersonnes, sizeof(Personne), nbPersonnes, f);
```

– **Accès direct**
  – positionnement en début de fichier :
    `fseek ( f, 0, SEEK_SET);`
    ou plus simplement
    `rewind ( f );`
  – positionnement en fin de fichier :
    `fseek ( f, 0, SEEK_END);`

C9

Exercice d'algorithmique sur les fichiers à préparer

Calcul du chiffre d'affaire

Vous disposez de deux fichiers texte *produits.txt* et *ventes.txt*. Le premier fichier contient la liste des produits avec le *code* de type Naturel, le *nom* de type chaîne et le *prix* de type réel. Le fichier ventes.txt contient une liste de codes et de nombres d'unités vendues, un même code pouvant apparaître éventuellement plusieurs fois dans le fichier ou pas du tout. Les deux fichiers sont triés suivant les codes croissants.
Définir une procédure *calculerCA* qui prend en entrée deux paramètres *fProduits* et *fVentes* de type Fichier et qui affiche, pour chaque produit vendu, son code, son nom, le nombre d'unités vendues et le chiffre d'affaire réalisé pour le produit. Elle affichera finalement le chiffre d'affaire total. Le fait que les deux fichiers soient triés par codes croissants permet une seule lecture séquentielle des deux fichiers.
Définir la fonction *principale* qui ouvre les deux fichiers, appelle la procédure *calculer CA* et ferme les deux fichiers avant de se terminer

produits.txt

```
61 AGRAFEUSE 37.60
73 BOITE_D'AGRAFES 2.30
178 BOITE_DE_TROMBONES 3.05
485 REGLE_GRADUEE 7.50
498 PAIRE_DE_CISEAUX  47.00
1100 COUPE_PAPIER 4.50
1125 RUBAN_ADHESIF 11.75
2172 TUBE_DE_COLLE 6.20
2183 BOITE_D'ELASTIQUES 11.25
4273 STYLO_BILLE 2.60
5332 STYLO_FEUTRE 5.10
6583 CRAYON_A_PAPIER 1.90
6622 TAILLE_CRAYON 3.80
9834 GOMME 4.50
```

ventes.txt

```
61 1
178 2
1100 5
1100 2
1100 3
2172 1
2172 1
2183 3
6583 1
6583 2
9834 10
```

TP7-TP8

LES EXERCICES À CODER

Exercices et algorithme à coder

1. **Ce que vous devez avant la séance de TP**

   (a) Lecture en mode **caractère**, en mode **ligne** et lecture **formatée** du fichier *produits.txt*
   Pour la lecture formatée, voir la définition du type *Produit* ci-dessous.

   (b) Implémenter l'algorithme du tri par sélection avec lecture des valeurs à partir d'un fichier texte.

2. **Ce que vous devez coder durant la séance de TP**

   (a) Calcul du chiffre d'affaire

   Les types *Produit* et *Vente* en C :

```
#define LONGUEUR 30

typedef struct { int code;
                 char nom[LONGUEUR];
                 float prix; } Produit;

typedef struct { int code;
                 int nbUnitesVendues; } Vente;
```

   Implémentez l'algorithme de calcul du chiffre d'affaire en lisant les fichiers *produits.txt* et *ventes.txt* avec la fonction de lecture formatée *fscanf*.

   (b) Ecrire un programme C qui ouvre deux fichiers :
   – le fichier *estBissextile.html* en lecture
   – le fichier *estBissextile.txt* en création

   Voici le fichier *estBissextile.html* :

estBissextile.html

```
<HTML>
<HEAD><TITLE>L'année bissextile</TITLE></HEAD>
<BODY>
<CENTER><H2><B> L'année bissextile </B></H2></CENTER>
<H3><B> Problème </B></H3>
<P ALIGN=JUSITFY>
  On veut déterminer si l'année <I>a</I> est bissextile ou non.
  Si <I>a</I> n'est pas divisible par 4 l'année n'est pas
  bissextile. Si <I>a</I> est divisible par 4, l'année est
  bissextile sauf si <I>a</I> est divisible par 100 et pas par
  400.<BR>
  Spécifier l'énoncé du problème. Définir la fonction
  <I>estBissextile</I> qui prend un paramètre <I>a</I> de type
  naturel est qui renvoie un booléen égal à VRAI si <I>a</I>
  est bissextile, à FAUX sinon.
```

estBissextile.html (suite)

```
</P>
<H3><B> Enoncé </B></H3>
<B> Donnée </B>  a : NaturelNonNul<BR>
<B> Résultat </B>  estBissextile(a) : Booléen
<H3><B> Algorithme </B></H3>
<PRE>
<B>fonction</B> estBissextile(a : Naturel) : Booléen
        <B>si</B> a <B>mod</B> 4 &lt;&gt; 0 <B>alors
                retourner</B> FAUX
        <B>sinon si</B> a <B>mod</B> 100 &lt;&gt; 0 <B>alors
                  retourner</B> VRAI
             <B>sinon si</B> a <B>mod</B> 400 &lt;&gt; 0 <B>alors
                       retourner</B> FAUX
                  <B>sinon
                       retourner</B> VRAI
                  <B>fsi
             fsi
        fsi
ffct</B> estBissextile
</PRE>
</BODY>
</HTML>
```

Il contient bien sûr des balises *HTML* qui spécifient, entre autre, la mise en page. Une balise est généralement associée à une contre-balise comme *<BODY>* et *</BODY>*. Certaines ne le sont pas comme la balise *
* qui permet un saut à la ligne. Les séquences *<* et *>* permettent d'afficher les caractères '<' et '>' sans interférer avec l'interprétation du fichier par le navigateur.

Votre programme devra contenir une fonction C *html2txt* typée **void** prenant deux paramètres de type **FILE*** correspondant aux deux fichiers ouverts dans la fonction *main*.
Elle doit lire le fichier *estBissextile.html* et écrire dans le fichier *estBissextile.txt* en retirant les balises *html*.
Les deux fichiers seront finalement fermés dans la fonction *main*.

3. **Pour aller plus loin**

   (a) Calcul du chiffre d'affaire

   Dupliquez votre source calculant le chiffre d'affaire à partir des fichiers texte et adaptez le pour lire les fichiers *produitsINTEL.bin* et *ventesINTEL.bin* avec la fonction de lecture *fread*

   (b) Conversion fichier binaire → fichier binaire codé hexadécimal

   Un fichier binaire codé hexadécimal est en fait un fichier texte. Chaque octet du fichier d'origine sera convertit en deux caractères de '0' à '9' ou de 'A' à 'F' correspondant aux valeurs des quartets de poids ford et de poids faible de celui-ci.

   (c) Vérification de l'en-tête d'une image *jpeg*

Les images *jpeg* (*Joint Photographic Experts Group*) sont obtenues par un algorithme de compression avec perte basé sur la transformation en cosinus discrète.
Elles sont strockées dans un fichier *JFIF* (*JPEG File Interchange Format*). Une application ouvrant un tel fichier doit vérifier son en-tête.
La valeur du premier octet doit être **`0xFF`** (héxadécimal). Si c'est le cas, il faut ensuite s'assurer que l'on trouve à partir du 7ème octet du fichier la chaîne de caractère "*JFIF*".

Ecrire le programme C *verifierJPEG* qui ouvre un fichier en lecture et qui affiche la réponse adéquate selon que ce fichier est au format *JFIF* ou non.

C10

LES POINTEURS EN C

Objectifs du C10

2

- Savoir manipuler les variables pointeur
  - les définir
  - accéder à l'objet pointé
- Maîtriser le passage d'un tableau en paramètre d'une fonction
- Savoir utiliser des constantes chaîne de caractères
- Savoir passer un pointeur sur fonction comme argument d'une autre fonction
- Connaître les principes de l'allocation dynamique
  - allocation/désallocation d'un tableau dynamique et d'une structure
  - accès aux champs d'une structure à partir d'un pointeur sur cette structure

3

Contenu du C10

1 Les pointeurs en C

4

5

1.1 Les variables pointeur

■ **Rappel sur l'adresse d'une variable**

♦ si *nb* est une variable, l'expression **&nb** dénote l'adresse de *nb*

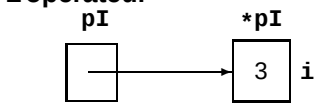
■ **Définition d'une variable pointeur sur un type**

```
#include <stdio.h>   // pour la constante symbolique NULL
int *pI = NULL;      // pI est un pointeur sur entier
char *pC = NULL;     // pC est un pointeur sur caractère
```

■ **Affectation d'une expression de type adresse à un pointeur**

```
int i = 3;
pI = &i;     // attention à la correspondance de type!!!
```

■ **L'opérateur `*`**



variable pointeur variable pointée

l'opérateur `*` permet d'accéder à la variable pointée

Il sert à **déréférencer** le pointeur

`i` ⟺ `* ( &i )`

■ Un pointeur qui ne recoit pas une adresse valide est **inutilisable**

– **&nb** est une « expression-adresse », c'est l'adresse du 1er octet de la zone mémoire utilisée pour le stockage de la valeur de **nb** (chaque octet en mémoire a une adresse propre).

– **Initialisation des variables pointeurs**
  – nous vous engageons, dans un premier temps, à initialiser systématiquement vos variables pointeur avec la constante symbolique **`NULL`**.
  – une variable pointeur non initialisée pointe sur n'importe quoi!!!
  – une variable pointeur initialisée à **`NULL`** ne pointe sur rien!!!, mais une tentative d'accès produira un arrêt immédiat de l'exécution et non un comportement erratique.

– L'opérateur de **déréférencement `*`** est appelé également opérateur d'**indirection**

1.2 Les opérations sur les variables pointeur

6

- **Le déréférencement** : **`*`**
- **L'affectation** : **`=`**
- **Les opérateurs de comparaison** : **`==`** et **`!=`**
- **L'arithmétique des pointeurs** :
  - Les opérateurs **`+`** et **`-`** avec des adresses et des expressions « entières »
- **La constante symbolique `NULL` correspond à une adresse non valide**
  - Un pointeur valant **`NULL`** ne pointe sur rien
- **Les fonctions d'allocation et de désallocation dynamiques de la mémoire offertes** par la librairie standard et **déclarées** dans le fichier *stdlib.h*
  - Allocation : **`malloc`**
  - Désallocation : **`free`**

– **Opérateurs de comparaison**
  – dans l'usage courant, le contenu d'une variable pointeur est souvent comparé avec la constante symbolique **`NULL`**

– **Arithmétique des pointeurs**
  – là, quelques précisions s'imposent :
    – on peut additionner le contenu d'une variable pointeur avec une expression entière, par exemple une constante entière ou le contenu d'une variable de type « entier »
    – mais que signifierait la somme du contenu de deux variables pointeur ?
    – par contre, on peut faire la différence du contenu de deux variables pointeur (usage peu fréquent)

– **Concernant les fonctions `malloc` et `free`**
  – nous verrons leur utilisation en détail un peu plus loin
  – c'est l'occasion de revenir sur un point qui génère souvent une confusion
    – les fonctions **`malloc`** et **`free`** ne sont pas **définies** dans le fichier *stdlib.h*
    – elles y sont uniquement **déclarées** ; on y trouve leur prototype
    – elles sont définies dans les fichiers de bibliothèque *libc.a* et *libc.so* ; vous verrez cela en détail dans le module AM12

1.3 Programme illustrant les pointeurs -

mesPremiersPointeurs.c

```
int main ( ) {
  int i = 5, j = 9;
  int *p1 = NULL, *p2 = NULL; // deux pointeurs sur entiers

  p1 = &i; // p1 pointe sur le contenu de la variable i
  p2 = p1; // p2 pointe également sur le contenu de la variable i

  *p1 = 9; // On a modifié le contenu de la variable i
  if ( p1 == p2 )
    printf ( "Nécessairement : *p1 == *p2 \n" );

  p2 = &j; // p2 pointe sur le contenu de la variable j
  if ( *p1 == *p2 )
    printf ( "N'implique pas : p1 == p2 \n" );
  return EXIT_SUCCESS;
}
```

7

– **Un petit exemple sans intérêt pratique**
  – si deux variables pointeur contiennent la même valeur (adresse), forcément il y a égalité des objets pointés
  – par contre, deux variables pointeur contenant des valeurs différentes peuvent tout à fait pointer sur des objets ayant la même valeur

1.4 Typage des pointeurs

8

■ **Type d'un pointeur**

♦ Un pointeur est typé par l'objet pointé

```
int *pI1, *pI2; // pI1 et pI2 sont de type pointeurs sur entier
char *pC;       // pC est de type pointeur sur caractère
```

■ **Affectation entre pointeurs**

♦ L'affectation **`p1 = p2;`** n'est **correcte** que si **`p1`** et **`p2`** sont de **même** type

♦ **`pI1 = pI2; // affectation correcte`**

♦ **`pI1 = pC;  // affectation eronnée`**

– **Arithmétique des pointeurs**
  – le compilateur gère l'arithmétique des pointeurs ; suivant le **type pointé**, cela se traduit de façon différente :
    – **`pI1++;`** : le contenu de **`pI1`** est incrémenté de quatre unités car le type pointé est un type **`int`** (sur quatre octets sous *Linux*)
    – **`pC++;`** : le contenu de **`pC`** est incrémenté d'une unité car le type pointé est un type **`char`** (sur un octet)

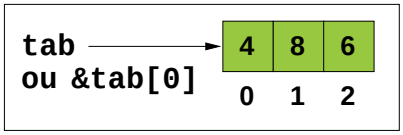
9

1.5 Pointeurs et tableaux

■ **Identificateur de tableau**

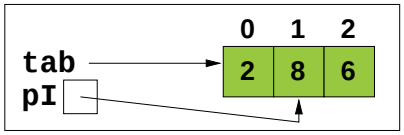
♦ L'identificateur d'un tableau est une « **constante** » dont la valeur est l'**adresse** du **premier** élément du tableau

```
#define N 3
int tab[N] = { 4, 8, 6};
```



■ **Opérateurs + et - sur les pointeurs**

```
int *pI;
pI=&tab[1]; // pI-1 pointe sur tab[0]
            // pI+1 pointe sur tab[2]
*(pI-1)=2;  // tab[0] est modifié
```



■ **Analogie tableau/pointeur**

`&tab[i]` ⟺ `tab+i` et donc `tab[i]` ⟺ `*(tab+i)`

– Le nom du tableau correspond à l'**adresse** de son **premier** élément

– Il y a réservation mémoire pour **les éléments** du tableau

– Quand on définit une variable pointeur, il n'y a réservation mémoire **que pour** la variable elle-même

1.6 Tableau en paramètre de fonction

10

■ **Prototypes de la fonction** *minTableau*

```
#define N 3
int minTableau ( int tab[N] );
```

ou

```
int minTableau ( int tab[] );
```

ou

```
int minTableau ( int *tab );
```

■ **Appel de la fonction** *minTableau*

```
int min, t[N] = { 3, 5, 7};
min = minTableau ( t );
```

■ **Passage du tableau** *t* **en paramètre**
à l'appel de *minTableau*, le paramètre formel **`tab`** est initialisé avec la valeur de l'expression **`&t[0]`**

■ **Un tableau est toujours passé par adresse**

– Dans les prototypes possibles de la fonction *minTableau*, on voit encore l'analogie entre tableaux et pointeurs
  – le tableau *t* a été défini dans l'**appelant**
  – dans le corps de la fonction *minTableau*, le paramètre formel *tab* n'est qu'une **référence** au premier élément du tableau
  – le nombre d'éléments du tableau n'est pas obligatoire ; c'est au **programmeur** à gérer cela

– Paramètre de type tableau à **deux dimensions**
  – un tableau de *N* lignes et *M* colonnes est stocké de manière **linéaire** en mémoire
  – pour accéder à l'élément de ligne *l* et de colonne *c*, le compilateur va calculer le déplacement $l \times M + c$ à partir de l'adresse du premier élément du tableau
  – il faut obligatoirement spécifier le **nombre de colonnes** du tableau pour le paramètre formel dans l'en-tête de la fonction, le nombre de lignes restant facultatif
  – **exemple** : la fonction *produitMatrices* qui réalise le produit de la matrice *N*×*M m1* par la matrice par *M*×*P m2* pour obtenir la matrice *N*×*P m3*
    `void produitMatrices (int m1[N][M], int m2[M][P], int m3[N][P]) { ... }`
    ou **`void produitMatrices (int m1[][M], int m2[][P], int m3[][P]) { ... }`**
    mais pas **`void produitMatrices (int m1[][], int m2[][], int m3[][]) { ... }`**

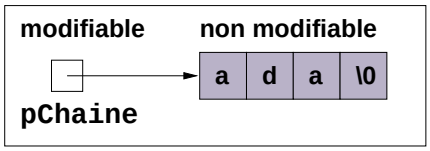
1.7 Pointeurs et chaînes de caractères

11

■ **Constantes chaînes de caractères**
- ♦ **`"pascal"`**, **`"ada"`** sont des **constantes** littérales
- ♦ une constante chaîne de caractères est implémentée comme une séquence de caractères terminée par le caractère conventionnel **`'\0'`**
- ♦ sa valeur est en fait l'**adresse constante** de son premier caractère
- ♦ son type est pointeur sur caractère

■ **Pointeur sur caractère**

```
char *pChaine = "ada";
```

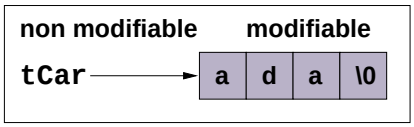


■ **Tableau de caractères**

```
char tCar[]="ada";
```

ou

```
char tCar[]={'a','d','a','\0'};
```



– **Définition d'une constante littérale**
  – on utilise bien sûr systématiquement la première syntaxe
  – **`char *pChaine = "ada";`**
  – si la variable pointeur **`*pChaine`** est modifiée, la référence à la chaîne **`"ada"`** est perdue

– **Constantes littérales référencées par le compilateur**
  – lorsque vous appelez les fonctions *scanf* et *printf*

```
printf ("Saisir m : ");
scanf ("%d", &m );
printf ("n + m = %d\n",n+m);
```

  – les constantes littérales **`"Saisir m : "`**, **`"%d"`** et **`"n + m = %d\n"`** sont référencées par le compilateur et stockées dans une zone de mémoire accessible uniquement en lecture

1.8 Pointeurs sur fonction -

12

- **Le langage C autorise la définition de pointeur sur fonction**
  - le **nom d'une fonction** est une constante de type pointeur sur fonction (de prototype correspondant) dont la valeur est l'**adresse** de sa première instruction (un peu comme le nom d'un tableau)
- **Passage d'une fonction en paramètre**

integration.c

```
double integrale(double(*f)(double),double a,double b,double n);
double carre ( double x) {
  return x * x;
}
int main ( ) {
  double aire;
  aire = integrale ( carre, -1., 2., 100);
  return EXIT_SUCCESS;
}
```

– Vous comprendrez mieux après avoir suivi le cours AM12 pourquoi l'identificateur d'une fonction correspond à l'adresse de la première instruction de cette fonction

– **Syntaxe du paramètre formel** : **`double(*f)(double)`**

  – **`double`** spécifie le type de la valeur de retour
  – **`(*f) ( )`** signifie que **`f`** est un pointeur **`*`** sur fonction **`( )`**
  Les parenthèses autour de **`*f`** sont obligatoires car sinon **`*f ( )`** est une fonction retournant un pointeur, du fait de la plus grande priorité de **`()`**
  – **`(double)`** spécifie que **`f`** prend **un** paramètre de type **`double`**

– **Syntaxe du paramètre effectif**

  – **`carre`** est une « expression-adresse »

1.9 Allocation dynamique de mémoire

13

■ **Principe de l'allocation dynamique**

- ♦ lorsque la taille mémoire nécessaire au stockage des données n'est pas connue à l'écriture du programme, le programmeur **doit demander** au système d'exploitation de lui **allouer** de la mémoire durant l'exécution
- ♦ cette mémoire est allouée dynamiquement dans une zone dédiée appelée **tas** (heap)

■ **Fonctions d'allocation/désallocation**

- ♦ **fournies** par la bibliothèque standard
- ♦ **déclarées** dans le fichier d'en-tête *stdlib.h*
- ♦ fonction d'allocation *malloc* : `void *malloc ( size_t size );`
- ♦ fonction de désallocation *free* : `void free (void *ptr);`

■ **Utilisations principales**

- ♦ Allocation/désallocation de tableaux **dynamiques** : dont la taille est fixée à l'**exécution** et non à la **compilation**
- ♦ Allocation/désallocation d'objets de type **structure**

– **Fonction** *malloc*

- *malloc* renvoie un pointeur générique `void *` qui pointe sur le début de la zone mémoire allouée
- ce pointeur peut être affecté à une variable pointeur sur n'importe quel type
- elle prend un seul paramètre `size` de type `size_t` (renommage du type `unsigned int`) : le nombre d'octets demandé
- il faut utiliser l'opérateur `sizeof`
- en cas d'échec, elle renvoie la constante symbolique `NULL`

– **Fonction** *free*

- *free* libère l'espace mémoire pointé par `ptr`

– **voir aussi les fonctions** *calloc* **et** *realloc*

1.10 Tableaux dynamiques -

14

■ **Tableau à une dimension**

tab1Ddyn.c

```
#include <stdlib.h> // malloc renvoie un pointeur générique (void *)
#include <stdio.h>  // pour printf et scanf

int main() {
  int nbElements, i, *t1D;  // t1D est un pointeur sur int
  printf("nombre d'éléments : ");
  scanf("%d",&nbElements);
  if((t1D=malloc(nbElements*sizeof(int)))==NULL) { // allocation mémoire
    printf("erreur allocation mémoire");           // test de la valeur de retour
    return EXIT_FAILURE;                           // NULL si échec
  }
  for (i=0;i<nbElements;i++) { // même utilisation qu'un tableau statique
    printf("t1D[%d] ? ",i);
    scanf("%d ",&t1D[i]);    // &t1D[i] ou t1D+i
  }
  free ( t1D ); // désallocation
  return EXIT_SUCCESS;
}
```

■ **Possibilité de gérer des tableaux à deux dimensions, voir plus . . .**

– **Il faut tester la valeur de retour de la fonction** *malloc*

  – conjointement avec l'appel comme sur le transparent
  – si l'écriture vous semble hermétique, vous pouvez tester cette valeur de retour après l'appel

```
t1D = malloc ( nbElements*sizeof(int) );
if ( t1D == NULL ) {
   printf("erreur allocation mémoire");
   return EXIT_FAILURE;
}
```

  – les programmes C anciens utilisaient souvent l'opérateur de conversion de type (*cast*)

```
t1D = (int *) malloc ( nbElements*sizeof(int) );
```

  – c'est inutile si vous avez inclus dans votre source le fichier *stdlib.h*

– Utilisation **identique** à un tableau statique à une dimension

– Pour un tableau dynamique à deux dimensions, c'est un peu plus compliqué → TP9-TP10

15

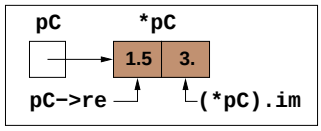
1.11 Gestion dynamique d'objets de type structure

■ **Gestion d'un objet de type structure : un complexe**

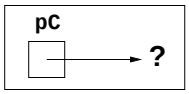
```
typedef struct {
    double re;
    double im; } Complexe;

Complexe *pC;// pointeur
             // sur Complexe
pC=malloc(sizeof(Complexe));
pC->re=1.5; // (*pC).re=1.5;
pC->im=3.;  // (*pC).im=3.;

free ( pC);
```

■ **Accès aux champs de l'objet pointé**

- ♦ déréférencement et utilisation de l'opérateur '**.**' **(*pC).re=1.5 // parenthésage**
- ♦ utilisation de l'opérateur '**->**' **pC->re = 1.5**



■ **Suppression** : **free (pC);**



déréférencement (***pC**) désormais illégal

– **Accès aux champs de l'objet pointé**

– utilisez de préférence l'opérateur **->**
– si vous déréférencez la variable pointeur, il faut parenthéser l'expression
– ***pC.re** provoquera une erreur car l'opérateur **.** est prioritaire par rapport à l'opérateur de déréférencement

– Remarque : les quatre opérateurs ayant la plus forte priorité en C sont **[]**, **.**, **->** et **()**

1.12 Les classes d'allocation mémoire en C -

16

- **Allocation statique** (déterminée par le compilateur)
  - variable **globale** : définie en dehors de toute fonction
  - variable **locale** à une fonction mais modifiée **`static`**
  - initialisation **par défaut** à 0 (variables non initialisées explicitement)
- **Allocation automatique** (réalisée pendant l'exécution)
  - variable **locale** à une fonction
  - **allocation** dans la **pile** de l'application
  - initialisation **indéterminée**
- **Allocation dynamique** avec *malloc* (exécutée à la demande)
  - variable **dynamique**
  - **allocation** à l'exécution dans le **tas**
  - initialisation **indéterminée**

– **Variables globales**

  – Nous nous en servirons très peu
  – Elles sont visibles dès leur définition et jusqu'à la fin du fichier source
  – Elles peuvent être modifiées inopinément à partir de n'importe quelle fonction (effet de bord)
  – Si l'on en abuse pas, elles sont parfois utiles

TP9-TP10

LES EXERCICES À CODER

AOÛT 2006

Exercices et algorithmes à coder

1. **Ce que vous devez coder avant la séance de TP**
   (a) Implémenter l'algorithme du tri par fusion [5.2] avec lecture des valeurs à partir d'un fichier binaire. Lors de l'exécution, vous redirigerez l'affichage du résultat vers un fichier texte
   (b) La valeur de retour et les arguments du main [exercices 1.1 et 1.2 ci-dessous]

2. **Ce que vous devez coder durant la séance de TP**
   (a) Les fonctions de manipulation de chaînes de caractère : palindrome [exercice 2 ci-dessous]
   (b) Le tableau dynamique à une dimension [exercice 3 ci-dessous]
   (c) Le tableau dynamique à deux dimensions [exercice 4 ci-dessous]

3. **Pour aller plus loin**
   (a) Le zéro d'une fonction par dichotomie en récursif avec pointeurs sur fonction [4.4]
   (b) Un tableau de pointeurs sur fonction [exercice 5 ci-dessous]
   (c) Application *MatriceEntiers* → indications sur le site *AP11*

La valeur de retour et les arguments du main

Bien que vous n'ayez pas utilisé cette possibilité jusqu'à présent, la fonction *main* renvoie une valeur de type **`int`**. Cette valeur peut être testée par le *shell*, elle indique comment s'est exécuté le processus.

Par convention la valeur nulle correspond à une exécution correcte, une valeur différente de zéro à un problème. Les constantes symboliques suivantes sont définies dans le fichier **`/usr/include/stdlib.h`** :

```
#define EXIT_FAILURE 1 /* Failing exit status. */
#define EXIT_SUCCESS 0 /* Successful exit status. */
```

On peut visualiser (et tester) cette valeur de retour. Elle correspond à une variable d'environnement prédéfinie du shell qu'on peut afficher sous *bash* avec la commande :

```
$ echo $?
```

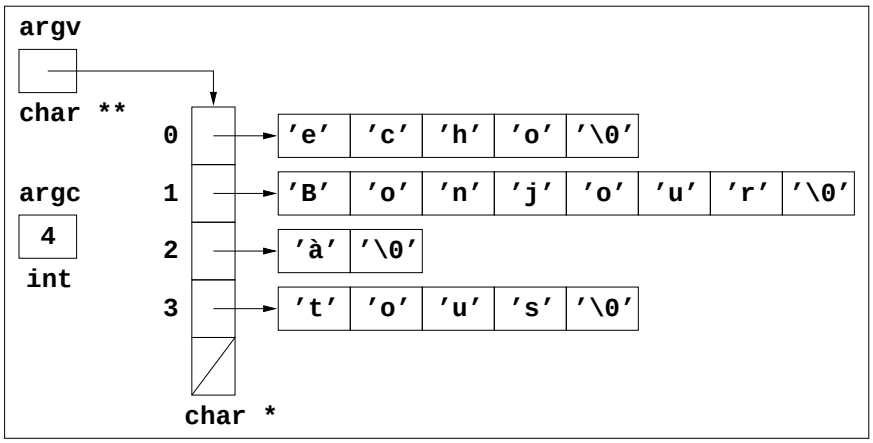
Le *main* peut être également défini avec des arguments, souvent deux, voire trois :

```
int main ( int argc, char *argv[]) { // ou int main ( int argc, char **argv)

       // définition de la fonction
}
```

Le premier paramètre *argc* correspond au nombre d'arguments de la ligne de commande (quand vous lancez l'application) y compris le nom de la commande elle-même. Le deuxième *argv* est un tableau de chaînes de caractères contenant les différents mots qui forment la ligne de commande, *argv[0]* correspondant au nom de la commande elle-même.

Par exemple, quand vous tapez la commande :

```
$ echo Bonjour à tous
```



Exercice 1.1

Ecrire un programme **`monEcho.c`** qui, après compilation

```
$ gcc -Wall monEcho.c -o monEcho
```

est lancé avec un nombre quelconque d'arguments et qui les affiche

```
$ monEcho Bonjour à tous
Bonjour à tous
$
```

Ne nommez pas votre exécutable **echo** car votre shell lancerait vraisemblablement l'exécution de la commande *echo* d'Unix.
Il en sera de même si vous nommer un exécutable *test* ou *eval* par exemple : voir la commande *which* qui vous indique quel fichier sera exécuté quand vous tapez une commande.

Exercice 1.2

Ecrire un programme **plus.c** qui, après compilation

```
$ gcc -Wall plus.c -o plus
```

est lancé avec deux arguments représentant deux valeurs entières

```
$ plus 45 38
```

et affiche :

```
45 + 38 = 83
$
```

Il faudra impérativement tester le nombre d'arguments de la ligne de commande, c'est à dire le contenu de *argc*. Si l'utilisateur tape par exemple :

```
$ plus 45
```

argv[2] vaudra **NULL**, ce qui provoquera une erreur à l'exécution.

Pour transformer les arguments de type pointeur sur caractère en valeur de type **int**, vous pouvez utiliser les fonctions *sscanf* (variante de *scanf*) ou *atoi* (ASCII to Integer). Consultez la documentation.

Fonctions de manipulation de chaînes de caractères : palindrome

L'objectif de cette exercice est avant tout de vous familiariser avec quelques fonctions de manipulation de chaînes de caractères. Ces fonctions de la bibliothèque standard sont déclarées dans le fichier *string.h*. Il suffit de taper la commande :

```
$ man string
```

pour en obtenir la liste. Tapez ensuite :

```
$ man strlen
```

pour disposer du manuel de la fonction *strlen* qui renvoie le nombre de caractères utiles de la chaîne passée en argument.

Un palindrome est une phrase qui se lit également de droite à gauche, mais en omettant la casse et l'accentuation des caractères ainsi que les espaces. Le plus connu est certainement :

Esope reste ici et se repose

Pour résoudre ce problème, il suffit de recopier la chaîne de caractères dans un tableau de caractères correctement dimensionné et d'appliquer l'algorithme d'inversion des éléments d'un tableau, version itérative (3.1.3) ou récursive (4.2.3).
Il faut bien sûr adapter cet algorithme pour pouvoir l'appliquer à un tableau de caractères. Finalement, en comparant le tableau inversé à la chaîne initiale, on peut déterminer si celle-ci est un palindrome ou non.

Exercice 2

Vous devez écrire un fichier source **`estUnPalindrome.c`** qui prend un seul argument sur la ligne de commande : la chaine de caractères à tester. Le programme résultant doit répondre simplement oui ou non suivant que la chaîne est un palindrome ou non.

Exemple d'exécution :

```
$ estUnPalindrome esoperesteicietserepose
oui
$
```

Les fonctions de manipulation de chaîne à utiliser sont *strlen*, *strncpy* et *strncmp* : voir le manuel en ligne.

Tableau dynamique à une dimension

Exercice 3

Ecrire un programme qui permettra de gérer un tableau dynamique d'entiers à une dimension. La taille de ce tableau ne sera pas fixée à la compilation, mais à l'exécution.

L'utilisateur devra saisir le nombre d'éléments du tableau. Un appel à la fonction *malloc* permettra de faire allouer par le système l'espace mémoire nécessaire. La valeur de retour sera affectée à une variable de type **`int *`**. Pensez à tester cette valeur de retour (voir le manuel).

On vous demande également d'affecter à chaque élément du tableau une valeur aléatoire comprise entre 0 et 99. Pour cela il faut utiliser les fonctions *random* et *srandom.*

La fonction *random* renvoie une valeur de type **`long`** comprise entre 0 et **`RAND_MAX`**. La constante symbolique **`RAND_MAX`** est définie dans le fichier *stdlib.h* :

```
#define RAND_MAX 2147483647
```

Elle correspond à la valeur maximum des types **`int`** ou **`long`** (sous *Linux*) tout comme **`INT_MAX`**

La fonction *srandom* permet d'initialiser le germe du générateur. Si elle n'est pas utilisée, la séquence générée sera la même à chaque exécution.
Elle prend un paramètre de type **`unsigned`** (i.e. **`unsigned int`**), c'est le germe.
On peut faire saisir ce germe par l'utilisateur. On peut aussi se servir de la primitive *time* qui renvoie le

nombre de secondes écoulées depuis le premier janvier 1970. On peut de préférence utiliser la primitive *getpid* qui retourne le *pid* du processus. Cette valeur est forcément différente pour deux exécutions consécutives.

Consultez la documentation.

L'exécution se terminera en affichant les valeurs du tableau.

Pensez à décomposer votre programme (définir plusieurs fonctions).

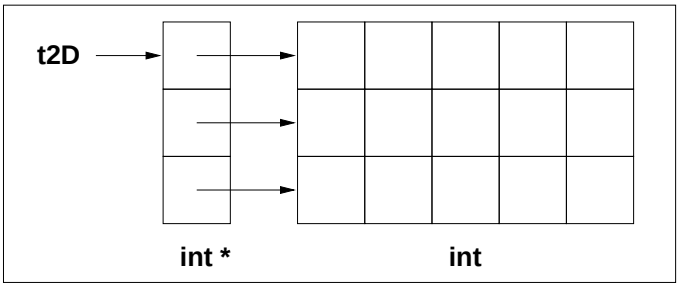
Tableau dynamique à deux dimensions

Exercice 4

Pour gérer un tableau dynamique d'entiers de nbLignes et de nbColonnes, il faut disposer de deux tableaux alloués dynamiquement :

- un tableau de nbLignes éléments de type pointeurs sur **`int`**
- un tableau de nbLignes x nbColonnes éléments de type **`int`**

Chaque élément du tableau de **`int *`** pointe sur le premier élément de la ligne correspondante dans le tableau de **`int`**. Voici un petit exemple avec 3 lignes et 5 colonnes :



L'allocation du tableau de **`int`** peut se faire avec un seul appel à la fonction *malloc*. Dans un schéma itératif, on affecte alors aux éléments du tableau de **`int *`** l'expression adéquate. On peut aussi faire un appel à la fonction *malloc* pour chaque ligne du tableau de **`int`**.
Dans les deux cas ceci autorise des lignes de longueur variable, ce qui peut être utile pour stocker « au plus juste » une matrice carrée symétrique (éventuellement sans la diagonale).

Concernant la désallocation, un appel à la fonction *free* avec **`t2D[0]`** comme paramètre permet de désallouer le tableau de **`int`**. Un dernier appel à la fonction *free* avec **`t2D`** comme paramètre permet de désallouer le tableau de **`int *`**.

Reprenez le scénario proposé pour le tableau dynamique à une dimension : un tirage aléatoire des valeurs et un simple affichage.

Un tableau de pointeurs sur fonction

Exercice 5

Vous devez préalablement définir quatre fonctions C prenant un paramètre *x* de type **`double`** et renvoyant une valeur de type **`double`** :

- $f1$ renvoie $2 \times x$
- $f2$ renvoie x^2 (utiliser une multiplication plutôt que la fonction *pow*)
- $f3$ renvoie $f1(x) + f2(x) + 1$
- $f4$ renvoie $f2(f2(x + 1))$

Vous définirez ensuite dans la fonction *main* un tableau de quatre pointeurs sur fonction initialisés avec *f1*, *f2*, *f3* et *f4*.
Vous afficherez, dans un schéma itératif, le résultat de l'appel de ces fonctions avec différentes valeurs de paramètre.

APPLICATION IMAGE

LISSAGE D'UNE IMAGE EN NIVEAUX DE GRIS

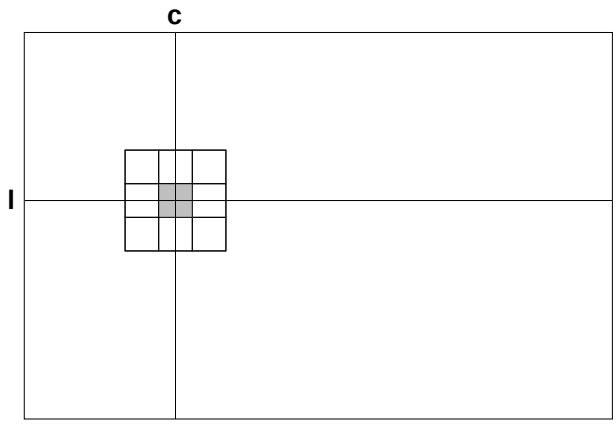
AOÛT 2006

Présentation

Cet exercice de synthèse, à réaliser en hors-présentiel, a pour but de vous faire réaliser rapidement une application en vous fournissant un ensemble de modules.
On veut, cette fois-ci, vous faire manipuler autre chose que des nombres ou des chaînes de caractères. Nous vous proposons donc de réaliser une opération de lissage sur une image en niveaux de gris au format *PGM* (*Portable Graymap*).

Enoncé du problème

Une image est représentée par un tableau de pixels à deux dimensions, de *L* lignes et de *C* colonnes. Chaque élément du tableau représente l'intensité du pixel correspondant. Typiquement, les images en niveaux de gris sont codées sur un octet, la plage de valeur est comprise entre 0 et 255.
Le lissage est un des traitements d'image de bas niveau parmi les plus simples. Il permet d'éliminer le bruit sur une image. Il consiste à réévaluer chaque valeur de pixel en déplaçant une fenêtre carrée de côté *côté*, où *côté* est impair et supérieur ou égal à 3, centrée sur le pixel à évaluer. La nouvelle valeur du pixel sera égale à la moyenne des valeurs des pixels de la fenêtre (pixel courant inclus). Si par exemple *côté* est égal à 3, la moyenne sera calculée à partir de 9 valeurs.



L'exemple ci-dessus montre la position d'une fenêtre 3x3 pour le pixel de ligne *l* et de colonne *c*.
Bien entendu, la valeur d'un pixel ne peut être modifiée que si la fenêtre centrée sur celui-ci est entièrement définie. Il y aura autour de l'image, des bandes de pixels non traités. Si *côté* est égal à 5, ces bandes auront deux pixels de large.

Spécifications du problème

Vous trouverez ci-dessous les définitions des types *Pixel* et *Image* et de la procédure *lisserImage*, qui prend en donnée un paramètre *coté* de type Naturel, un paramètre *imageEntrée* de type *Image* et en résultat un paramètre *imageSortie* de type *Image*.

Définition des types

type Pixel = Naturel

type Image = **tableau**[1..L][1..C] **de** Pixel

Application Image

Enoncé

Données côté : Naturel, imageEntrée : Image

Résultat imageSortie : Image

Algorithme

procédure lisserImage (**donnée** côté : Naturel, imageEntrée : Image, **résultat** imageSortie : Image)

delta = côté **div** 2 : Naturel

nbPixels = côté × côté : Naturel

// traitements des pixels concernés par l'opération de lissage

pour ligne = 1 + delta : Naturel **à** L - delta

pour colonne = 1 + delta : Naturel **à** C - delta

imageSortie[ligne][colonne] ← sommePixels (imageEntrée, ligne, colonne, delta) **div** nbPixels

fpour colonne

fpour ligne

fproc lisserImage

// définition de la fonction *sommePixels*

fonction sommePixels(image : Image, y, x, delta : Naturel) : Naturel

somme=0 : Naturel

pour ligne = y - delta : Naturel **à** y + delta

pour colonne = x - delta : Naturel **à** x + delta

somme ← somme + image[ligne][colonne]

fpour colonne

fpour ligne

retourner somme

ffct sommePixels

Application Image

// On devrait compléter l'algorithme pour recopier les « bords » de l'image en entrée sur l'image en sortie.

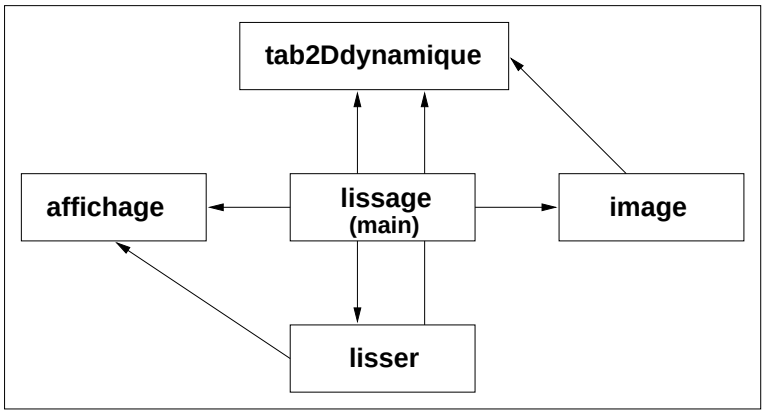
// Ce n'est pas nécessaire car pour avoir un affichage de l'image lissée en « temps réel »,

// l'image en entrée sera intégralement recopiée sur l'image en sortie.

Mise en œuvre

Pour que cela soit réalisable dans le temps imparti, trois modules vous sont fournis complets. Vous aurez uniquement à compléter le quatrième.
Vous disposez également du programme principal et de deux images pour pouvoir tester l'application.

Voici l'architecture de l'application : c'est, en plus modeste, ce que vous aurez à construire pour votre projet informatique *PI12*.



1. **Le module image** (entrée-sortie pour des images)

   – **fichier `image.h`**
     – définition des types *Pixel* et *Image*

```
typedef unsigned char Pixel;
typedef struct { int nbLignes;
                 int nbColonnes;
                 Pixel **t2D;   } Image;
```

     – déclaration des fonctions *lireImage* et *ecrireImage* permettant respectivement le chargement et le stockage une image au format *PGM* (*Portable GrayMap*) → **`$ man pgm`**

```
Image lireImage ( char *nomFichier );
void ecrireImage ( Image image, char *nomFichier );
```

   – **fichier `image.c`**
     – définition des fonctions *lireImage* et *ecrireImage*
     – les fonctions *lireImage* et *ecrireImage* sont appelées à partir du programme principale **`lissage.c`**

2. **Le module tab2Ddynamique** (pour travailler avec des images de taille quelconque)

   – **fichier `tab2Ddynamique.h`**
     – déclaration des fonctions *allouer* et *desallouer*

```
Pixel **allouer ( int nbLignes, int nbColonnes );
Pixel **desallouer ( Pixel **t2D );
```

- **fichier `tab2Ddynamique.c`**
  - définition des fonctions *allouer* et *desallouer*
  - la fonction *allouer* est appelée à partir du module *image*
  - la fonction *desallouer* est appelée à partir du fichier **`lissage.c`**

3. **Le module affichage** (pour afficher l'image dans une fenêtre X-Window)

   - **fichier `affichage.h`**
     - définition du type *Ecran* : un peu technique, c'est l'utilisation de la *Xlib*, mais vous pouvez y jeter un œil
     - déclaration des fonctions *initialiserAffichage*, *libererAffichage*, *afficherImage*, *changerLigne* et *changerBloc*

     ```
     Ecran initialiserAffichage(int largeur,int hauteur,char *nomFichier);
     void libererAffichage(Ecran ecran);
     void afficherImage(Ecran ecran,Image *pIm);
     void changerLigne(Ecran ecran,Image *pIm,int y);
     void changerBloc(Ecran ecran,Image *pIm,int y,int h);
     ```
   - **fichier `affichage.h`**
     - définition des fonctions *initialiserAffichage*, *libererAffichage*, *afficherImage*, *changerLigne* et *changerBloc*
     - les fonctions *initialiserAffichage*, *libererAffichage* et *afficherImage* sont appelées à partir du fichier **`lissage.c`**
     - les fonctions *changerLigne* et *changerBloc* sont appelées à partir du module *lisser*

4. **Le module lisser** (à compléter)

   - **fichier `lisser.h`**
     - déclaration de la fonction *lisserImage*

     ```
     void lisserImage(int cote,Image imEntree,Image *pimSortie,Ecran ecran);
     ```
   - **fichier `lisser.c`**
     - définition partielle de la fonction *lisserImage*
     - la fonction *lisserImage* est appelée à partir du fichier **`lissage.c`**

5. **Le programme principal** : **`lissage.c`**

   - il contient la définition de la fonction *main* qui récupère les arguments de la ligne de commande et qui appelle les fonctions :
     - *lireImage*
     - *initialiserAffichage*
     - *afficherImage*
     - *lisserImage*
     - *ecrireImage*
     - *libererAffichage*
     - *desallouer*

6. **Ce que vous avez à faire**

   - éditer le fichier **`lisser.c`** : **`$ xemacs lisser.c&`**
   - définir la fonction *sommePixels*

- compléter la fonction *lisserImage* en appelant, soit la fonction *changerLigne* dès qu'une ligne de l'image est traitée, soit la fonction *changerBloc* dès que **HAUTEUR_BLOC** lignes sont traitées. **HAUTEUR_BLOC** est une constante symbolique définie dans le fichier **lisser.c**
- Ecrire le fichier *makefile* correspondant à cette application
- produire l'exécutable : **$ make**
- lancez l'exécution : **$ lissage**

```
usage : lissage nomFichierSource nomFichierDest cote
$ lissage lena.pgm lenaL5.pgm 5
```

C11

LES STRUCTURES LINÉAIRES

2

Objectifs du C11

- Connaître les principes
  - de la structure de pile
  - de la structure de file
  - de la structure de liste linéaire
- Connaître le principe et les détails d'implémentation
  - de la structure de liste linéaire triée

3

Contenu du C11

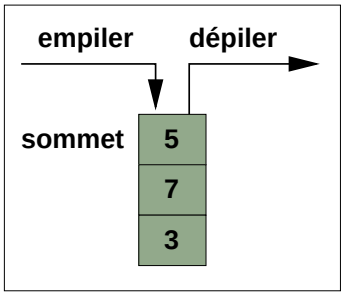
1 La structure de pile

4

1.1 La pile

5

- **Ensemble d'éléments de même type en nombre variable** ≥ 0
- **Gestion de type** *LIFO* **:**
  Last In First Out
- **Accès unique** : le *sommet* de la pile
- **Exemple** : une pile d'entiers



- **Les opérations définies sur une pile d'éléments de type** *TypeValeur*

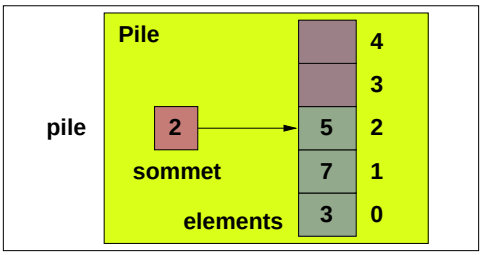
nouvellePile	:		→	Pile
estPileVide	:	Pile	→	Booléen
empiler	:	Pile × TypeValeur	→	Pile
dépiler	:	Pile − { ∅ }	→	Pile
valeurSommet	:	Pile − { ∅ }	→	TypeValeur

– **Opérations** *nouvellePile*
  – renvoie une pile vide

– **Opérations** *dépiler* **et** *valeurSommet*
  – elles sont, dans la pratique, souvent utilisée conjointement
  – on récupère la valeur de l'élément en sommet de pile, puis on dépile cet élément
  – c'est ce qui se passe avec les **instructions processeur** de gestion de la pile de l'application (ou de la pile système) → module AM12

6

1.2 Représentation/implémentation par tableau




```
#define NB_MAX_ELEMENTS 5

typedef struct {
   int elements[NB_MAX_ELEMENTS];
   int sommet; } Pile;

Pile pile = nouvellePile ();
```

style « fonctionnel »

```
Pile empiler( Pile p, int v ) {

  (p.sommet)++;
  (p.elements)[p.sommet] = v;
  return p;
}
// appel : pile=empiler( pile, 1 );
```

style « procédural »

```
void empiler( Pile *pP, int v ) {

  (pP->sommet)++;
  (pP->elements)[pP->sommet] = v;
}

// appel : empiler ( &pile, 1 );
```

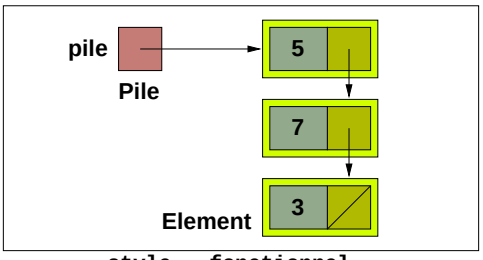
- **Utilisation d'un tableau dynamique**
  - redéfinition du type *pile*

```
#define GRANULARITE 50
typedef struct { int *elements;
                 int sommet;
                 int nbMaxElements;
                 int nbElements     } Pile;
```

  - à l'initialisation de la pile
    - **`pile.elements = malloc ( GRANULARITE*sizeof(int) );`**
    - **`pile.nbMaxElements = GRANULARITE;`**
    - **`pile.nbElements = 0;`**
    - **`pile.sommet = -1;`**
  - dès que **`pile.nbElements`** devient égal à **`pile.nbMaxElements`**
    - **`pile.elements = realloc (pile.elements,`**
      `( pile.nbMaxElements + GRANULARITE ) * sizeof(int) );`
    - **`pile.nbMaxElements += GRANULARITE;`**
  - consultez le manuel pour l'utilisation de la fonction *realloc*

7

1.3 Représentation/implémentation par pointeurs




```
typedef struct element {
   int valeur;
   struct element *lien; } Element;

typedef Element *Pile;

Pile pile = nouvellePile ();
```

style « fonctionnel »

```
Pile empiler( Pile p, int v ) {
  Pile pTmp;
  pTmp = malloc( sizeof(Element) );
  pTmp->valeur = v;
  pTmp->lien = p;
  return pTmp;
}
// appel : pile=empiler( pile, 1 );
```

style « procédural »

```
void empiler( Pile *pP, int v ) {
  Pile pTmp;
  pTmp = malloc( sizeof(Element) );
  pTmp->valeur = v;
  pTmp->lien = *pP;
  *pP = pTmp;
}
// appel : empiler ( &pile, 1 );
```

- **Nommage de la structure**
  - contrairement aux types structure utilisés précédemment (type *Produit* par exemple), ici le nommage de la structure est obligatoire
  - on définit le champ lien de type **`struct element *`**
  - autre possiblité :

```
typedef struct element *Pile;

typedef struct element { int valeur;
                         Pile  lien; } Element;
```

- **Détail sur les implémentations de la fonction** *empiler*
  - la variable locale **`pTmp`** qu'on aurait pu nommée **`ptrNouvelElement`** peut également être définie de type **`Element *`**
  - veillez systématiquement à la cohérence entre la déclaration/définition d'une fonction C et son utilisation (appel)
  - si vous définissez la fonction *empiler* en style « fonctionnel » et que vous l'appelez de la façon suivante :

    `empiler ( pile, 1 );`

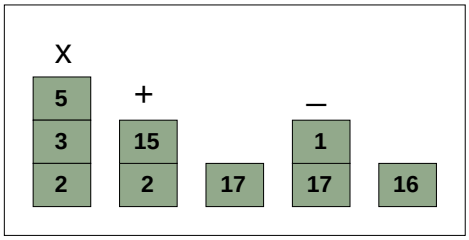
    Il n'y aura pas de problème à la compilation ; par contre, il y en aura un à l'exécution. Et pas forcément facile à détecter.

1.4 Applications de la structure de pile -

8

■ **Evaluation d'expressions arithmétiques**

♦ **expression postfixée ou**
notation polonaise inverse :
► $2\ 3\ 5\ \times\ +\ 1\ -$
► les opérandes sont empilés
► les opérateurs
★ dépilent deux éléments
★ effectuent l'opération
★ empilent le résultat



■ **Gestion par le compilateur des appels de fonctions**
♦ les paramètres, l'adresse de retour et les variables locales sont stockées dans la **pile** de l'application

■ **Parcours « en profondeur » de structures d'arbre, de graphe**

■ **Amélioration de la complexité des algorithmes**
♦ l'algorithme de Graham pour déterminer l'enveloppe convexe d'un ensemble de points

– **Evaluation d'expressions arithmétiques**
  – pour une expression en notation infixée et parenthésée, c'est un peu plus compliqué. Cela nécessite :
    – une pile d'opérandes
    – une pile d'opérateurs

– **Parcours « en profondeur » de structures d'arbre, de graphe**
  – généralement une implémentation **récursive** est beaucoup plus concise
    – la récursion utilise la pile de l'application et donc, « simule » l'usage explicite d'une pile

– **Amélioration de la complexité des algorithmes**
  – c'est une utilisation importante des structures de données
  – vous vous en rendrez compte :
    – dans les modules d'optimisation de 2ème année pour les problèmes d'*arbre de recouvrement minimum* et de *plus courts chemins*

2 La structure de file

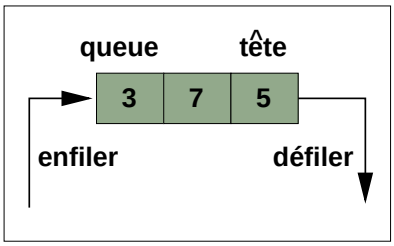
9

2.1 La file

10

- **Ensemble d'éléments de même type en nombre variable** ≥ 0
- **Gestion de type** *FIFO* **: First In First Out**
- **Deux accès**
  - ♦ insertion → *la queue de la file*
  - ♦ suppression → *la tête de la file*

- **Exemple** : une file d'entiers



- **Les opérations définies sur une file d'éléments de type** *TypeValeur*

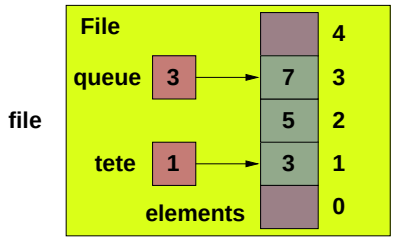
nouvelleFile	:		→	File
estFileVide	:	File	→	Booléen
enfiler	:	File × TypeValeur	→	File
défiler	:	File − { ∅ }	→	File
valeurTête	:	File − { ∅ }	→	TypeValeur

– **Opérations** *nouvelleFile*
  – renvoie une file vide

– **Opérations** *défiler* **et** *valeurTête*
  – de la même façon que pour la pile, elles sont, dans la pratique, souvent utilisées conjointement
  – on récupère la valeur de l'élément en tête de file, puis on défile cet élément

11

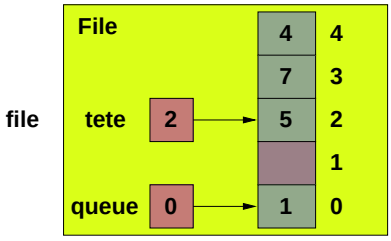
2.2 Représentation d'une file par tableau




```
#define NB_MAX_ELEMENTS 5

typedef struct {
   int elements[NB_MAX_ELEMENTS];
   int tete;
   int queue; } File;

File file = nouvelleFile ();
```

■ **Gestion circulaire**

- ♦ optimiser l'utilisation du tableau
- ♦ incrémentation des champs *tête* et *queue* **modulo** la taille du tableau
- ♦ état de la file après avoir :
  - ► enfilé la valeur 4
  - ► défilé la valeur 3
  - ► enfilé la valeur 1



– **Vous l'aurez compris**
  – la représentation par tableau avec une gestion basique n'offre guère d'intérêt.

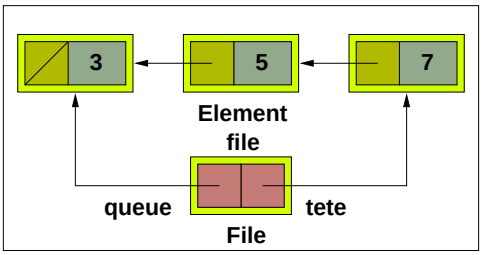
– **Gestion circulaire du tableau contenant la file**
  – si la *queue* de la file est égal à l'indice **`NB_MAX_ELEMENTS-1`**, un nouvel élément est inséré à la position d'indice 0 du tableau, si cet emplacement est libre (non occupé par l'élément en *tête* de file).
  – si la *tête* de la file est égal à l'indice **`NB_MAX_ELEMENTS-1`**, la suppression de cet élément conduit à un nouvel élément en tête de file à l'indice 0 du tableau (si la file n'est pas devenue vide).
  – les opérations *enfiler* et *défiler* correspondent à une incrémentation **modulo `NB_MAX_ELEMENTS`** des champs **`queue`** et **`tete`**.
  – les valeurs respectives des champs **`tete`** et **`queue`** ne suffisent plus à caractériser l'état de la file.
    – si **`tete`** est égal à **`queue`**, la file contient un élément.
    – mais si **`tete`** est égal à **`queue`**+1 (au modulo près), la file peut être soit vide soit pleine.
  – d'où la nécessité d'introduire un nouveau champ **`nbElements`** au type *File* et une opération supplémentaire *estFilePleine* associée à ce type.
  – **définition du type** *FileCirculaire*

```
typedef struct { int elements[NB_MAX_ELEMENTS];
                 int tete;
                 int queue;
                 int nbElements; } FileCirculaire;
```

  – le champ **`nbElements`** sera initialisé à zéro dans la fonction **`nouvelleFile`**, incrémenté dans la fonction **`enfiler`** et décrémenté dans la fonction **`defiler`**.
  – l'utilisation de l'opération *enfiler* sera conditionnée par la valeur de retour de l'opération *estFilePleine*.

2.3 Représentation/implémentation par pointeurs

12




```
typedef struct element {
   int valeur;
   struct element *lien; } Element;
typedef struct {
   Element *tete;
   Element *queue; } File;
File file = nouvelleFile ();
```

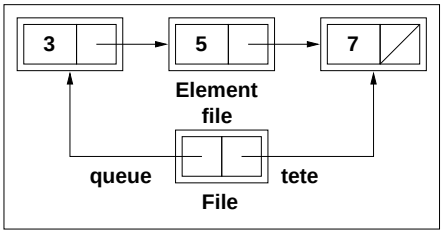
la fonction enfiler en style « procédural »

```
void enfiler ( File *pF, int v ) {
  Element *ptrNouvelElement; // référence le nouvel élément
  ptrNouvelElement = malloc( sizeof(Element) );
  ptrNouvelElement->valeur = v;
  ptrNouvelElement->lien = NULL;
  if ( estFileVide(*pF) ) pF->tete = pF->queue = ptrNouvelElement;
  else { (pF->queue)->lien = ptrNouvelElement;
         pF->queue = ptrNouvelElement; }
}
```

- **Type** *File*
  - comme la file est caractérisée par deux « points » d'accès, le type *File* est une structure à deux champs de type **`Element *`**.

- **Conseils pour l'implémentation des opérations** (valables pour pour les structures chaînées en général)
  - faire des schémas pour bien décomposer les différentes actions à exécuter.
  - valider la définition de la fonction C correspondant à l'opération à implémenter en envisageant tous les cas particuliers.
    Pour l'opération *enfiler*, il y a seulement deux cas :
    - la file contient au moins un élément
    - la file est vide

- **Sens du chaînage**
  - il semble, à première vue, inversé par rapport à la «circulation» des éléments dans la file.
    → la métaphore de la « **queue au supermarché** »
  - quelle opération serait impossible à implémenter sans parcours si nous inversions le sens du chaînage comme ci-dessous ?



2.4 Applications de la structure de file -

13

- **Programmation système**
  - **service d'impression**
    - les requêtes sont placées dans une file
    - le service retire une requête dès que l'impression est exécutée
  - **allocation du processeur aux processus en attente d'exécution**
    - notion de files d'attente avec **priorité**
  - **gestion des entrées-sorties**
    - tampon **circulaire** pour stocker les caractères en provenance du clavier
- **Parcours « en largeur » de structures d'arbre, de graphe**

– **allocation du processeur aux processus**
  – l'*ordonnanceur* gère plusieurs files de processus de priorité croissante → module AM12.

– **gestion des entrées-sorties**
  – les caractères en provenance du clavier sont stockés provisoirement dans une file circulaire, le tableau correspondant à une zone mémoire dédiée et gérée par le système ; typiquement :
    – 32 octets sous *DOS*
    – 4096 sous *Linux*

– **Parcours « en largeur » de structures d'arbre, de graphe**
  – contrairement au parcours « en profondeur » où l'implémentation récursive permet d'éviter de recourir à une structure de pile, le parcours « en largeur » d'un arbre ou d'un graphe nécessite l'utilisation d'une structure de file.
  – voir le parcours en largeur d'un arbre dans le C12.

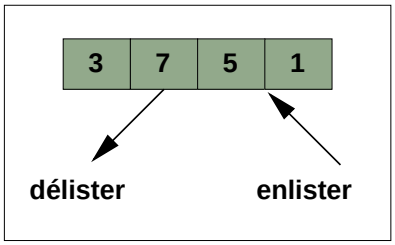
3 La structure de liste linéaire

14

3.1 La liste

15

- **Ensemble d'éléments de même type en nombre variable** ≥ 0
- **Ordre quelconque**
  - ♦ accès à n'importe quel élément
  - ♦ positionnement par parcours (partiel)
  - ♦ insertion/suppression à n'importe quelle position
- **Exemple** : une liste d'entiers



- **Les opérations définies sur une liste d'éléments de type** *TypeValeur*

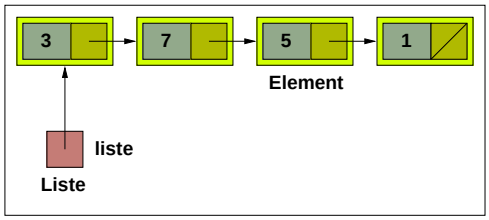
nouvelleListe	:		→	Liste
estListeVide	:	Liste	→	Booléen
nbElémentsListe	:	Liste	→	Naturel
enlister	:	Liste × TypeValeur	→	Liste
délister	:	Liste − { ∅ }	→	Liste
valeurElémentListe	:	Liste − { ∅ }	→	TypeValeur
suiteListe	:	Liste	→	Liste

– **A priori pour la structure de liste**
  – contrairement aux structures de pile et de file pour lesquelles la représentation par tableau est envisageable, l'insertion et la suppression d'un élément à n'importe quelle position dans une liste fait que sa représentation par tableau sera en général tout à fait inefficace.
  – pensez à l'algorithme de tri par insertion où il faut décaler une portion du tableau pour « faire une place » à l'élément à insérer.
  – néanmoins dans le cas où la structure de données n'évolue pas dans le temps, le stockage par contiguïté (tableau) reste la meilleure solution → voir la représentation d'un graphe.

16

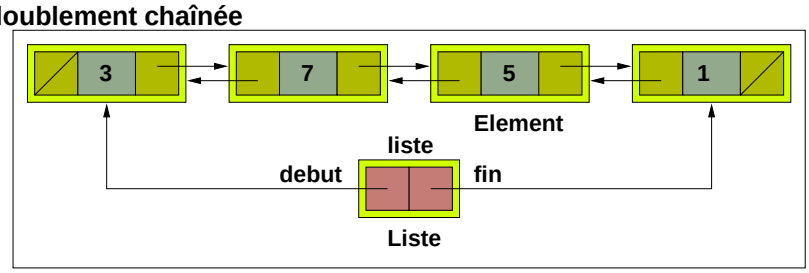
3.2 Représentation d'une liste par pointeurs

■ **Liste simplement chaînée**




```
typedef struct element {
   int valeur;
   struct element *lien; } Element;

typedef Element *Liste;

Liste liste = nouvelleListe ();
```

■ **Liste doublement chaînée**



- **Liste simplement chaînée**
  - parcours unidirectionnel
  - remarquez que la définition du type *Liste* proposée est identique à celle du type *Pile*.
  - on peut enrichir le type en introduisant la notion de *position courante* ou (*curseur*) permettant d'effectuer des enlistages ou délistages à n'importe quelle position.
- **Liste doublement chaînée**
  - parcours bidirectionnel
  - définition du type *Liste* (doublement chaînée)

```
typedef struct element { int valeur;
                         struct element   *successeur;
                         struct element *predecesseur; } Element;
typedef struct { Element *debut;
                 Element   *fin; } Liste;
```

- **Structures génériques**
  - que ce soit pour la file, la pile ou la liste, le champ **`valeur`** du type **`Element`** a toujours été de type **`int`** dans les exemples.
  - on aurait pu choisir **`double`** ou tableau de caractères, peu importe.
  - il n'empêche que si vous disposez d'une structure contenant un type donné, il faudra l'adapter pour l'utiliser avec un autre type.
  - on peut définir une structure de données générique en typant le champ *valeur* **`void *`** (pointeur générique) :

```
typedef struct element { void *valeur;
                         struct element *lien; } Element;
```

  - les prototypes des fonctions d'insertion auront alors un paramètre **`v`** de type **`void *`**.
  - attention à la durée de vie des objets pointés.

3.3 Algorithmes et fonctions C de parcours

17

■ Version itérative

procédure parcourirListe(**donnée** liste : Liste)

    tant que non estListeVide (liste) **faire**
        afficher (valeurElémentListe (liste))
        liste ← suiteListe (liste)
    ftq
fproc parcourirListe

```
void parcourirListe(Liste liste) {
 while( !estListeVide (liste) ) {
  int val = valeurElementListe (liste);
  printf ( "%d\n", val );
  liste = suiteListe ( liste );
 }
}
```

■ Version récursive

procédure parcourirListe(**donnée** liste : Liste)

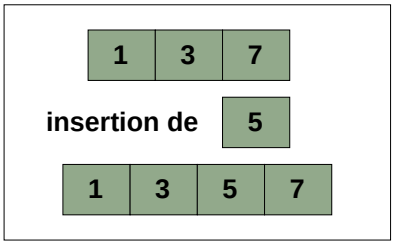
    si non estListeVide (liste) **alors**
        afficher (valeurElémentListe (liste))
        parcourirListe (suiteListe (liste))
    fsi
fproc parcourirListe

```
void parcourirListe(Liste liste) {
 if( !estListeVide (liste) ) {
  int val = valeurElementListe (liste);
  printf ( "%d\n", val );
  parcourirListe( suiteListe (liste) );
 }
}
```

3.4 La liste triée

18

- **Liste avec des éléments d'un type** *TypeValeur* **totalement ordonné**
  - ♦ rangement des éléments
    - ► suivant l'ordre des valeurs
    - ► on considère qu'il n'y a pas de doublon
- **Exemple** : une liste d'entiers



- **Les opérations propres à une liste triée d'éléments de type** *TypeValeur*

enlister	:	Liste × TypeValeur	→	Liste
délister	:	Liste × TypeValeur	→	Liste
estElémentListe	:	Liste × TypeValeur	→	Booléen
parcourirListe	:	Liste × Fonction	→	

par exemple, une fonction **traiterValeur** qui effectue une action sur la valeur de l'élément

traiterValeur	:	TypeValeur	→	

– **Opération enlister**
  – elle est bien sûr différente de celle de la liste ordinaire dans le sens où il faut rechercher la position d'insertion.

– **Opération délister**
  – pour la liste triée, on passe un argument supplémentaire : la valeur à retirer de la liste.

– **Opération estElémentListe**
  – renvoie VRAI si la valeur en argument est présente dans la liste, FAUX sinon.

– **Opération parcourirListe**
  – on peut envisager, pour la liste triée, de parcourir la liste en effectuant un traitement (par exemple un affichage) des valeurs.

3.5 Fonction itérative de recherche dans une liste triée

- **Données** : *liste* : ListeTriée, *v* : TypeValeur
- **Résultat** : un booléen égal à VRAI si *v* est dans *liste*, à FAUX sinon
- **Algorithme**

19

```
fonction estElémentListe ( liste : ListeTriée, v : TypeValeur ) : Booléen
        tant que non estListeVide ( liste ) faire
                si valeurElémentListe ( liste ) < v alors
                        liste ← suiteListe ( liste )
                sinon si valeurElémentListe ( liste ) = v alors
                                retourner VRAI
                        sinon
                                retourner FAUX
                        fsi
                fsi
        ftq
        retourner FAUX
ffct estElémentListe
```

– **Fonction C**

```
Booleen estElementListe ( ListeTriee liste, int v ) {

   Element *ptrElementCourant = liste; // pointeur sur l'élément courant

   while ( ptrElementCourant != NULL ) {

     if ( ptrElementCourant->valeur < v )

       // on poursuit la recherche dans le reste de la liste
       ptrElementCourant = ptrElementCourant->lien;

     else {
       if ( ptrElementCourant->valeur == v )
         return VRAI;    // v est dans la liste

       else
        return FAUX; // v n'est pas dans la liste
     }
   }
   return FAUX; // v est supérieure à toutes les valeurs
                // présentes dans la liste
}
```

3.6 Fonction récursive de recherche dans une liste triée

- **Données** : *liste* : ListeTriée, *v* : TypeValeur
- **Résultat** : un booléen égal à VRAI si *v* est dans *liste*, à FAUX sinon
- **Algorithme**

fonction estElémentListe (liste : ListeTriée, v : TypeValeur) : Booléen

si estListeVide (liste) **ou** valeurElémentListe (liste) $>$ v **alors**
retourner FAUX
sinon si valeurElémentListe (liste) $=$ v **alors**
retourner VRAI
sinon
retourner estElémentListe (suiteListe (liste), v)
fsi
fsi

ffct estElémentListe

20

– **Fonction C**

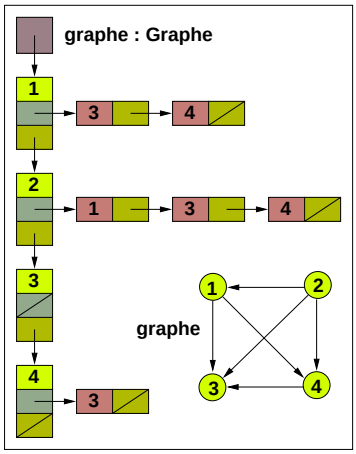
```
Booleen estElementListe ( ListeTriee liste, int v ) {

   if ( liste == NULL || liste->valeur > v )
     return FAUX;   // v n'est pas dans la liste

   else {
     if ( liste->valeur == v )
       return VRAI;    // v est dans la liste

     else

       // on poursuit la recherche dans le reste de la liste
       return estElementListe ( liste->lien , v );
   }
}
```

21

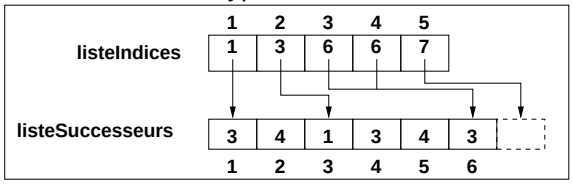
3.7 Applications de la structure de liste -

- **Stockage d'un ensemble évolutif d'éléments**
  - optimisation des opérations d'insertion et de suppression
  - recherche d'un élément parmi n
    - complexité proportionnelle à $n/2$ si la liste est triée
- **Implémentation d'un logiciel de traitement de texte**
  - un texte est représenté par une liste de paragraphes
- **Représentation d'un graphe**
  - un ensemble V de sommets
  - un ensemble E d'arcs $e = (v_1, v_2), v_1, v_2 \in V$

- utilisation d'une liste de listes (bag)
  - listes de successeurs



- **Implémentation d'un logiciel de traitement de texte**
  - quand vous insérez un paragraphe au milieu d'un texte existant, le logiciel de traitement de texte ne va pas décaler la fin du texte. Il va établir, à partir de la position d'insertion, un chaînage vers le nouveau paragraphe.
  - mais dès qu'il y aura sauvegarde, le texte sera reconstitué en prenant en compte les chaînages liés aux récentes modifications.
- **Représentation d'un graphe**
  - là, il faut « enfoncer le clou ». Les structures **flexibles** (gérées par chaînage) ne sont pas la panacée universelle. Elles sont utiles pour représenter un ensemble de données qui évolue dans le temps (pour lesquelles il faut pouvoir insérer ou supprimer un élément de manière efficace).
  - l'utilisation d'une liste de listes chaînées pour représenter un graphe ne s'impose que si la structure du graphe évolue dans le temps.
  - si la structure du graphe n'évolue pas (et c'est souvent le cas dans les problèmes d'optimisation dans les graphes), d'autres représentations sont plus économes en espace mémoire. La liste de listes peut être représentée avec deux tableaux.
  - pour l'exemple du transparent, si $|V| = n$ et $|E| = m$, il faut :
    - un tableau *listeIndices* de $n + 1$ éléments de type Naturel
    - un tableau *listeSuccesseurs* de m éléments de type Naturel



  - les successeurs du sommet v_i d'indice *i* sont accessibles dans le tableau *listeSuccesseurs* entre les indices *listeIndices[i]* et *listeIndices[i+1]-1*

TD9

LES FONCTIONS C À PRÉPARER

TD9

Les fonctions C à préparer

Remarque : Nous allons travailler sur une liste linéaire triée d'entiers. L'objectif des TP11-12 étant de gérer un répertoire de personnes, il faudra adapter vos fonctions C → utiliser la fonction **`strcmp`**.

Les fonctions de recherche et de de parcours sont dans le cours.

1. **durant la séance de TD**
   (a) *nouvelleListe*
   (b) *estListeVide*
   (c) *enlister* en itératif
   (d) *enlister* en récursif

2. **avant la séance de TP**
   (a) *nombreElementsListe* en itératif
   (b) *nombreElementsListe* en récursif
   (c) *delister* en itératif
   (d) *delister* en récursif

TP11-TP12

LISTE LINÉAIRE TRIÉE : UN RÉPERTOIRE DE PERSONNES

AOÛT 2006

Liste linéaire triée : Gestion d'un répertoire de personnes

1. Définir les fonctions implémentant les opérations du type abstrait *Liste* (triée).
   Les fonctions *enlister*, *delister*, *nombreElementsListe* et *parcourirListe* seront définies dans un premier temps avec un schéma itératif.

   Pour cela vous disposez des fichiers :

   (a) *listeTriee.h* : fichier d'en-tête définissant le type *Liste* et contenant la déclaration des fonctions manipulant une liste linéaire triée de personnes gérée par pointeurs.

listeTriee.h

```
// définition des types nécessaire a la représentation d'une liste triée

#include "personne.h"
#include "booleen.h"

typedef struct element { Personne valeur;
                         struct element *lien; } Element;

typedef Element *Liste;

// prototypes des fonctions itératives gérant la liste triee
Liste nouvelleListe(void);
Liste enlister(Liste l,Personne p); // style fonctionnel
Liste delister(Liste l,Personne p); // style fonctionnel
Booleen estListeVide(Liste l);
int nombreElementsListe(Liste l);
void parcourirListe(Liste l);
```

   (b) *personne.h* : ce fichier contient la définition du type *Personne* ainsi que les prototypes des fonctions permettant de saisir une personne (nom, prénom, adresse et numéro de téléphone), de saisir un nom (pour la fonction *delister*) et d'afficher une personne (pour la fonction *parcourir-Liste*).

personne.h

```
// définition du type Personne nécessaire à la gestion d'un répertoire

#define LONGUEUR_NOM           (15+1)
#define LONGUEUR_PRENOM        (15+1)
#define LONGUEUR_ADRESSE       (30+1)
#define LONGUEUR_NO_TELEPHONE  (10+1)

typedef struct { char nom[LONGUEUR_NOM];
                 char prenom[LONGUEUR_PRENOM];
                 char adresse[LONGUEUR_ADRESSE];
                 char noTelephone[LONGUEUR_NO_TELEPHONE]; } Personne;

// prototype des fonctions
void afficherPersonne(Personne p);
Personne saisirPersonne(void);
Personne saisirNom(void); // seul le champ nom est signifiant
```

   (c) *personne.c* : ce fichier contient les définitions des fonctions déclarées dans le fichier *personne.h*.

(d) *repertoire.c* : ce fichier contient la définition de la fonction *main* qui propose un menu pour enlister et delister une personne ainsi qu'afficher le nombre de personnes dans le répertoire et les entrées du répertoire. Le choix de l'utilisateur est géré avec l'instruction **`switch`** du langage C.

(e) *booleen.h* : ce fichier contient la définition du type *Booleen*.

(f) le fichier *makefile* permettant de générer l'application *répertoire*.

Vous devez écrire :
le fichier *listeTriee.c* définissant les fonctions qui sont déclarées dans le fichier *listeTriee.h* et qui gèrent une liste de personnes triée suivant le champ nom.
Il faudra utiliser la fonction *strcmp* déclarée dans le fichier *string.h* qui permet de comparer deux chaînes de caractères (**`man strcmp`**). Attention à la distinction minuscule majuscule.

2. Dans un autre répertoire, développez la même application en utilisant des fonctions C récursives. Vous disposez du fichier *listeTriee.h*.

listeTriee.h

```
// définition des types nécessaires a la représentation d'une liste triée

#include "personne.h"
#include "booleen.h"

typedef struct element { Personne valeur;
                         struct element *lien; } Element;

typedef Element *Liste;

// prototypes des fonctions récursives gérant la liste triee

Liste nouvelleListe(void);
void enlister(Liste *pL,Personne p); // style procédural
void delister(Liste *pL,Personne p); // style procédural
Booleen estListeVide(Liste l);
int nombreElementsListe(Liste l);
void parcourirListe(Liste l);
```

Les implémentations des fonctions *nouvelleListe* et *estListeVide* sont identiques et les fichiers *personne.h*, *personne.c*, *repertoire.c*, *booleen.h* et *makefile* à utiliser sont les mêmes que précédemment.

TP11-TP12

C12

LES STRUCTURES ARBORESCENTES

Objectifs du C12

2

- Connaître le principe
  - de la structure d'arbre binaire
- Connaître les détails d'implémentation
  - de la structure d'arbre binaire de recherche

3

Contenu du C12

1 La structure d'arbre binaire

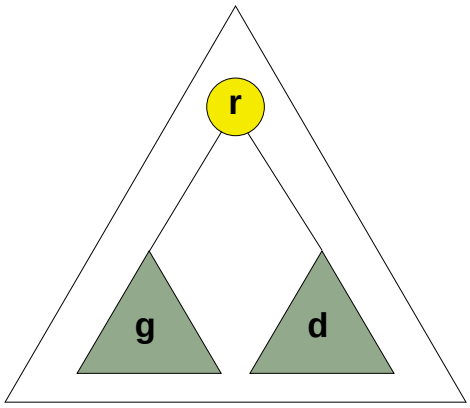
4

1.1 Définition de l'arbre binaire

5

- **Ensemble d'éléments de même type en nombre variable** ≥ 0 **appelés** *nœuds*
- **Organisation hiérachique** des *nœuds*

- **Un arbre binaire** a est :
  - soit **vide** : $a = ()$
  - soit composé de :
    - un nœud **racine** r
    - deux **arbres binaires disjoints** :
      - un **sous-arbre gauche** g
      - un **sous-arbre droit** d
    - $a = (r, g, d)$
  - la définition est **récursive**



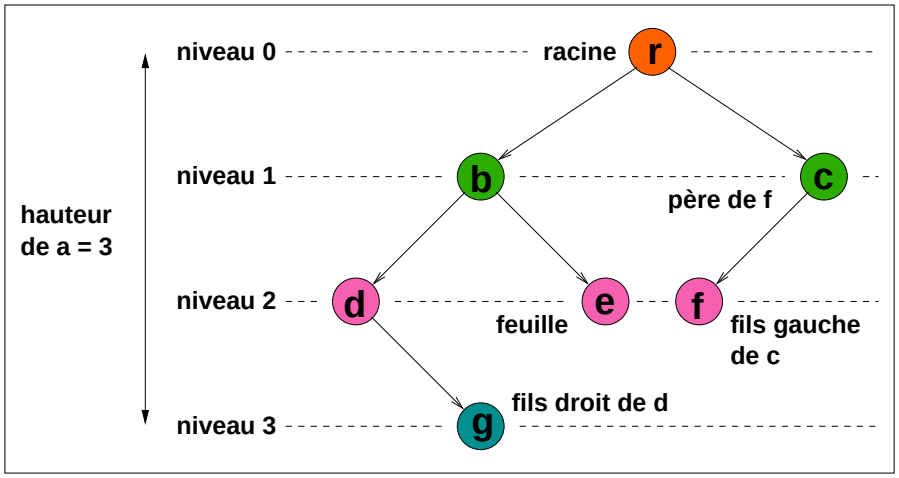
- **Généralisation**
  - un arbre est composé de :
    - un nœud racine
    - un nombre quelconque de sous-arbres

- **Arbre vs arborescence**
  - structures de données
    - arbre : concept orienté
  - théorie des graphes
    - arbre : concept non orienté
    - arborescence : concept orienté $\rightarrow$ arbre (structures de données)

6

1.2 Schéma d'un arbre binaire

$a = (\ r, (\ b, (\ d, (), (\ g, (), ()\ )), (\ e, (), ()\ )\ ), (\ c, (\ f, (), ()\ ), ()\ )\ )$



- **Equilibrage de l'arbre binaire**
  - soit n le nombre de nœuds de l'arbre binaire
  - la hauteur h de l'arbre binaire varie entre $\lfloor log_2 n \rfloor$ et $n-1$
    - $h = \lfloor log_2 n \rfloor$: l'arbre binaire est **bien équilibré**.
    - $h = n-1$: l'arbre binaire est **dégénéré** (cas *filaire* ou *filiforme*)
  - construire ou garder un arbre binaire bien équilibré est très important pour le problème de la recherche d'un élément :
    - arbre binaire équilibré $\rightarrow$ complexité en $O(\log_2 n)$
    - arbre binaire dégénéré $\rightarrow$ complexité en $O(n)$

1.3 Terminologie

7

- **L'*arité* d'un nœud**
  - nombre de sous-arbres non vides (0, 1 ou 2)
  - un nœud d'arité nulle est appelé *feuille*
- **Un *chemin* dans un arbre**
  - une séquence de nœuds $(n_0, n_1, \ldots, n_i, \ldots, n_p)$ où n_{i-1} est le père de n_i
  - la longueur du chemin $(n_0, n_1, \ldots, n_i, \ldots, n_p)$ est égale à p
- **Le *niveau* d'un nœud**
  - c'est la longueur de l'**unique** chemin allant de la racine à ce nœud
  - le niveau du nœud racine est égal à 0
- **La *hauteur* d'un arbre**
  - c'est le **maximum** des niveaux de tous les nœuds
  - la hauteur d'un arbre **réduit au nœud racine** vaut 0
  - la hauteur d'un arbre **vide** vaut par **convention** -1

1.4 Les opérations sur l'arbre binaire

■ **Chaque nœud de l'arbre contient une valeur de type** *TypeValeur*

8

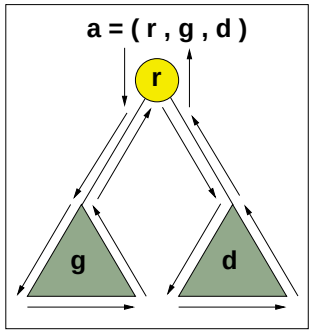
nouvelArbre	:		→	Arbre
estArbreVide	:	Arbre	→	Booléen
valeurRacineArbre	:	Arbre $-\{\emptyset\}$	→	TypeValeur
construireArbre	:	Arbre $\times$ Arbre $\times$ TypeValeur	→	Arbre
sousArbreGauche	:	Arbre $-\{\emptyset\}$	→	Arbre
sousArbreDroit	:	Arbre $-\{\emptyset\}$	→	Arbre
estElémentArbre	:	Arbre $\times$ TypeValeur	→	Booléen

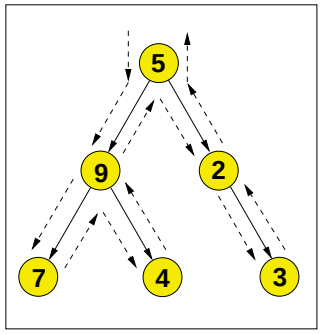
– **nouvelArbre**
  – renvoie un arbre vide

– **construireArbre**
  – la méthode consiste à créer un nœud contenant l'argument de type *TypeValeur* qui aura pour sous-arbres les deux arguments de type *Arbre*.
  – nous utiliserons une autre opération pour construire progressivement un arbre binaire particulier : l'arbre binaire de recherche.

9

1.5 Parcours en profondeur





- **Parcours RGD** de a
  - traiter la racine r
  - parcours RGD de g
  - parcours RGD de d
- → **(5, 9, 7, 4, 2, 3)**

- **Parcours GRD** de a
  - parcours GRD de g
  - traiter la racine r
  - parcours GRD de d
- → **(7, 9, 4, 5, 2, 3)**

- **Parcours GDR** de a
  - parcours GDR de g
  - parcours GDR de d
  - traiter la racine r
- → **(7, 4, 9, 3, 2, 5)**

– **Parcours** → implémentation récursive très concise
  – les trois types de parcours diffèrent selon la position de l'appel de l'opération *traiter*
    – parcours RGD : **avant** les appels récursifs de l'opération *parcours*
    – parcours GRD : **entre** les appels récursifs de l'opération *parcours*
    – parcours GDR : **après** les appels récursifs de l'opération *parcours*

– **Définition de la procédure récursive parcoursGRD** (avec affichage des valeurs contenues dans chaque nœud)

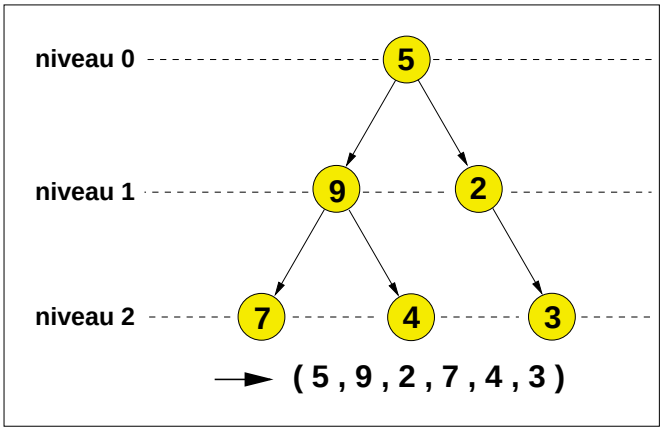
```
procédure parcoursGRD ( donnée arbre : Arbre )

    si non  estArbreVide ( arbre ) alors

        parcoursGRD ( sousArbreGauche ( arbre ) )
        afficher ( valeurRacineArbre ( arbre ) )
        parcoursGRD ( sousArbreDroit ( arbre ) )

    fsi

fproc parcoursGRD
```

10

1.6 Parcours en largeur

■ **Principe**

- ♦ parcours de l'arbre par **niveau** en partant du niveau **zéro**
- ♦ chaque niveau est parcouru de **gauche** à **droite**



– **Parcours en largeur** : comme nous l'avons déjà mentionné, le parcours en largeur d'un arbre nécessite une structure de file.

– **Définition de la procédure itérative parcoursEnLargeur** (avec affichage des valeurs contenues dans chaque nœud)

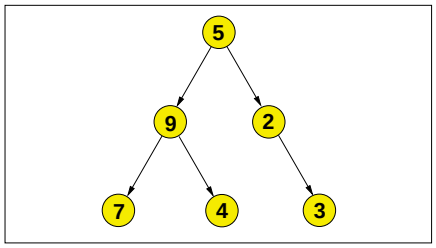
procédure parcoursEnLargeur (**donnée** arbre : Arbre)

```
    file : File // de Nœud
    file ← nouvelleFile ( )
    si non estArbreVide ( arbre ) alors
        enfiler ( file , arbre )
    fsi
    tant que non estFileVide ( file ) faire
        arbre ← valeurTête ( file )
        défiler ( file )
        afficher ( valeurRacineArbre ( arbre ) )
        si non estArbreVide ( sousArbreGauche ( arbre ) ) alors
            enfiler ( file , sousArbreGauche ( arbre ) )
        fsi
        si non estArbreVide ( sousArbreDroit ( arbre ) ) alors
            enfiler ( file , sousArbreDroit ( arbre ) )
        fsi
    ftq
```

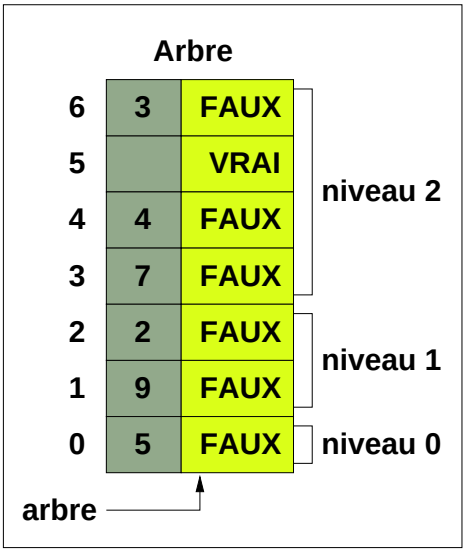
fproc parcoursEnLargeur

11

1.7 Représentation par tableau



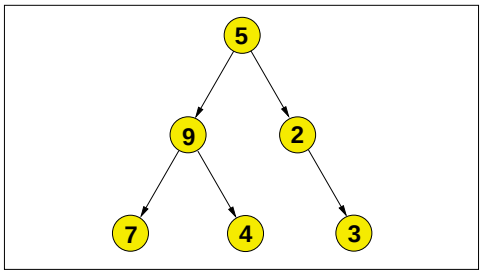

```
#define NB_MAX_NOEUDS 7
typedef enum {FAUX,VRAI} Booleen;

typedef struct {
   int    valeur;
   Booleen libre; } Noeud;
typedef Noeud Arbre [NB_MAX_NOEUDS];

Arbre arbre;
nouvelArbre ( arbre );
```



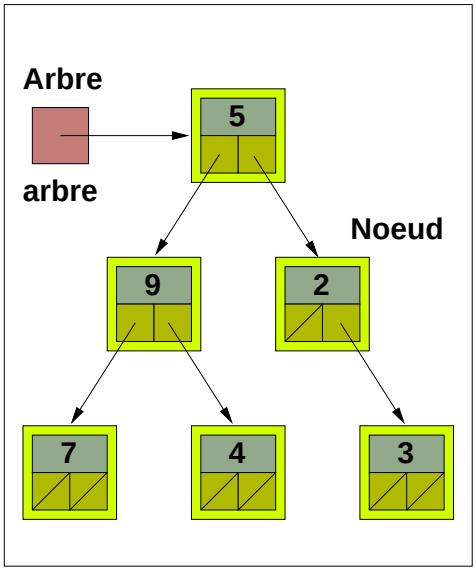
- **Organisation**
  - pour stocker un arbre binaire de hauteur h, il faut un tableau de $2^{h+1} - 1$ éléments.
  - le nœud racine a pour indice 0 (en C)
  - soit le nœud d'indice i du tableau :
    - son fils gauche a pour indice $2i + 1$
    - son fils droit a pour indice $2(i + 1)$
    - si $i \neq 0$ son père a pour indice $\lfloor (i - 1)/2 \rfloor$
- **Remarque sur l'utilisation du type booléen**
  - comme il est défini ici, il est stocké comme un type **`int`** (quatre octets sous *Linux*)
    $\rightarrow$ un tableau de 7 éléments de type **`Noeud`** nécessitera 56 octets.
  - on pourrait redéfinir le type **`Booleen`** pour qu'il soit stocké sur un octet :
    ```
    #define FAUX 0
    #define VRAI 1
    typedef char Booleen;
    ```
  - mais pour des raisons d'alignement en mémoire sur des frontières de quatre octets (voir le module AM12), le tableau résultant nécessitera le même nombre d'octets.
  - une implémentation efficace nécessiterait
    - soit d'utiliser une « valeur conventionnelle » pour indiquer une case libre dans le tableau, si cela est possible.
    - soit une *bitmap* des éléments du tableau : un bit à 1 indique une case occupée, un bit à 0 une case libre.

12

1.8 Représentation par pointeurs




```
typedef struct noeud {
    int valeur;
    struct noeud *sag;
    struct noeud *sad; } Noeud;

typedef Noeud *Arbre;

Arbre arbre = nouvelArbre ();
```



– **Autre possibilité de définition**

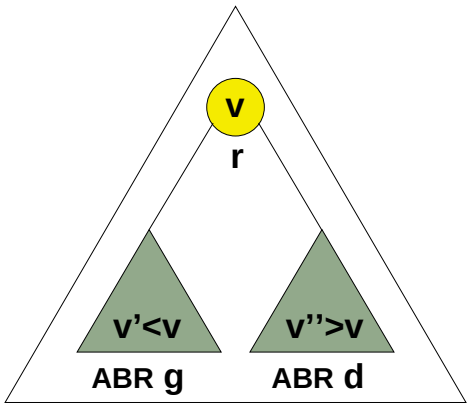
```
typedef struct noeud *Arbre;

typedef struct noeud { int valeur;
                       Arbre  sag;
                       Arbre  sad; } Noeud;

Arbre arbre = nouvelArbre ();
```

1.9 L'arbre binaire de recherche (ABR)

13

■ **Un arbre binaire de recherche (ABR) est un arbre où les nœuds sont organisés en fonction d'un ordre qui existe sur les valeurs qu'ils contiennent**

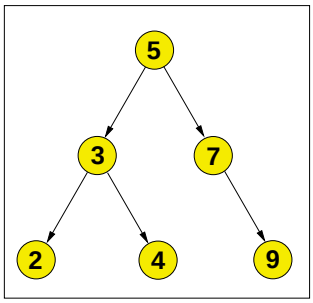
■ **Un ABR** a est :
- ♦ soit **vide** : $a = ()$
- ♦ soit **non vide** : $a = (r, g, d)$, **si** v est la valeur de la racine r, **alors**
  - ► **si** g est non vide, **alors**
    - ★ $\forall\, v' \in g, v' < v$
    - ★ g est un ABR
  - ► **si** d est non vide, **alors**
    - ★ $\forall\, v'' \in d, v'' > v$
    - ★ d est un ABR
- ♦ il n'y a **pas de doublon**

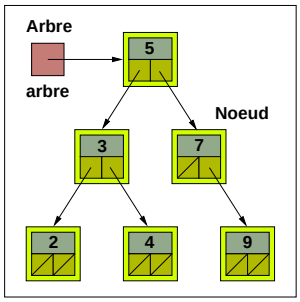


– **La définition est récursive**
  – pour trouver le plus petit (resp. plus grand) élément de l'arbre, il faut, en commençant par la racine, s'orienter systématiquement vers le sous-arbre gauche (resp. droit) du nœud courant.
  Quand le nœud courant n'a plus de sous-arbre gauche (resp. droit) la valeur qu'il contient est la plus petite (resp. plus grande) de l'arbre.

14

1.10 ABR : exemple et opérations spécifiques





ajouterElémentArbre : Arbre $\times$ TypeValeur $\rightarrow$ Arbre
supprimerElémentArbre : Arbre $\times$ TypeValeur $\rightarrow$ Arbre
parcourir<MODE> : Arbre $\times$ Fonction $\rightarrow$
où **MODE = RGD, GRD, GDR**
et une fonction, par exemple **traiterValeur** qui effectue une action sur la valeur
traiterValeur : TypeValeur $\rightarrow$

- **Opération** *ajouterElémentArbre*
  - il y a toujours une place pour insérer un nouvel élément dans l'arbre.
  - pour la position d'insertion, voir les algorithmes de recherche plus loin.
  - mais suivant la succession des valeurs qui sont incorporées dans l'arbre, celui-ci peut ne pas être « **bien équilibré** », voir quasiment « **dégénéré** ».
  - d'où la nécessité de développer des techniques qui permettent de **créer** ou de **maintenir** l'arbre « bien équilibré » $\rightarrow$ *arbre AVL*.
- **Opération** *supprimerElémentArbre*
  - pour la recherche de l'élément à supprimer, voir les algorithmes de recherche plus loin.
  - les différents cas :
    - arbre réduit à un nœud feuille : sa suppression rend l'arbre vide.
    - suppression d'un nœud feuille (non racine) : conduit à mettre à **`NULL`** le champ **sag** (resp. **sad**) de son père suivant que ce nœud est fils gauche (resp. fils droit) de celui-ci.
    - suppression d'un nœud n non feuille ayant un fils gauche (resp. fils droit) : conduit à « sauter un niveau » dans la hiérarchie et remplacer le sous-arbre (dont n est racine) par le sous-arbre gauche (resp. droit) de n. Si le nœud à supprimer n'a pas de père, le sous-arbre devient le nouvel arbre (modification de la racine de l'arbre).
    - suppression d'un nœud ayant un fils gauche et un fils droit : conduit à rechercher le plus petit élément du sous-arbre droit du nœud à supprimer et de le transformer en la nouvelle racine du sous-arbre dont la racine est le nœud à supprimer (on pourrait de façon équivalente rechercher le plus grand élément du sous-arbre gauche pour en faire la nouvelle racine . . .).
- **Opération** *parcourir<MODE>* (avec affichage)
  - affiche les valeurs contenues dans l'arbre par ordre croissant (mode **GRD**) ou décroissant (mode **DRG**).

1.11 Fonction itérative de recherche dans un ABR

- **Données** : *arbre* : Arbre (ABR), *v* : TypeValeur
- **Résultat** : un booléen égal à VRAI si *v* est dans *arbre*, et à FAUX sinon
- **Algorithme**

15

```
fonction estElémentArbre ( arbre : Arbre, v : TypeValeur ) : Booléen
    tant que non estArbreVide ( arbre ) faire
        si v = valeurRacineArbre ( arbre ) alors
            retourner VRAI
        sinon si v < valeurRacineArbre ( arbre ) alors
                arbre ← sousArbreGauche ( arbre )
            sinon
                arbre ← sousArbreDroit ( arbre )
            fsi
        fsi
    ftq
    retourner FAUX
ffct estElémentArbre
```

– **Fonction C**

```
Booleen estElementArbre ( Arbre arbre, int v ) {

   Noeud *ptrNoeudCourant = arbre ; // pointeur courant sur un noeud

   while ( ptrNoeudCourant != NULL ) {

     if ( v == ptrNoeudCourant->valeur )
       return VRAI;                        // on a trouve v dans a
     else if ( v < ptrNoeudCourant->valeur )

            // on poursuit la recherche dans le SAG du noeud courant
            ptrNoeudCourant = ptrNoeudCourant->sag;

          else
            // on poursuit la recherche dans le SAD du noeud courant
            ptrNoeudCourant = ptrNoeudCourant->sad;
   }
   return FAUX; // v n'est pas dans l'arbre
}
```

1.12 Fonction récursive de recherche dans un ABR

16

- **Données** : *arbre* : Arbre (ABR), *v* : TypeValeur
- **Résultat** : un booléen égal à VRAI si *v* est dans *arbre*, et à FAUX sinon
- **Algorithme**

```
fonction estElémentArbre ( arbre : Arbre, v : TypeValeur ) : Booléen
    si estArbreVide ( arbre ) alors
        retourner FAUX
    sinon si v = valeurRacineArbre ( arbre ) alors
            retourner VRAI
        sinon si v < valeurRacineArbre ( arbre ) alors
                retourner estElémentArbre ( sousArbreGauche ( arbre ), v )
            sinon
                retourner estElémentArbre ( sousArbreDroit ( arbre ), v )
            fsi
        fsi
    fsi
ffct estElémentArbre
```

– **Fonction C**

```
Booleen estElementArbre ( Arbre arbre, int v ) {

   if ( arbre == NULL )
     return FAUX;   // v n'est pas dans l'arbre

   else {
     if ( v == arbre->valeur )
       return VRAI;    // v est dans l'arbre

     else if ( v < arbre->valeur )

            // on poursuit la recherche dans le SAG du noeud courant
            return estElementArbre ( arbre->sag , v );

          else
            // on poursuit la recherche dans le SAD du noeud courant
            return estElementArbre ( arbre->sad , v );
   }
}
```

1.13 Applications des arbres binaires -

17

- **Recherche dans un ensemble de n valeurs**
  - complexité proportionnelle à $\log_2 n$ dans une structure de *tas*
- **Tri d'ensemble de n valeurs**
  - parcours **GRD** d'un arbre binaire de recherche
  - utilisation d'une structure de *tas* (*heap sort*)
  - complexité en $O(n \log_2 n)$
- **Représentation d'une expression arithmétique**
  - parcours **GRD** $\rightarrow$ notation infixée
  - parcours **RGD** $\rightarrow$ notation préfixée
  - parcours **GDR** $\rightarrow$ notation postfixée (notation polonaise inverse)
- **Méthodes de compression**
  - codage de *Huffman* $\rightarrow$ utilise des arbres binaires
  - compression d'image par *quadtree* (arbre quaternaire)

– **La structure de** *tas* (pour la recherche de l'élément minimum dans un ensemble de valeur)
  – arbre binaire avec la propriété suivante : Soit v la valeur contenue dans un nœud donné, toutes les valeurs contenues dans les sous-arbres de ce nœud sont supérieures ou égales à v.
  – il faut inverser la propriété si on recherche l'élément de valeur maximum.
– **Algorithme** *heap sort*
  – utilise une structure de *tas*.
  – un élément de valeur minimum est forcément contenu dans la racine de l'arbre.
  – cet élément est alors supprimé.
  – l'arbre est réorganisé pour maintenir la propriété avec une complexité en $O(\log_2 n)$.
  – le tas est généralement représenté par un tableau.
– **Codage de** *Huffman*
  – compression réversible de fichier.
  – utilise la fréquence d'apparition des différents caractères dans le fichier.
  – un caractère apparaissant très fréquemment sera codé sur un petit nombre de bits.
  – un caractère apparaissant rarement sera codé sur un plus grand nombre de bits.
  – le « codage » des caractères nécessite la construction progressive d'un arbre de décompte d'occurrences à partir d'arbres binaires.
– **Compression d'image par** *quadtree*
  – compression d'image en noir et blanc ou en niveaux de gris.
  – utilise un arbre *quaternaire* : chaque nœud non feuille a exactement quatre fils.
  – la racine de l'arbre correspond à l'image entière.
  – l'image est divisée en quatre *quadrants* (les quatre fils de la racine), et ceci récursivement.
  – dès qu'un quadrant devient homogène, la récursivité est stoppée : le nœud reste une feuille.

TD10

LES FONCTIONS C À PRÉPARER

AOÛT 2006

Les fonctions C à préparer

Les fonctions C testant la présence d'une valeur dans l'arbre sont dans le cours.

1. **durant la séance de TD**
   (a) *nouvelArbre*
   (b) *estArbreVide*
   (c) *parcoursGRD*
   (d) *supprimerToutGDR*
   (e) *ajouterRecur*
   (f) quelques indications pour la fonction *supprimerIter*

2. **en hors présentiel**
   (a) *ajouterIter*
   (b) *supprimerIter*

TP13-TP14

ARBRE BINAIRE DE RECHERCHE : TRI DE VALEURS ENTIÈRES

AOÛT 2006

Arbre binaire de recherche : Tri de valeurs entières

Vous disposez de trois fichiers :

1. le fichier d'en-tête *arbreBinaire.h* qui contient les définitions des types *Noeud* et *Arbre* ainsi que la déclaration des fonctions implémentant les opérations du type abstrait Arbre.

arbreBinaire.h

```
#ifndef _ARBRE_BINAIRE_H
#define _ARBRE_BINAIRE_H

// Interface du type Arbre Binaire de Recherche (Arbre)
// contenant des valeurs entières sans doublon

#include "booleen.h"

// Redéfinition du type int
typedef int Type;

// Le type Noeud définit la représentation d'un élément de l'arbre

typedef struct noeud {  Type valeur;
                        struct noeud *sag;
                        struct noeud *sad; } Noeud;

// Le type Arbre définit la représentation d'un arbre binaire de recherche

typedef Noeud *Arbre;

//Cette fonction initialise un arbre
Arbre nouvelArbre ( void );

// Cette fonction retourne VRAI si l'arbre a est vide, FAUX sinon
Booleen estArbreVide ( Arbre a );

// Ces deux fonctions ajoutent un nouvel élément de valeur v à l'arbre a
// Comme a est suceptible d'être modifié, passage par adresse
// Version itérative
void ajouterIter ( Arbre *pA, Type v );

// Version récursive
void ajouterRecur ( Arbre *pA, Type v );

// Cette fonction itérative supprime l'élément de valeur v de l'arbre a
// Comme a est suceptible d'être modifié, passage par adresse
void supprimerIter ( Arbre *pA, Type v );

// Ces deux fonctions testent si la valeur v est présente dans l'arbre a
// Si v est dans l'arbre a, la fonction retourne VRAI, sinon FAUX
// Version itérative
Booleen appartientIter ( Arbre a, Type v );
// Version récursive
Booleen appartientRecur ( Arbre a, Type v );
```

arbreBinaire.h (suite)

```
// Cette fonction parcourt en profondeur l'arbre a de maniere GRD
// Elle affiche dans la fenêtre les valeurs rencontrées par ordre croissant
void parcoursGRD ( Arbre a );

// Cette fonction parcourt en profondeur l'arbre a de maniere DRG
// Elle affiche dans la fenêtre les valeurs rencontrées par ordre décroissant
void parcoursDRG ( Arbre a );

// Cette fonction utilise un parcours GDR pour libérer tous les
// noeuds de l'arbre a. Apres exécution, l'arbre est vide
// Comme a sera modifié, passage par adresse
void supprimerToutGDR ( Arbre *pA );

#endif
```

2. le fichier d'entête *booleen.h* qui contient la définition du type Booleen.
3. le fichier *useArbreBinaire.c* qui vous permettra de tester rapidement votre code.

Vous devez écrire :

1. le fichier *arbreBinaire.c* contenant les définitions des fonctions déclarées dans *arbreBinaire.h*
2. le fichier *makefile* permettant d'obtenir l'application *useArbreBinaire*.

Allez-y progressivement. Il vous suffit de mettre en commentaire dans l'instruction **`switch`** du fichier *useArbreBinaire.c* les cas correspondants aux fonctions qui ne sont pas encore implémentées.
Vous pouvez vous baser sur une réalisation en trois paliers :

1. **Premier niveau** :
   (a) *nouvelArbre*
   (b) *estArbreVide*
   (c) *ajouterRecur*
   (d) *parcoursGRD*
2. **Deuxième niveau** :
   (a) *appartientRecur*
   (b) *appartientIter*
   (c) *parcoursDRG*
   (d) *supprimerToutGDR*
3. **Troisième niveau** :
   (a) *ajouterIter*
   (b) *supprimerIter*

TP13-TP14

DÉVELOPPEMENT D'APPLICATION

MINI-TABLEUR-ENVELOPPE CONVEXE

Mini-tableur (proposé par Alain Pétrowski)

1. **Présentation du mini-tableur**

   Lorsqu'on lance la commande

   $ tableur 8 6 ↩

   il s'affiche un écran similaire à la figure ci-dessous :

```
    |     A     |     B     |     C     |     D     |     E     |     F     |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  1 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  2 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  3 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  4 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  5 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  6 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  7 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  8 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+

Cellule ?
```

   L'écran affiche un tableau de *l* lignes et de *c* colonnes où *l* et *c* sont les paramètres de la commande. Si aucun paramètre n'est donné, *l* et *c* prennent les valeurs 8 et 6 par défaut. L'affichage représente le contenu de la feuille de calcul de notre tableur. Il ne peut gérer qu'une seule feuille.

   Chaque cellule de la feuille est désignée par un couple de coordonnées formé de l'indice de colonne ('A' à 'F' pour 6 colonnes) concaténé avec l'indice de ligne ('1' à '8' pour 8 lignes). Une cellule peut être vide ou contenir une valeur ou une expression arithmétique simple. Pour modifier le contenu de la cellule «$A1$», il suffit de taper «**a1**↩» ou «**A1**↩» après l'invite «**Cellule?**». Le tableur affiche alors le contenu de cette cellule suivi d'une autre invite : un point d'interrogation. A ce moment, le programme attend qu'une valeur ou une expression arithmétique associée à cette cellule soit saisie au clavier. Ce contenu peut être

   - inchangé, lorsqu'un simple retour-chariot est tapé.
   - une simple valeur, éventuellement précédée du signe $-$.
   - une expression arithmétique simple composée d'un argument suivi d'un opérateur et d'un autre argument, par exemple «$D3 + 4.5$», ce qui signifie que la cellule «$A1$» prendra la valeur de la somme du contenu de la cellule «$D3$» avec la valeur 4.5.

   Après avoir saisi le contenu de la cellule «$A1$» et tapé «↩», la feuille de calcul mise à jour s'affiche à nouveau et attend les coordonnées d'une cellule pour en afficher le contenu et éventuellement le modifier. Par exemple, on obtiendra l'affichage suivant :

```
Cellule ? a1

? D3 + 4.5
    |     A     |     B     |     C     |     D     |     E     |     F     |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  1 |        4.5|           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  2 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  3 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  4 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  5 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  6 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  7 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  8 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+

Cellule ?
```

Il est ensuite possible de donner une valeur à la cellule «$D3$», par exemple avec l'expression «$2 * 3.7$» en saisissant

```
Cellule ? d3

? 2 * 3.7
    |     A     |     B     |     C     |     D     |     E     |     F     |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  1 |       11.9|           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  2 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  3 |           |           |           |        7.4|           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  4 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  5 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  6 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  7 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  8 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+

Cellule ?
```

Si on désire modifier le contenu d'une cellule qui n'est pas vide, il suffit de donner ses coordonnées

après l'invite «**Cellule?**» comme pour une cellule vide. Le contenu courant de la cellule est alors affiché au dessus de l'invite permettant de saisir le nouveau contenu.

```
Cellule ? a1
= D3 + 4.5
? C2 / d3
    |     A     |     B     |     C     |     D     |     E     |     F     |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  1 |          0|           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  2 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  3 |           |           |           |        7.4|           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  4 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  5 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  6 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  7 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  8 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+

Cellule ?
```

Si une référence circulaire est présente, un message d'erreur doit être affiché et les calculs doivent être arrêtés. Exemple :

```
Cellule ? c2

? a1 + 1
Référence circulaire !
    |     A     |     B     |     C     |     D     |     E     |     F     |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  1 |   0.135135|           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  2 |           |           |          1|           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  3 |           |           |           |        7.4|           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  4 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  5 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  6 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  7 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+
  8 |           |           |           |           |           |           |
----+-----------+-----------+-----------+-----------+-----------+-----------+

Cellule ?
```

On doit pouvoir quitter le programme en tapant «**q**⟻» ou «**Q**⟻» après l'invite «**Cellule?**». Si une expression tapée après une invite contient davantage de caractères que ce qui est attendu, les caractères en trop seront ignorés et ne devront pas influer sur les saisies suivantes. Si l'expression est erronée, un message d'erreur devra être affiché. Le contenu de la cellule concernée pourra alors être affectée de façon non spécifiée.

2. **Implémentation du mini-tableur**

   (a) A partir de l'énoncé identifier les mots clés qui correspondront aux types de données utilisés par le tableur.

   (b) Déterminer les actions qui associent les différents types précédemment identifiés.

   (c) Identifier les cas d'utilisation du tableur.

   (d) Déterminer les scénarios associés aux cas d'utilisation du tableur.

   (e) Contruire les diagrammes de séquences associés aux différents cas d'utilisation.

   (f) Construire le diagramme des relations entre types.

Enveloppe convexe : Algorithme de Graham (proposé par Gilles Protoy)

1. **Définition**

   L'enveloppe convexe $H(S)$ d'un ensemble S de points dans le plan est le polygone minimal contenant S et dont les sommets sont appelés points extrêmes. Ces points ne peuvent pas être obtenus par une combinaison linéaire convexe d'autres points de S.

2. **Objectifs**

   A partir d'un fichier contenant l'ensemble de points S, extraire les points extrêmes et les ordonner. L'ordre de parcours sera le sens trigonométrique. L'affichage des points et de l'enveloppe dans une fenêtre *X-Window* permettra de visualiser le résultat.
   Il faudra également développer l'application permettant de stocker une instance du problème dans un fichier texte ou binaire.

3. **Borne inférieure de complexité du problème**

   On peut aisément montrer que la complexité de ce problème est équivalente à celle d'un tri par comparaisons. On peut transformer le problème du tri de n valeurs en une détermination d'enveloppe convexe. A la valeur x, on associe un point $p(x)$ du plan sur la parabole d'équation $y = x^2$. Les coordonnées de $p(x)$ seront donc x et x^2. Cette transformation est manifestement réalisée avec une complexité linéaire.
   L'ordre obtenu sur les points extrêmes de l'enveloppe (ils le sont tous dans cette instance) donne un tri de l'ensemble de valeurs initiales. La borne inférieure de complexité pour les tris par comparaisons étant en $nlog_2n$, on obtient une borne inférieure identique pour la détermination de l'enveloppe convexe.

4. **Les Algorithmes**

   Nous allons décrire brièvement les différents algorithmes permettant de résoudre ce problème en partant du plus complexe (au sens fonction de complexité). L'algorithme de *Graham*, un des deux à atteindre la borne inférieure de complexité, sera bien sûr beaucoup plus détaillé.

   (a) **Algorithme naïf**
   Un point est non extrême s'il est à l'intérieur d'un triangle fermé dont les sommets sont des points de S et s'il n'est pas lui-même un sommet de ce triangle.
   L'algorithme naïf consiste, pour chaque point p de S, à énumérer tous les triangles dont les sommets sont des points de $S \setminus \{p\}$. S'il existe un triangle tel que la propriété énoncée ci-dessus est vérifiée, le point est déclaré non extrême. La complexité de l'algorithme naïf est en $O(n^4)$.

   (b) **Algorithme des arêtes extrêmes**
   Au lieu d'identifier les sommets extrêmes, on peut déterminer les arêtes extrêmes (les arêtes du polygone). Intuitivement une arête est extrême si tous les sommets non-extrémité de cette arête sont à sa gauche, ceci à condition de lui choisir une orientation correcte.
   Cet algorithme énumère tous les couples ordonnés de sommets de S. Il élimine ceux pour lesquels il existe au moins un point p de S qui ne soit pas à leur gauche. Sa complexité est en $O(n^3)$.

   (c) **Algorithme « Gift Wrapping »**
   Il « emballe » progressivement l'ensemble de points. On part du point d'ordonnée minimale et on calcule, à chaque itération, l'angle formé par l'arête courante (p_{i-1}, p_i) avec chaque arête (p_i, p), p non inclus. On choisit le sommet dont l'angle associé est minimal. La complexité de cet algorithme est en $O(n^2)$.

(d) **Algorithme « QuickHull »**
Il cherche à éliminer, dès la première itération, le plus grand nombre possible de points. Considérons, sans aborder les cas particuliers, les points p_1 et p_3 d'ordonnée minimale et maximale et les points p_4 et p_2 d'abscisse minimale et maximale. Ils sont bien sûr extrêmes et tous les points inclus dans le quadrilatère fermé qu'ils déterminent peuvent être évincés de la recherche.
Il reste à traiter les quatre coins. Considérons celui déterminé par le segment (p_1, p_2), soit p le point de ce coin le plus éloigné du segment. Il est extrême et tous les points à l'intérieur du triangle fermé déterminé par p_1, p_2 et p sont à écarter. On recommence jusqu'à ce que tous les points de ce coin soient traités.
QuickHull est à l'enveloppe convexe ce que *QuickSort* est au problème du tri. Il fonctionne très bien en pratique mais, dans le pire des cas, sa complexité reste en $O(n^2)$.

(e) **Algorithme divide-and-conquer**
Dans un premier temps il faut trier les points de S suivant les abscisses croissantes. Ceci est réalisé en $O(nlog_2n)$. On divise ensuite l'ensemble S en deux sous-ensembles A et B de cardinalité identique. Les enveloppes $H(A)$ et $H(B)$ sont déterminées de manière récursive. Une phase de réunion des sous-problèmes en $O(n)$ permet d'obtenir $H(S)$ à partir de $H(A)$ et de $H(B)$. La complexité globale de la méthode est en $O(nlog_2n)$, la borne inférieure de complexité est atteinte.

(f) **Algorithme de Graham**
Il reprend la méthode « Gift Wrapping ». Grâce à une phase de tri préalable et à l'utilisation d'une structure de pile, sa complexité est comme le précédent en $O(nlog_2n)$. Détaillons le.

5. **Algorithme de Graham**

La seule contrainte sur l'ensemble S, $|S| = n$, est que deux points ne soient pas confondus.La méthode commence par la recherche du point $p_1 = (x_1, y_1)$ d'ordonnée minimale. S'il en existe plusieurs, on prendra par convention le point d'abscisse minimale.

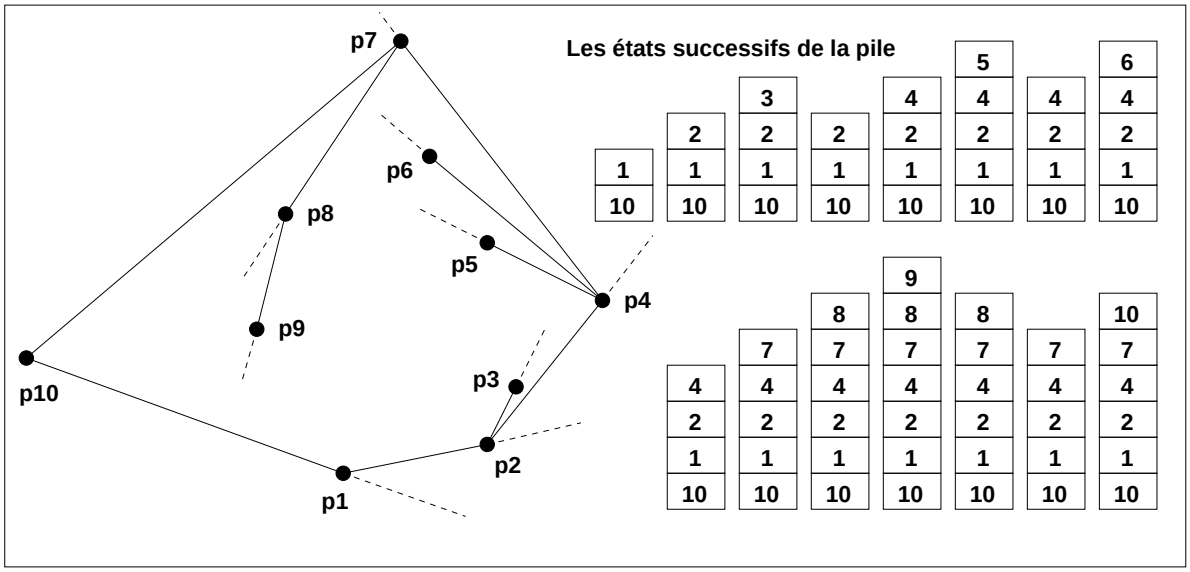
Les points $p_i, 2 \leq i \leq n$, sont alors triés suivant l'angle formé par le segment (p_1, p_i) avec la demi-droite d'équation $x = x_1$. Les angles obtenus vont de $-\pi/2$ à $+\pi/2$, ce qui est plus pratique pour l'utilisation de la fonction arctangente (*atan* de la bibliothèque mathématique). De plus pour gérer les problèmes de colinéarité, si plusieurs angles sont égaux, on ordonnera les points correspondants suivant leurs distances croissantes avec p_1.

Pour l'instant, on peut en déduire que p_n et p_1 sont extrêmes.

Une pile d'indices de points est initialisée et les indices n et 1 sont successivement empilés. Les points $p_i, 2 \leq i \leq n$, sont alors évalués. Si p_i est à droite ou sur la demi-droite orientée générée par les deux points dont les indices sont en sommet de pile, cela signifie qu'on s'est trop orienté vers la gauche (par rapport au sens trigonométrique du parcours). Dans ce cas, l'indice en sommet de pile sera dépilé.

On réitère jusqu'à ce que p_i soit strictement à gauche de la demi-droite orientée générée par les deux points dont les indices sont en sommet de pile. L'indice i est alors empilé.

Voici un exemple du déroulement de l'algorithme et des états successifs de la pile.



6. **Formalisation de l'algorithme**

rechercher le point $p_1 = (x_1, y_1)$ d'ordonnée minimale

si il existe plusieurs points d'ordonnée minimale **alors** // ceci peut être réalisé simultanément
  choisir $p_1 = (x_1, y_1)$ d'abscisse minimale // avec la recherche du point
fsi // d'ordonnée minimale

pour i = 2 : Naturel **à** n
  calculer l'angle formé par le segment (p_1, p_i) avec la demi-droite d'équation $x = x_1$
  calculer la distance entre p_1 et p_i
fpour i

trier les points $p_i, 2 \leq i \leq n$, suivant les angles croissants et en cas d'égalité, suivant les distances croissantes

Pile pile = nouvellePile ( ) // pile d'indices de points

pile ← empiler (pile , n)
pile ← empiler (pile , 1)

pour i = 2 : Naturel **à** n

  tant que p_i est à droite ou sur la demi-droite orientée générée par
      les deux points dont les indices sont en sommet de pile **faire**
    pile ← dépiler (pile)
  ftq

  pile ← empiler (pile , i)

fpour i

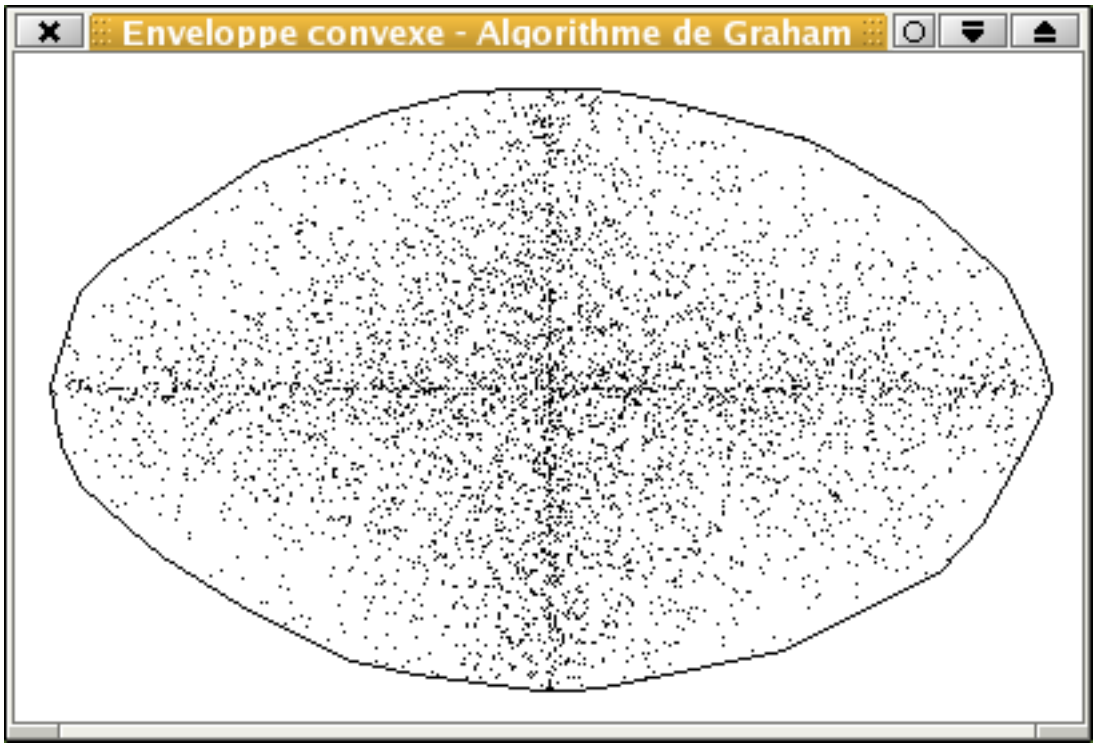
// il n'y a plus qu'à dépiler successivement les indices de la pile et les stocker dans un tableau

// attention, il vont apparaître dans l'ordre inverse du parcours et n sera présent deux fois

7. **Complexité de l'algorithme**

   La complexité de l'algorithme est déterminée par la phase préalable de tri. Le parcours lui-même a une complexité linéaire. Chaque sommet sera empilé une fois et une seule et dépilé au plus une fois.
   La complexité de l'algorithme de Graham est en $O(nlog_2n)$.

8. **Exemple de résultat escompté**



L'ESSENTIEL DU LANGAGE C

LUC LITZLER

AOÛT 2006

309

1 Avant-propos

1.1 Introduction

Ce petit recueil ne prétend pas décrire le langage C de manière exhaustive. Il existe de nombreuses références sur le sujet et nous n'avons pas l'ambition d'en ajouter une nouvelle. Nous invitons le lecteur à consulter les ouvrages cités dans la bibliographie. Notre ambition est de proposer un document à mi-chemin entre l'aide-mémoire et le manuel de référence du langage C destiné à aider les *débutants* en C (néanmoins familiers des concepts communs aux langages impératifs*). Notre approche se veut donc essentiellement pratique et concise. La plupart des concepts sont introduits par des exemples.

Nous donnons des informations destinées à faciliter la mise en œuvre de programmes C dans le contexte naturel de ce langage, celui du système d'exploitation Unix, ainsi que des recommandations de style pour prendre au plus vite de bonnes habitudes (et surtout éviter de laisser s'en installer de mauvaises ! !), ainsi qu'un aide-mémoire rapide mettant en correspondance les principales constructions du langage algorithmique utilisé et leurs équivalents en langage C. Nous espérons que cela initiera un questionnement (qui est à la base de l'apprentissage) et développera le désir d'en savoir davantage.

Ce fascicule peut aussi servir de référence lors d'un premier apprentissage mais dans ce cas une lecture linéaire n'est pas appropriée. On trouvera donc ci-dessous quelques indications permettant un parcours progressif de son contenu, afin d'en faciliter l'abord.

* Un langage impératif est un langage fondé sur la notion de variable et l'instruction d'affectation. Une variable est une abstraction d'un emplacement mémoire de la machine de Von Neumann. L'effet désiré du programme est réalisé en modifiant dans un certain ordre la valeur des variables au moyen de l'opération d'affectation. Par exemple : Fortran, Pascal, C, Ada sont des langages impératifs.

1.2 Guide de lecture

Nous proposons ici un parcours de ce document en trois temps (initiation, consolidation, perfectionnement), parcours subjectif qui part de l'indispensable pour le débutant, pour terminer par ce qui peut apparaître comme un luxe dans un premier temps mais a vocation à devenir le quotidien du programmeur C. Nous indiquons pour chaque thème la liste des parties du document à étudier. On pourra à tout moment se référer à l'aide-mémoire rapide qui présente en fin de polycopié les équivalences avec le langage algorithmique par niveau de difficulté.

– **1er niveau : Initiation**

  Un algorithme construit pas à pas la solution d'un problème. Un programme impératif (tel qu'un programme C) traduit un algorithme : il identifie des variables (modifiables au cours des traitements) et décrit les opérations à effectuer sur ces variables ; la première chose est donc d'apprendre à déclarer des variables et à décrire des traitements en C.

  – **Déclarer des variables** : types de base, objets simples, objets composés (tableaux)
    – 2.2 Types scalaires de base
    – 2.3.1 Définition - Déclaration
    – 2.9.1 Tableaux à une dimension : Définition d'un tableau, Initialisation, Utilisation (pas la suite " Relations ... ")
    – 2.9.2 Tableaux multidimensionnels

  – **Décrire des traitements** : expressions, instructions, structures de contrôle, fonctions
    – 2.4 Opérateurs et expressions :

    - le tableau des priorités,
    - Expressions, Affectation, Expressions booléennes
  - 2.5 Instructions
  - 2.6 Structures de contrôle :
    - 2.6.1, en laissant " Cas où " pour le 2ème niveau
    - 2.6.2, en laissant " Rupture de séquence " pour le 2ème niveau
  - 2.8 Fonctions et procédures : 2.8.1 et 2.8.2 seulement

- **Présenter un programme** : syntaxe, recommandations
  - 2.1 Généralités : 2.1.1 et 2.1.2 seulement
- 5 Recommandations de style

- **Compiler et exécuter**
  - 4.2 : La chaîne de développement : 4.2.1 à 4.2.4 seulement
  - 4.3 : Les erreurs fréquentes
  - 4.4 : Résumé

- **2ème niveau : Consolidation**

  Après avoir écrit quelques programmes simples, il est possible d’aller un peu plus loin dans la connaissance du C en affinant les déclarations et la description des traitements ; il est aussi important de commencer à se familiariser avec son environnement de développement (bibliothèques standards, chaîne de développement).

  - **Déclarations plus complexes** : structures, déclarations de types, constantes, pointeurs
    - 2.9.3 Structures
    - 2.10 L’ « instruction » *typedef*
    - 2.1.3 Les constantes
    - 2.7 Pointeurs : jusqu’au paragraphe " Une illustration de ce qui précède ... " compris

  - **Traitements** : procédures, quelques schémas et expressions classiques
    - 2.8 Fonctions et procédures : 2.8.3 Procédures et 2.8.4 Résumé
    - 2.6 Structures de contrôle : 2.6.1 : " Cas où " et 2.6.2 : " Rupture de séquence "
    - 2.4 Opérateurs et expressions : Opérateur de conversion (cast), Incrémentation et décrémentations, *Sizeof*

  - **Environnement** : bibliothèques, chaîne de développement
    - 3 Les bibliothèques standards
    - 4 4.1 Le compilateur et son environnement

- **3ème niveau : Perfectionnement**

  Après une pratique minimum, il est possible d’aborder quelques points plus délicats concernant la gestion mémoire et le développement de programmes plus importants.

  - **Gestion mémoire** : allocation, stockage, pointeurs, expressions conditionnelles et composées
    - 2.3 Allocation, Objets externes et Classe de stockage : de 2.3.2 à 2.3.4 compris
    - 2.7 Pointeurs : de " *Malloc* et *free* " jusqu’à la fin
    - 2.9 Tableaux :
      - 2.9.1 : de " Relations entre ... " jusqu’à la fin
    - 2.4 Opérateurs et expressions : Expressions conditionnelles, Expressions composées

– **Développement de programmes importants** : compilation séparée, commande *make*
  – 4.2 La chaîne de développement : 4.2.5 et 4.2.6

1.3 Remerciements

Luc Litzler, dans sa version du manuel [1], faisait les remerciements suivants :

Je remercie Jean Michel Augier, Daniel Millot et Michel Maingenaud pour leurs contributions substantielles. Je suis également reconnaissant à Philippe Lalevée, Christian Parrot, Christian Schüller et Olivier Volt pour leurs relectures et leurs nombreux commentaires.

[1] Ce document a été remanié par Philippe Lalevée en 2001 et intégré dans ce polycopié en 2005 par Gil Protoy.

2 Le langage

2.1 Généralités

2.1.1 Un premier programme

```
/* afficher bonjour */  -- Commentaire
#include <stdio.h>      -- Inclusion des déclarations des fonctions offertes
                        -- par la bibliothèque standard d’entrée-sortie

main ()    -- Il doit toujours y avoir une fonction main quelque part
           -- () signifie que (dans l’exemple) elle ne reçoit pas d’arguments
{          -- Les accolades délimitent un bloc d’instructions
  printf ("bonjour\n");  -- Fonction d’affichage de la bibliothèque
                         -- \n signifie un retour à la ligne
}
```

Remarques.
– Le texte précédé de tirets ne fait pas partie du programme.
– Le langage C distingue les minuscules et les majuscules.

2.1.2 Commentaires

– Ne sous-estimez pas leur intérêt ! Ce n’est pas seulement pour les autres !
– Pour connaître les principes qui régissent l’écriture des commentaires se reporter à la section « Recommandations de style ».
– Attention, tout ce qui est entre `/*` et `*/` est ignoré :

```
/* ceci est le début d’un commentaire qui se termine mal
for (i=1; i<=N; i++) {
  printf ("N’importe quoi car cela ne s’affichera pas !\n");
}
/* et qui se poursuit jusqu’ici ! */
```

2.1.3 Constantes

Mot-clé `const`

– Toute déclaration de variable peut être précédée du mot clé `const` afin de signifier que la valeur de la variable ne doit pas être modifiée.
– Attention, le résultat d’une tentative de modification du contenu dépend des compilateurs !

```
const double Pi = 3.14;               /* variable de type double */
const char Tc [] = "elements constants";  /* tableau de caractères */
```

Constantes symboliques

– `#define NOM texte_de_remplacement`
– Toutes les occurrences de `NOM` (qui ne sont pas placées entre guillemets et qui ne font pas partie d’une désignation) sont substituées par `texte_de_remplacement` (voir la section « outils de développement »).
– `NOM` représente un symbole. Ce n’est pas une variable (`NOM` n’a pas d’adresse). On peut l’assimiler à une constante.

```
#define N 10      /* toutes les occurrences de N seront remplacées
                     par 10 */

#define N 10;     /* attention, une erreur classique consiste ici à
                     placer un ";" qui fait que toutes les
                     occurrences de N seront alors remplacées par
                     "10;" */

#define PI 3.14   /* attention, PI n’est pas une variable donc &PI
                     est une erreur */

#define T tab[10] /* le texte de remplacement n’est pas limité aux
                     nombres */
```

– Conseil : Écrivez les constantes symboliques en majuscules pour les distinguer des variables que vous écrirez en minuscules.

Constantes caractères

– `\n` : retour à la ligne (*newline*).
– `\t` : tabulation (*tab*)
– `\b` : retour en arrière (*backspace*)
– `\"` : guillemet
– `\\` : antislash (*backslash*)

Constantes chaînes de caractères

– Les chaînes de caractères sont mémorisées comme un tableau de caractères qui se termine par `\0`
– Attention, la modification d’une chaîne littérale est interdite.

Constantes énumérées

```
enum booleen {FAUX = 0, VRAI = 1}; /* associe 0 à la constante FAUX */
```

– Les constantes doivent être différentes
– Les valeurs associées aux constantes peuvent être identiques
– L’ordre d’écriture des constantes permet de les initialiser par défaut : `0, 1`...

```
enum booleen {FAUX, VRAI}; /* identique à l’exemple qui précède */
```

– Conseil : s’il ne faut retenir qu’une seule des deux formes, la première à l’avantage d’être plus générale...
– Attention : FALSE et TRUE sont déjà utilisés dans `gcc` (voir la section « Compilation et exécution »).

2.2 Types scalaires de base

– `char` : caractères représentés par leurs codes ASCII, `[-128..127]` ou `[0..255]` (selon les machines).
– `int` : entiers ; 2 ou 4 octets (selon les machines).
– `float` : nombres réels à virgule flottante en simple précision ; 4 octets.
– `double` : nombres réels à virgule flottante en double précision ; 8 octets.

– Des variantes utilisent les préfixes modificateurs : `signed`, `unsigned`, `short`, `long`.
– Le type booléen n'existe pas en C89. La valeur entière `0` correspond à faux et toute valeur entière différente de zéro à vrai (C comme ça !).

```
/* Le type Booleen peut être défini comme un ensemble de constantes
   énumérées. */
typedef enum {FAUX = 0, VRAI = 1} Booleen;
```

2.3 Définition, déclaration, allocation et classe de stockage

En dehors de la première sous-section, cette section peut être ignorée lors d'une première lecture.

2.3.1 Définition et déclaration

Une **définition** réalise une association entre un nom et un objet ainsi qu'une allocation mémoire pour cet objet. On peut initialiser les variables au moment où on les définit.

```
int x;  /* x  est un entier et correspond généralement à un mot machine */
float y = 1.5, z = 2.0; /* y contiendra une valeur approchée de 1.5 */
```

Une **déclaration** est une permission d'accès à un objet. Elle annonce les propriétés de l'objet.

```
extern int x; /* x est un entier qui a été défini ailleurs */
```

Attention. Il faut obligatoirement déclarer tous les objets (variables, etc.) avant de s'en servir...

2.3.2 Allocation des variables

L'allocation des variables met en oeuvre différents mécanismes et différentes zones mémoires. On distingue trois modes d'allocation :

Allocation statique. Elle est réalisée lors de la compilation. C'est le cas pour les variables externes (définies à l'extérieur de toute fonction). La durée de vie des variables est celle du programme. Les variables statiques sont initialisées par le compilateur à 0.

Allocation automatique. Elle est réalisée lors de l'exécution et elle utilise une zone mémoire appelée *pile* (*stack*). Les paramètres et les variables internes à une fonction sont des variables automatiques. Elles sont créées lors de l'appel et sont supprimées à la sortie de la fonction. Les variables automatiques ne sont pas initialisées par le compilateur.

Allocation dynamique. Elle est réalisée lors de l'exécution avec la fonction `malloc`. Elle utilise une zone mémoire appelée *tas* (*heap*). Les variables ont une valeur indéterminée.

Attention, n'oubliez pas d'initialiser vos variables.

2.3.3 Objets externes

Un objet est externe lorsqu'il a été défini à l'extérieur d'une fonction. Toutes les fonctions sont des objets externes. Pour qu'une variable soit externe, il faut que sa définition soit à l'extérieur de toute fonction.
Une désignation externe (*external link*) désigne toujours le même emplacement mémoire (par exemple, ce n'est pas le cas des variables locales à une fonction). Il y a une seule définition d'un objet externe.

La **portée** (*scope*) d'un objet externe va de l'endroit où il est déclaré à la fin de fichier.
Attention, pour référencer un objet externe on utilise le qualificatif : `extern`. Celui-ci est facultatif lorsque la définition de l'objet précède la référence (c'est généralement le cas car elles sont souvent en début de fichier).

```
extern int x;
extern int t []; /* la taille des tableaux est facultative dans une
                    telle déclaration */
```

Nous insistons sur le fait que la définition de l'objet n'a lieu qu'une fois (dans un second fichier par exemple) :

```
int x = 0;
int t [10];  /* la taille d'un tableau est obligatoire lors de sa
                définition */
```

Conseil. Il faut limiter strictement l'usage des *effets de bord* internes pour des raisons de lisibilité. Un effet de bord interne apparaît lorsqu'une fonction modifie la valeur d'une variable qui n'a pas été passée en paramètre (le qualificatif interne / externe situe l'objet par rapport au programme ; on parlera d'effet de bord externe lorsqu'un programme écrit dans un fichier).

```
/* effet de bord interne */
#include <sdtio.h>
void effet_de_bord (void);

int x = 0;     /*   a partir d'ici tout le monde à accès à x ! */
main () {
  printf ("%d\n", x);  /* affiche la valeur 0 */
  effet_de_bord ();   /* Attention, effet de bord en préparation */
  printf ("%d\n", x);  /* affiche la valeur 1 ! */
}

void effet_de_bord (void) {
  x++;     /* effet de bord */
}
```

2.3.4 Classe de stockage

Le concept de **classe de stockage** (*storage class*) d'un objet détermine la durée de vie de la zone mémoire qui lui est associée.

Classe statique. Le préfixage d'une déclaration par le mot clé `static` indique l'appartenance de l'objet à la classe statique. Il y a deux cas :

1. Appliquée à un *objet externe*, une déclaration de classe `static` limite la portée de l'objet à la suite du fichier source en cours de compilation. Cela permet de masquer une désignation.
2. Appliquée à des *variables internes* une déclaration de classe `static` rend permanente cette variable tout en maintenant la localité de celle-ci au corps de la fonction. Entre deux appels elle conserve sa valeur.

```
void compteur (void) {
  static int i = 0;  /* i est initialisé à 0 lors du premier appel */
  i = i+1;   /* entre deux appels la valeur de i est conservée */
  printf ("%d\n", i);
}
```

2.4 Opérateurs et expressions

Ce qui n’est pas fondamental de connaître lorsqu’on débute est en italique...

Opérateur	Symbole	Exemple	Associativité
indexation champs de structure appel de fonction	`[]` `. ->` `()`	`t[i][j]` `c.reel p->imag` `max(t)`	gauche
négation logique *négation bit à bit* incrémentation décrémentation moins unaire plus unaire conversion taille adresse indirection	`!` `~` `++` `--` `-` `+` `(type)` `sizeof` `&` `*`	`!x` `x̃` `i++ ++i` `i-- --i` `-2` `+2` `(int)` `sizeof(int) sizeof(p)` `&x` `*p`	droite
multiplication division modulo addition soustraction	`*` `/` `%` `+` `-`	`x * y` `x / y` `x % y` `x + y` `x - y`	gauche
décalage à gauche *décalage à droite*	`<<` `>>`	`x << 4` `x >> n`	gauche
opérateurs relationnels opérateur d’(in)égalité	`< <= > >=` `== !=`	`x<=0 x>y` `x==y x!=0`	gauche
et bit à bit *ou exclusif bit à bit* *ou bit à bit* et logique ou logique	`&` `^` `|` `&&` `||`	`x & b` `x ^ y` `x | y` `(x==0) && (y>0)` `(x==0) || (y>0)`	gauche
expression conditionnelle	`? :`	`(x>0)? x : -x`	droite

Opérateur	Symbole	Exemple	Associativité
affectation affectation étendue	= += -= &= >>=...	x = 1 x %= 10 y += 1	droite
composition d'expressions	,	x=1, b=x+1; z+=10	gauche

La priorité des opérateurs est décroissante par niveau et du haut vers le bas du tableau. L'associativité des opérateurs, qui indique l'ordre des opérations lorsqu'on omet les parenthèses, peut être :
– associativité gauche : `x + y + z` correspond à `((x + y) + z)`
– associativité droite : `x = y = z` correspond à `(x = (y = z))`

2.4.1 Expressions

Chaque expression possède un type et une valeur. L'ordre d'évaluation des opérandes d'une expression n'est pas spécifié dans le langage (donc il est dépendant de la machine et du compilateur utilisés).

2.4.2 Opérateur de conversion (cast)

Il force la conversion de type de la valeur d'une expression quelconque en la valeur correspondante dans le type spécifié.

```
int n = 0, m = 0;
float x = 1.0, y = 1.25;

n = (int) x;  /* n = 1 */
m = (int) y;  /* m = 1 */
```

2.4.3 Affectation

C'est un opérateur binaire. La valeur d'une expression d'affectation est la valeur affectée.

```
x = (y = 1); /* x = 1 et y = 1 */
```

Attention, l'erreur classique suivante est souvent commise :

```
if (x = 1)
  printf ("partie alors\n");  /* toujours exécutée car x = 1 retourne
                                 vrai car x a une valeur différente de 0 */
else
  printf ("partie sinon\n"); /* jamais exécutée ! */
```

Pour ce qui est des opérateurs +=, -=... : `expr1 op= expr2` équivaut à : `expr1 = expr1 op expr2`

```
i += 1;            /* équivalent à : i = i + 1 */
x *= y + 1;        /* équivalent à : x = (x)*(y+1)
                      et non pas à : x = (x*y)+1 ! */
f((i+=1)+(i+=3));  /* Attention, si i vaut 4, l'expression peut
                      valoir f(13) ou f(15)
```

Conseil : il faut privilégier la lisibilité à l'économie de quelques caractères ! Il est fortement déconseillé de faire des affectations dans les conditions ou dans les expressions.

2.4.4 Incrémentations et décrémentations

Un exemple vaut mieux qu'un long discours :

```
int i=0, j=0, x=0, y=0;
i++;       /* incrémente i : i vaut 1 */
++j;       /* incrémente j : j vaut 1 */
/* maintenant, il faut faire attention à la position de ++... */
x = i++;  /* x vaut 1 et i vaut 2 : on fait l'affectation x = i
             puis on incrémente i */
y = ++j;  /* y vaut 2 et j vaut 2 : on incrémente j de 1
             puis on réalise l'affectation x = j */
/* même chose avec -- */
```

Conseil : en résumé, ++ c'est comme le bon vin il ne faut pas trop en abuser (ou même l'éviter si on veut conduire en toute sécurité).

2.4.5 Expressions booléennes

Elles ont pour valeur 0 si elles sont fausses, 1 si elles sont vraies.

```
! (i < N)   /* non (i inférieur strict à N) : i >= N */
(i < N) && (t[i] != 0)   /* i inférieur strict à N et t[i] différent de 0 */
(i >= N) || (t[i] == 0)  /* i supérieur ou égal à N ou t[i] égal à 0
*/
```

2.4.6 Expressions conditionnelles

```
(expr1) ? expr2 : expr3  /* expr2 si expr1 est vraie et expr3 sinon */
z = (y != 0) ? x / y : x / (y+1) /* si y différent de 0 alors z = x/y
                                    sinon z = x/(y+1) */
```

Conseil : nous vous déconseillons l'usage des expressions conditionnelles pour les raisons de lisibilité.

2.4.7 l'opérateur `sizeof`

Il permet de connaître la taille (nombre d'octets) d'un objet particulier ou d'un type.

```
/* sizeof (nom_type) : attention aux parenthèses */
/* sizeof nom_objet */
int taille;
taille = sizeof (int);
if (sizeof taille == taille) printf("normal");
else printf("ca alors !");
```

2.4.8 Expressions composées

L'opérateur `,` permet de composer des expressions. L'évaluation de la dernière expression donne le type et la valeur de l'expression composée.

```
x = (y = 4, z = 2 * y);  /* y = 4 puis z = 8 et enfin x = 8 */
```

2.5 Instructions

2.5.1 Le point-virgule

Une instruction est une expression bien formée suivie par un point-virgule. En C, le ’ ;’ est un marqueur de fin d’instruction élémentaire. En Pascal le ’ ;’ n’a pas le même sens : c’est un séparateur d’instructions.

```
if (x == 0)
  x = 1;  /* le ’;’ termine l’instruction d’affectation  */
else
  /* ... */
```

2.5.2 Blocs d’instructions

Une instruction est soit simple ou élémentaire (terminée par un ;) soit composée, c’est à dire une liste d’instructions (simples ou composées) encadrée d’accolades.

```
{   /* une accolade ouvrante marque le début d’un bloc */
    /* déclarations des variables locales au bloc */
  int i;
  /* Instructions du bloc */
  if (i == 0) {
    int k ;   /* déclaration locale de k à la partie "alors" du si */
    for (k=0; k<= N; k++) {
      /* ... */
    }
  }
}   /* une accolade fermante marque la fin d’un bloc */
```

– La portée des variables définies en début de bloc est délimitée par l’accolade fermante correspondante.
– Conseil : nous déconseillons la déclaration de variable en milieu de bloc pour des raisons de lisibilité.
– Attention : éviter les constructions qui masquent les variables.

```
main () {
  int i = 0;                /* i de la fonction main */
  printf ("i = %d\n", i);   /* affiche i = 0 */
  if (i == 0) {
    int i = 10;             /* i du bloc "alors" */
    printf ("i = %d\n", i); /* affiche i = 10 */
    i = i + 1;              /* i = 11 */
  }
  printf ("i = %d\n", i);   /* affiche i = 0 */
}
```

2.6 Structures de contrôles

2.6.1 Schémas de choix

Si alors sinon. Les instructions sont simples ou composées. Attention : les parenthèses sont nécessaires et il n’y a pas de mot clé *then*.

```
if (expr)
  instr1
else
  instr2
```

– La partie `else` est facultative.
– Imbrication de si alors sinon :

```
if (expr1)
  instr1
else if (expr2)
  instr2
else if (expr3)
  instr3
...
else
  instrn
```

– Conseil : adoptez ce style d'écriture ! Il permet de rester lisible tout en se décalant pas trop rapidement vers la droite.
– Attention : chaque partie `else` est associée au *if* le plus proche !
– Quelques erreurs (fréquentes) :

```
if (x = 1)   /* Attention, il faut écrire if (x == 1) */
  y = 0;
else
  y = 1;
/* c'est bizarre : y est toujours égal à 0 ! */

if (x == 1)
  y = 0      /* Attention, il faut écrire y = 0;  */
else
  y = 1;

if (n > 0)
  if (a > b) printf ("n > 0 et a > b\n");
else
  printf ("n <= 0\n"); /* Erreur, l'indentation ne trompera pas le
                          compilateur : quand n <= 0 rien ne s'écrit ! */
```

Cas où. Il permet de réaliser une sélection en fonction de la valeur d'une expression de contrôle.

```
switch (expr) {    /* Attention : les parenthèses sont nécessaires */
  case expr_cste1: instr1  /* exécutée si expr vaut expr_cste1 */
  case expr_cste2: instr2  /* exécutée si expr vaut expr_cste2 */
  ...
  case expr_csten: instrn  /* exécutée si expr vaut expr_csten-1 */
  default : instrd         /* optionnelle mais conseillée */
}
```

– Les cas peuvent être regroupés.
– L'instruction `break` permet de sortir de l'instruction `switch` :

```
/* compter les espaces (blancs et tabulations), les lignes et les
   autres caractères */
switch (c) {      /* c est un caractère */
  case ' ':       /* si c'est un blanc, */
  case '\t': nb_espaces++;
             break;    /* ou une tabulation, incrémenter nb_espaces */
  case '\n': nb_lignes++; break;  /* si c'est \n, incrémenter nb_lignes */
  default: nb_autres++; break;    /* sinon incrémenter nb_autres */
```

```
  }
```

2.6.2 Schémas itératifs

Voici trois façons de réaliser une boucle :

```
/* Trois manières différentes d’afficher la séquence : 1, 2, ..., 100. */
#define N 100
/* ... */
i = 1;
while (i <= N) {   /* le test est réalisé en début de boucle */
   /* le corps de la boucle n’est pas exécuté si i > N */

  printf ("%d\n", i);
  i++;    /* i = i + 1 */
}
/* en sortie, i > N */

for (i=1; i<=N; i++) { /* equivaut au while ci-dessus en plus concis */
  printf ("%d\n", i);
}

i = 1;
do {     /* Attention, quelque soit les valeurs de i et de n
            le corps de la boucle est exécuté au moins une fois */
  printf ("%d\n, i);
  i++;
} while (i <= N);   /* le test est réalisé en fin de boucle */
/* en sortie, i > N */
```

Conseil : dans la mesure du possible, évitez les conditions basées sur une égalité car elles sont moins robustes (elles conduisent plus facilement à des boucles infinies s’il y a une erreur de programmation) que celles qui utilisent des inégalités. Par exemple, écrivez `i < N` plutôt que `i != N`.

Concernant les rupture de séquence :
– `break` : permet de sortir de l’instruction `switch`, `while`, `do`, `for` la plus intérieure.
– `continue` : permet de réaliser un branchement au pas suivant de la boucle.
– `goto` : on n’en parlera pas. c’est inutile, tous les algorithmes peuvent être définis sans !

```
  /* Imprimer les éléments strictement positifs d’un tableau jusqu’à
     rencontrer la fin ou 0 */
  #define N 10
  int t[N]; /* déclaration de t */
  /* initialisation de t */
  for (i=0; i<N; i++) {
    if (t[i] < 0) continue; /* si t[i] < 0, on passe à l’élément suivant */
    if (t[i] == 0) break;   /* si t[i] = 0, on sort de l’instruction for */
     printf ("%d\n", t[i]); /* sinon (t[i] > 0), on imprime t[i] */
  }
/* ici, i = N ou (i < N et t[i] = 0) */
```

Conseil : l’usage de ces instructions doit être strictement limité car il conduit généralement à un texte difficile à comprendre.

```
/* le même programme sans continue, ni break... */
i = 0;
while (i < N && t[i] != 0) {
  if (t[i] > 0) printf ("%d\n", t[i]);
  i++;
}
```

Attention : en algorithmique, le *ou* et le *et* booléens sont des opérateurs commutatifs. En C, le *ou* et le *et* booléens sont des opérateurs non commutatifs. Dans le code ci-dessus, la condition `(i<N && t[i]!=0)` devient fausse dès que la première condition `i<N` est fausse, c'est à dire `i>=N` ; par suite, la seconde condition `t[i]!=0` n'est pas évaluée avec `i>=N` (si ce n'était pas le cas, cela provoquerait une erreur d'exécution).

2.7 Pointeurs

La signification d'une variable peut s'interpréter selon deux points de vue différents, selon que l'on parle de son adresse mémoire ou de sa valeur :

```
int i = 5;
i = i + 1;
```

Dans une instruction, si une variable `i` est citée à gauche d'un symbole `=` (le langage C utilise le terme *left-value*), ce `i` désigne l'adresse mémoire où est stockée la valeur de `i` ; si cette variable est citée à droite d'un symbole `=`, ce `i` désigne la valeur de cette variable, `5` dans l'exemple. La difficulté consiste à repérer le statut d'une variable quand elle ne figure pas dans une affectation ou une initialisation ; par exemple, dans l'expression `++i`, `i` désigne une adresse.

2.7.1 Accès à l'adresse mémoire d'une variable

L'opérateur `&` permet d'accéder à l'adresse d'une variable.

```
int i = 0;
/* &i désigne l'adresse de i */
```

2.7.2 Notion de pointeur

Un pointeur est une **variable** qui contient l'adresse d'un objet (entier, `float`, ...). Un pointeur a donc une adresse (c'est une variable). Il se déclare de la manière suivante :

```
int * pi;  /* pi est un pointeur d'entiers;
              * n'est pas l'opérateur de déréférence ! */
```

2.7.3 Accès à la valeur d'un objet dont l'adresse est dans un pointeur

L'opérateur de déréférencement `*` permet d'accéder à la valeur d'un objet dont on connaît l'adresse.

```
*pi = 2;  /* l'entier désigné par *pi reçoit la valeur 2 */
```

La valeur `NULL` est une constante symbolique correspondant à une valeur non définie pour les pointeurs ; on dit qu'un pointeur à `NULL` ne pointe sur rien. Elle est définie dans `<stdio.h>` (voir la section « bibliothèque standard ») dont la valeur vaut `0`. L'utilisation d'un pointeur valant `0` provoquera l'interruption immédiate du programme avec un message d'erreur ressemblant à celui-ci : *Segmentation fault*

2.7.4 Opérations sur les pointeurs

Les pointeurs étant des variables les opérateurs suivants : `*`, `=`, `==`, `!=`, `+`, `-` s'appliquent. Voici une illustration de ce qui précède :

```
int * p1, * p2;  /* p1 et p2 sont deux pointeurs d'entiers */
int i = 5, j = 0;

p1 = &i;  /* p1 = adresse de la variable i */
p2 = &j;  /* p2 = adresse de la variable j */

p2 = p1;    /* p2 = adresse de la variable i  */
j = *(&i);  /* identique à : j = i */
*(p2) = 0;  /* identique à : i = 0 */
```

2.7.5 `malloc` et `free`

La fonction `malloc` permet d'allouer dynamiquement de la mémoire (dans le tas). Elle prend en entrée le nombre d'octets nécessaire et elle retourne un pointeur sur la zone mémoire allouée. La fonction `free` permet de redonner au système une zone mémoire obtenue par appel de `malloc`. Elle prend en entrée un pointeur sur la zone que l'on veut libérer (elle est sans action sur la valeur du pointeur).

```
struct point {
  float x;
  float y;
} p;
struct point * pp1;    /* pointeur de structure point. */
struct point * pp2;    /* pointeur de structure point. */

/* Allocation : 2 variantes liées à sizeof  */
pp1 = malloc (sizeof (struct point)) ;
pp2 = malloc (sizeof p);

free (pp1);  /* libère la zone mémoire pointée par pp1. Attention,
                désormais pp1 pointe sur une zone non accessible */
pp1 = NULL;  /* voir le conseil si dessous */
```

Conseil : après l'instruction `free` (`pp1`), il est conseillé de faire une affectation `pp1 = NULL ;` de manière à pouvoir tester plus tard dans le programme l'existence de l'objet pointé.

2.7.6 Pointeurs et tableaux

L'opérateur `[]` est utilisable aussi avec les pointeurs, ce qui donne une équivalence avec les tableaux, **uniquement** pour ce qui concerne l'accès mémoire.

```
int t[100];  /* t désigne l'adresse du premier élément du tableau
                c'est à dire &t[0] */
/* t est une constante de pointeur sur un entier */
/* Attention, la valeur associée à t ne peut pas être modifiée */
```

2.7.7 Pointeurs et fonctions

Les fonctions possèdent également une adresse qui peut être stockée dans un pointeur. Avec ce pointeur, la fonction dont l'adresse est contenue dans le pointeur pourra être appelée.

```
float carre (float x) {
  /* carre est une constante de pointeur sur sa première instruction */
  /* Attention, la valeur associée à carre ne peut pas être modifiée */
  return x*x;
}

float integrale (float a, float b, float (*pf) (float)) {
  /* ... */
}

/* exemple d'utilisation */
float (*ptrf)(float);
ptrf = carre;
y = ptrf(5.2);
printf ("%f\n", integrale (0.5, 5.0, carre));
```

2.7.8 Pointeurs, tableaux et chaînes de caractères

Les pointeurs sont utilisés très souvent pour parcourir les tableaux, en particulier les chaînes de caractères.

```
char * pc = "abc";
char tc [] = {'a', 'b', 'c', '\0'}
```

Attention : la valeur de `pc` peut être modifiée car c'est une variable (pointeur). Par contre, `*pc` qui est une constante de pointeur sur la chaîne `abc\0` ne peut pas être modifiée. D'autre part, `tc` n'est pas modifiable (c'est une constante). On peut changer les valeurs des éléments. On ne peut pas changer le nombre d'éléments (la taille de `tc` est déterminée par le compilateur).
Voici un exemple de copie d'une chaîne dans une autre (Attention, comme `s` et `t` sont des variables pointeurs, écrire `s=t ;` revient à écrire quelque chose du genre : `200=100!`).

```
/* strcpy : copie s (source) dans t (target).
   Nécessite que:  longueur (s) <= longueur (t) */
void strcpy (char * t, char * s) {
  while ( *s != '\0') {
    *t = *s;
    s++;
    t++;
  }
}
```

2.8 Fonctions et procédures

2.8.1 Le passage de paramètres en C

C'est le point essentiel à comprendre : au moment de l'appel (de la fonction ou de la procédure) il y a :
– création des variables automatiques dans la pile (paramètres formels, variables locales ne faisant pas partie de la classe `static`) ;

– évaluation des paramètres effectifs (constantes, variables, expressions) et affectation des valeurs aux variables créées dans la pile.

La transmission de l'appelant à l'appelé est toujours réalisée de cette façon. C'est la nature de l'information (adresse ou valeur) qui diffère et qui conditionne le fait que nous parlerons de passage par valeur ou par adresse. Attention, l'ordre d'évaluation des paramètres n'est pas normalisé. Il ne faut pas utiliser d'opérateurs à effet de bord comme +=, -=... lors de l'appel d'une fonction.

2.8.2 Fonctions

Conceptuellement, une fonction algorithmique est une abstraction destinée à calculer une valeur unique à partir d'un ensemble de données fournies en entrée (les arguments sont donc laissés intacts après un appel). Une fonction C doit retourner une valeur mais elle peut modifier ses paramètres effectifs. Par exemple, la fonction `scanf` retourne un entier (le nombre de données correctement affectées ou `EOF` en fin de fichier), a un argument en entrée (le format) et un nombre variable d'arguments en sortie (les données lues).

```
/* Elevation au carré d'un nombre */
#include <stdio.h>

  float carre (float);
  /*
     Cela s'appelle un prototype.
     Le compilateur à besoin de connaître :
       - le type de la valeur retournée ;
       - le nombre et le type des paramètres.
   Ne pas oublier le ";".
   Les noms des paramètres sont des commentaires pour le compilateur
   mais peuvent donner des indications sur leur usage.
   Définir la fonction à cet endroit marche mais ce n'est pas l'usage... */

  main () {
    float x, y;
    scanf ("%f", &x);  /* lire x */
    y = carre (x+1);  /* appel de la fonction carre () */
    printf ("\n%f\t%f\n", x, y); /* nouvelle ligne, x, tabulation, (x+1)2, nouvelle ligr
}

  /* définition de la fonction carre(z) */
  float carre (float z) {  /* il est possible d'écrire x au lieu de z */
    return z * z;          /* return permet de retourner la valeur z2 */
  }
```

L'appel de `carre` avec comme paramètre `(x+1)` et `x = 1` se traduit par :

– création automatique d'une variable z dans la pile ;
– évaluation de x+1 ;
– affectation z = 2 ;
– évaluation de z*z en 4 ;
– retour de 4 à l'appelant (affectation y = 4).

La valeur retournée est un type arithmétique, une structure, une union, un pointeur ou `void` (rien). Dans le cas de la fonction `main`, elle retourne un entier qui indique à l'environnement comment s'est terminé le programme (`0` indique une terminaison normale). Il peut y avoir plusieurs instructions `return` dans le corps d'une fonction (une seule sera exécutée !).

2.8.3 Procédures

Conceptuellement, une procédure algorithmique est une abstraction qui représente une action sur des données. Elle peut utiliser des paramètres en donnée seulement, en donnée et en résultat ou en résultat seulement. Une procédure C est une fonction C qui ne retourne rien (type `void`) et qui possède des paramètres en donnée (passés par valeur), en donnée et résultat ou en résultat seulement (passés par adresse).

```
/* échange du contenu de deux variables entières */
#include <stdio.h>

void echangeRien (int, int );
void echangeVraiment (int *, int *);

main () {
  int i = 0, j = 1;

  echangeRien (i, j);        /* x = i, y =j  */
  printf ("%d%d\n", i, j);    /* i = 0, j = 1 */
  echangeVraiment (&i, &j);  /* px = &i, py = &j */
  printf ("%d%d\n", i, j);    /* i = 1, j = 0 */
}

void echangeRien (int x, int y) {
  int a;  /* variable auxiliaire, locale à la fonction echangeRien */

  /* Echange des contenus de x et de y */
  a = y;
  y = x;
  x = a;
  /* ... mais les variables x, y sont distinctes de i, j !*/
}

void echangeVraiment (int * px, int * py) {
  int a;  /* variable auxiliaire, locale à la fonction echangeVraiment */

  a = *py;
  *py = *px;
  *px = a;
  /* on a échangé les contenus des objets (entiers) situés
     aux adresses &i et &j */
}
```

2.8.4 Résumé

Si le paramètre est en donnée, on utilise une variable scalaire : `char`, `int`, `float`... Si le paramètre est en donnée ou en donnée et résultat, il faut donner l'adresse de l'objet comme argument d'appel et il faut utiliser un pointeur en paramètre. La procédure doit connaître l'adresse de l'objet où elle doit mettre une valeur !

```
int x;
int t[10];
...
scanf ("%d", &x);  /* et non pas  scanf ("%d",  x); ! */
scanf ("%d", &t[i]);  /* et non pas scanf ("%d", t[i]); ! */
```

2.9 Structures de données

2.9.1 Tableaux à une dimension

Définition d'un tableau.

```
int t[N]; /* tableau de N entiers */
```

Attention, `N` doit être une constante (le compilateur à besoin de connaître le nombre d'octets consécutifs qui constituent le tableau ; les indices vont de `0` à `N-1`.

Initialisation.

```
int t1 [3] = {-1, 0, 1};  /* tableau de 3 entiers */
int t2 [] = {-1, 0, 1};   /* idem, mais on a laissé compter le compilateur */

char texte1 [] = "bonjour"; /* tableau de 8 caractères ; t[7] = '/0' */
char texte2 [] = {'b', 'o', 'n', 'j','o', 'u', 'r', '\0');
```

Conseil : laissez compter le compilateur à votre place.

Utilisation. `t[i-1]` accède au i-ème élément (les tableaux commencent à l'indice `0`!). Attention, il n'existe pas d'opérateur d'affectation et de comparaison.

Relations entre les pointeurs et les tableaux. Un nom de tableau t apparaissant dans une expression (sauf si ce nom est un argument d'une procédure) est converti en un pointeur (qui n'est pas une variable) vers le premier élément du tableau (ie qui a la valeur `&t[0]`). Ainsi, l'expression `t[i]` est l'abréviation de `*(t + i)`. La valeur de `*(t + i)` équivaut à `*(&t[0] + i)`. En fait toute expression `expr1 [expr2]` est remplacée par `*((expr1) + expr2)`.
Remarque : le compilateur doit tenir compte de la taille des objets en mémoire. Par exemple, si les entiers sont représentés sur 16 bits, l'obtention de l'élément i nécessite une incrémentation de 2*i octets par rapport à l'adresse de base du tableau. `t` n'est pas converti en pointeur si `t` est un argument de `sizeof` ou quand `t` est opérande de `&`. Voici quelques exemples :

```
main () {
  int t[] = {-1, 0, 1};
  int *pi = NULL;

  pi = t;    /* équivaut à pi = &t[0] */
  printf ("t[0] = %d\n", t[0]);  /* affiche t[0] = -1 */
  printf ("t[0] = %d\n", *pi); /* affiche t[0] = -1 */
  pi++;
  printf ("t[1] = %d\n", *pi);  /* affiche t[1] = 0 */
  /* Attention, t++ est une erreur (t n'est pas une variable) */
}
```

Attention, quelque chose comme `if (t1 == t2)` est valide, mais cela ne teste pas l'égalité de deux tableaux.

Paramètres de type tableau. L'expression `f(t)` est convertie en `f(&t[0])`. Les déclarations de tableaux en tant qu'arguments formel déclarent un pointeur de même type que celui qui résulte de la conversion du nom de tableau constituant l'argument effectif. L'exemple ci-dessous illustre cet aspect :

```
/* Lecture et somme des éléments d'un tableau */
#include <stdio.h>
#define N 10
void lire_tab (int t[], int n);
int somme1 (int t[], int n);
int somme2 (int * t , int n);

main () {
  int a[N];
  lire_tab (a, N);  /* c'est l'adresse de a[0] qui est passée en paramètre */
  printf ("Somme : %d\t%d\n", somme1 (a, N), somme2 (a, N));
}

/* Lecture de n entiers */
void lire_tab (int t[], int n) {
  int i;
  for (i=0; i<n; i++) {
    printf ("\nEntrer t[%d] : ", i);
    scanf ("%d", &t[i]); /* Attention, adresse de t[i] */
  }
}

/* somme des n premiers éléments de t (première version) */
int somme1 (int t[], int n) {
  int i = 0, s = 0;
  while (i < n) {
    s = s + t[i];
    i++;
  }
  return s;
}

/* somme des n premiers éléments de t (seconde version) */
int somme2 (int * t , int n) {
  int i = 0, s = 0;
  while (i < n) {
    s = s + t[i] ; /* équivalent à : s = s + *(t+i); */
    i++;
  }
  return s;
}
```

2.9.2 Tableaux multidimensionnels

Le mécanisme de tableau est répété pour chaque dimension : un tableau à deux dimensions est un tableau de tableau.

```
int matrice [2][3] = {
  {1, 2, 3},    /* on vous conseille cette présentation... */
```

```
  {4, 5, 6}
};
```

L’accès à un élément est effectué comme ceci : `t[ligne][colonne]`. Attention, `t[ligne, colonne]` n’est pas correct. De plus, la constante qui indique la taille de seconde dimension est obligatoire quelque soit le contexte. Un tableau à deux dimensions est un tableau à une dimension dont chaque élément est un tableau.

2.9.3 Structures

```
struct point {  /* point est optionnel mais utile... */
float x;          /* premier champ (membre) de la structure point */
float y;
} p1, p2, p3;     /* p1, ...p3 est optionnel si l’on a struct point */

struct point p4;  /* ... d’où l’utilité de "point" */

p1.x              /* référence au membre x */

struct point milieu (struct point p1, struct point p2) {
                  /* c’est une fonction... */
  struct point m;

  m.x = (p1.x + p2.x) / 2;
  m.y = (p1.y + p2.y) / 2;
  return m;    /* on retourne un objet de type : struct point */
}
```

Attention, ne pas oublier le " ;" qui termine la définition d’une structure.

Pointeur sur une structure.

```
void point_init (struct point * pp) {
  (*pp).x = 0;  /* Attention, *pp.x est évalué comme *(pp.x)
                   car . a la priorité sur *  */
  pp->y = 0;    /* accès au champ y plus facile à retenir */
}

main () {
  struct point p1;
  /* ... */
  point_init (&p1);
  /* ...*/
}
```

Tableaux de structures.

```
/* Définition et initialisation d’un tableau de struct point. */

struct point t1[] = {{0, 1}, {2, 3}, {4, 5}}; /* initialisation standard */

struct point t2[] = {0, 1, 2, 3, 4, 5}; /* autre initialisation possible */
```

```
struct point t3[3];
t3[0].x = 0;
t3[0].y = 1;  /* champ (membre) y de t3[0] */
t3[1].x = 2;
t3[2].y = 3;
```

Le nombre d’éléments est calculé si les valeurs initiales sont présentes et si on a écrit ’[]’.

2.10 Définition de types

La définition de type sert à paramétrer les programmes et permet de définir des nouveaux types en renommant un type existant ou en nommant un type construit.

```
typedef int Longueur ;   /* renommage de int  par Longueur */
Longueur l;              /* équivalent à : int l; */

struct point {    /* définition de struct point */
  float x;
  float y;
};
typedef struct point Point ;   /* renommage de struct point par Point */
typedef struct point point;    /* c’est possible aussi... */
Point p1, p2;                  /* équivalent à struct point p1, p2; */

#define N 10
typedef int Tableau [N];       /* renommage de int[N] par Tableau */
Tableau t;                     /* équivalent à int t [N]; */

typedef struct point * Ppoint; /* renommage de struct point * par Ppoint */
typedef Point * Ptr_point;     /* identique (si Point a déja été défini) */
```

Le mécanisme de substitution (qui est réalisé par le compilateur) est plus puissant que celui mis en œuvre par le préprocesseur (`#define`). Cela peut même devenir assez compliqué à comprendre :

```
typedef int (* PFI) (char *,  int);
```

désigne `PFI` comme un pointeur de fonction ayant un pointeur de caractères et un entier en paramètre et qui retourne un entier.
Pour améliorer la lisibilité des programmes utilisez des noms de type commençant par une majuscule. Un truc peut vous permettre de vérifier la syntaxe de votre `typedef`, il vous suffit d’enlever le mot `typedef` et de vérifier qu’il s’agit bien d’une déclaration légale d’une variable dont le type est celui que vous voulez renommer.

3 Les bibliothèques standards

Le langage C a été normalisé avec un certain nombre de fonctions standards regroupées dans des bibliothèques. Une aide en ligne existe pour ces fonctions : `man <nom de la fonction>`.

3.1 Les entrées-sorties

Les déclarations des fonctions d'entrées-sorties standards sont regroupées dans le fichier `stdio.h` ; tout fichier doit donc comporter (`#include <stdio.h>` si une telle fonction y est utilisée.

Lecture entrée standard.

```
int scanf (char * format, ...)
scanf ("%d%d", &x, &y);
```

Le premier argument appelé `format` décrit ce qui doit être lu. Les espaces et tabulations sont ignorés dans le format (quand on lit des entiers). Les spécifications de conversion les plus courantes sont : `%d` pour un `int` en base 10, `%f` pour un `float` et `%c` pour un caractère. Les arguments sont des pointeurs et les paramètres doivent être des adresses (paramètres en résultat).

Écriture sortie standard.

```
int printf (char * format, ...);
printf ("xxx\nab"); printf("c\n");

int x = 10;
printf ("Le stock est de %d unites", x); => Le stock est de 10 unites
```

Écriture mémoire dans une chaîne de caractère.

```
int sprintf (char * s, const char * format, ...)
```

`sprintf` est similaire à `printf`, mis à part le fait que le résultat est écrit dans une chaîne de caractères `s` terminée par `'\0'`. Attention, `s` doit être assez grande pour recevoir le résultat. `sprintf` retourne le nombre de caractères écrits (`'\0'` non compris).

3.2 Les fonctions de tests de catégories de caractères

Les fonctions de tests de catégories de caractères sont déclarées dans le fichier `ctype.h`. `isalpha (c)` pour les lettres, `isdigit (c)` pour les chiffres en sont des exemples.

3.3 Les fonctions mathématiques

Les fonctions mathématiques sont déclarées dans le fichier `math.h`. Par exemple, `ceil (x)` retourne la partie entière supérieure de `x`, `floor (x)` retourne la partie entière inférieure de `x` et `fabs (x)` retourne la valeur absolue de `x` sans oublier `exp`, `log`, `pow`, `sin`, `cos`, `tan`, `atan`, ...

3.4 Les utilitaires

Les fonctions utilitaires sont déclarées dans le fichier `stdlib.h`. Par exemple, `int atoi (const char *s )` convertit `s` en un entier, `void * malloc (size_t size)` et `void free (void * p)` que nous avons vue plus haut.

4 Outils de développement

4.1 Le compilateur et son environnement

Avant de conduire (l'ordinateur), il vaut mieux avoir quelques notions de *mécanique*. `gcc` est un ensemble d'outils développés dans le cadre du projet GNU. Vous pouvez les visualiser avec l'option `-v` (*verbose*) :

```
gcc -v fich.c
```

4.1.1 Le préprocesseur

Le préprocesseur (*preprocessor*) a pour rôle de :
– produire ce que sera le texte d'entrée du compilateur. Les directives le concernant sont signalées par `#`.
– définir des macro-définitions :
  – `#define N 10` : définition d'une constante symbolique. La fin de ligne marque la fin du texte de remplacement ; ; pour prolonger la définition sur plusieurs lignes, il faut placer \ en fin de chaque ligne. Comme indiqué dans les recommandations de style, les majuscules sont vivement conseillées.
  – `#include <stdio.h>` : inclusion de fichiers d'en-tête (définitions et déclarations des variables communes)
  – Compilation conditionnelle avec `#ifdef`, `#else`, `#endif`...

Lorsque le nom d'un fichier d'en-tête est entre guillemets, la recherche du fichier commence en général dans le répertoire où on se trouve quand on lance la commande `gcc`, puis selon une règle définie lors de l'implantation. L'option `-I` permet de spécifier un répertoire. Un fichier d'inclusion peut à son tour contenir des directives `#include`.
Pour visualiser le travail du pré-compilateur, vous pouvez taper :

```
gcc -E fich.c
```

Lorsque votre application informatique met en œuvre plusieurs fichiers (voir la section « compilation séparée »), nous placerons dans un fichier d'en-tête (*header file*) suffixé par `.h` (par exemple `global.h`) toutes les informations communes : inclusions de bibliothèques, définitions de pseudo-constantes, prototypes... Dans ce cadre chaque module (`.c`) débute par une déclaration d'inclusion du type : `#include "global.h"`. Pour éviter d'inclure plusieurs fois les même informations il est possible de réaliser une inclusion conditionnelle. A la vue de l'instruction `#ifndef NOM` le préprocesseur réalise l'inclusion s'il ne connaît pas `NOM`.

```
/* contenu type d'un fichier d'en-tête */
/* inclusion conditionnelle : si _ENTETE_H est connue du préprocesseur,
   il ne réalise pas l'inclusion */
#ifndef _ENTETE_H
#define _ENTETE_H 1  /* définition de la pseudo constante _ENTETE_H
                  lors d'autres inclusions, le test précédent sera faux */

/* inclusion de bibliothèques */
#include <stdio.h>

/* pseudo constantes */
#define N 10

/* définitions de type */
typedef enum {FAUX, VRAI} Booleen;

/* prototypes */
void afficher_tab (int t[]);

#endif    /* Marqueur de fin d'inclusion conditionnelle. */
```

4.1.2 Le compilateur

Le compilateur (*compiler*) analyse le texte produit par le préprocesseur (analyse lexicale, analyse syntaxique, contrôle de type). Il produit un programme cible en langage d'assemblage. Avec `gcc`, le fichier produit (extension `.s`) peut être visualisé avec l'option `-S` :

```
gcc -S fichier.c
```

4.1.3 L'assembleur

L'assembleur (*Assembler*) traduit le langage s'assemblage très proche du processeur de la machine en binaire ; c'est en fait un compilateur *simple* qui produit un fichier binaire translatable (car on ne connaît pas encore l'adresse d'implantation du programme en mémoire). Le code des fonctions en bibliothèque utilisées dans le programme sera incorporé plus tard.

4.1.4 L'éditeur de liens et le chargeur

Le chargeur (*loader*) et le relieur (*linker*) assure le chargement et la reliure (édition de liens). Le chargement consiste à prendre du code machine translatable et à placer les instructions et les données en mémoire. Le relieur permet de constituer un programme unique à partir de plusieurs fichiers contenant du code machine translatable (modules de compilation séparés, routines de bibliothèque). Il doit en particulier résoudre les références externes qui apparaissent quand le code d'un fichier référence un objet défini dans un autre fichier.
Il y a deux types d'éditions de liens, selon le type de bibliothèque utilisée :

Statique Il y a extraction du code des fonctions puis recopie dans le fichier binaire exécutable. Le chargement en mémoire est direct. C'est simple et les binaires sont autonomes...

Dynamique Les bibliothèques sont des objets partagés. Ce mode autorise une prise en compte transparente des modifications des fonctions en bibliothèque. Il augmente le temps de chargement et de l'exécution, car l'exécutable est « incomplet ».

Dans les deux cas, c'est l'éditeur de liens qui produit l'exécutable après compilation. Son travail doit être complété lors du chargement lorsque des bibliothèques dynamiques sont utilisées.

Exemple :

```
gcc fich.c -o fich -lm
```

Si l'on a `#include <math.h>`, l'édition de lien se fait avec `-lm`.

4.2 Chaîne de développement

Programmer ce n'est pas seulement écrire des programmes. La programmation recouvre l'ensemble des activités qui fait passer d'un *problème* à un *programme* supposé représenter une solution informatique au problème. La conception de l'algorithme est la phase centrale de ce processus.

4.2.1 Conception de l'algorithme

Il ne faut pas se laisser aveugler par l'objectif final : le codage ! Il faut rester le plus abstrait possible. Pour désigner les objets de votre algorithme, reportez vous à la section *Recommandations de style*. Il faut aussi se détacher des contraintes de tel ou tel langage de programmation afin de faire la différence entre les contraintes propres à un langage (richesse d'expression...) et les difficultés inhérentes à un problème donné, choisir la méthode **la plus claire et la plus concise** pour résoudre le problème. En général un texte en pseudo-code est la meilleure solution pour exprimer un algorithme.

4.2.2 Édition du programme

Vous devez avoir un algorithme validé qui répond au problème posé sous les yeux. Si ce n'est pas le cas retournez à la case départ.

Choix de l'éditeur de texte. `Xemacs` (ou `emacs`) est l'éditeur que nous recommandons : il est répandu et puissant. Certaines versions d'`emacs` peuvent être configurées pour C avec un fichier `.emacs` dans votre répertoire de connexion. On dispose alors de fonctions syntaxiques : mots clés en couleur, contrôle de l'indentation et des accolades... Pour que celles-ci soient actives, il est nécessaire d'indenter les textes des programmes (touche tabulation : `->|`). La barre de menu permet d'accéder aux principales commandes. Il existe des raccourcis clavier (`C-` signifie que l'on doit maintenir la touche `Control` enfoncée et appuyer simultanément sur le caractère qui suit) :

– `C-x C-s` : sauvegarder le tampon dans un fichier
– `C-x C-w` : sauvegarder le tampon sous un nom qui sera précisé dans la ligne de commandes.
– `C-x C-c` : quitter `Emacs` (sans sauvegarder).
– `C-p` (resp. `C-n`) : ligne précédente (resp. suivante).
– `C-b` (resp. `C-f`) : un caractère à gauche (resp. à droite).
– `C-g` : annule la commande en cours.

Style. Reportez vous à la section *Recommandations de style*.

– Indentez : beaucoup d'erreurs seront ainsi évitées.
– Commentez : ou on en reparlera dans six mois.

4.2.3 Compilation et exécution

```
gcc -ansi -pedantic fich.c
gcc -ansi -pedantic fich.c -o fich
```

La première stocke les instructions exécutables dans le fichier `a.out`. Le nom de ce fichier est fixé par l'option `-o` dans la deuxième forme.

Conseil : utilisez les options `-ANSI` et `-pedantic` pour produire du C ansi. La traque aux bogues peut commencer... Pour progresser, interdisez vous toute erreur conceptuelle (algorithmique). Les seules erreurs acceptables à ce niveau sont les erreurs liées à la traduction de votre algorithme en C.

4.2.4 Résumé

Pour la conception de l'algorithme, il faut quelques feuilles de papier, un crayon et une gomme... Pour l'édition du programme, utiliser `emacs` : `emacs fich.c`. Pour la compilation, taper `gcc -ansi -pedantic fich.c` ou `gcc -ansi -pedantic fich.c -o fich`. Pour l'exécution, taper `a.out` ou `fich`.

4.2.5 Compilation séparée

La compilation séparée est très recommandée dès que vos logiciels sont importants.

1. Compiler séparément sans faire l'édition des liens chacun de vos fichiers avec l'option `-c` : `gcc -ansi -pedantic -c fichier.c`. Cela génère un fichier avec extension `.o`, ici il y a la création de `fichier.o`, mais pas de fichier `a.out`.
2. Faire l'édition des liens : `gcc -o executable fichier.o utilitaires.o`. Cela génère un fichier de nom `executable` (option `-o`) à partir des différents codes objet.

Lorsque vous êtes amené(e)s à utiliser un ensemble de modules (comprendre de fichiers contenant du code), il est bien pratique de ne recompiler que ce qui est utile après des modifications. Dans la solution ci-dessus, quand vous faites des modifications, c'est à vous de gérer les dépendances et ne recompiler que ce qui est nécessaire (à moins de tout recompiler).

4.2.6 L'outil make

Pour gérer les dépendances et ne recompiler que ce qui est nécessaire, il existe un outil `make` paramétré avec un fichier `makefile` ; il n'y a qu'à laisser le système se débrouiller (il fait ça très bien !). Pour construire un fichier `makefile`, il faut un ensemble de sections constituées chacune de deux parties : une partie cible/dépendances et une partie action :

Partie cible/dépendances La cible est le résultat de l'action si les dépendances ne sont pas vérifiées. Par exemple, le fichier objet cible (`.o`) sera produit par une action (compilation) si les fichiers dont la cible dépend (les sources) sont plus récents.

Partie action C'est la commande à exécuter pour obtenir la cible. Cette action doit être obligatoirement précédée du caractère tabulation.

Voici un exemple :

```
<nom executable> : <noms des fichiers objets>
      gcc -o <nom executable> <noms des fichiers.o>
<nom fichier>.o : <noms des fichiers dont il a besoin>
      gcc -ansi -pedantic -c <nom fichier>.c
```

L'interprétation est la suivante :
– Pour faire un exécutable, j'ai besoin d'un certain nombre de codes objet (dépendance) et je construis l'exécutable à partir d'un ordre de compilation (`gcc`).
– Pour faire un code objet, j'ai besoin d'un certain nombre de `<fichier>.h` ou de `<fichier>.c` et je construis le code objet à partir d'un ordre de compilation sans édition de liens (option `-c`).

Exemple (à découper selon les pointillés) :

```
-------------------------------------------------------------
all : jeu travail

jeu : jeu.o utilitaires.o
        gcc -o jeu jeu.o utilitaires.o

travail : travail.o utilitaires.o
        gcc -o travail travail.o utilitaires.o

utilitaires.o : utilitaires.c utilitaires.h
        gcc -ansi -pedantic -c utilitaires.c

jeu.o : jeu.h jeu.c utilitaires.h
        gcc -ansi -pedantic -c jeu.c

travail.o : travail.h travail.c utilitaires.h
        gcc -ansi -pedantic -c travail.c
-------------------------------------------------------------
```

L'appel de la compilation se fait par l'ordre `make` qui prendra par défaut le fichier `makefile`. La première fois, il compilera tout car les fichiers cible n'existent pas. Après une modification dans `utilitaires.h`, l'appel de `make` provoquera la recompilation de l'ensemble. Après une modification dans `jeu.c`, il ne recompilera que `jeu.c` et fera l'édition des liens de `jeu` mais ne touchera pas à `travail` par exemple.
Les erreurs les plus fréquentes sont :
– Pour poursuivre d'une ligne sur l'autre une commande, terminer la ligne par *backslash*.
– `make` impose une tabulation au début de la commande (et non des espaces ! !).
– Attention aux dépendances cycliques (A dépend de B qui dépend de A).

4.3 Erreurs fréquentes

Il n'est bien sûr pas question d'être exhaustif, mais de donner quelques idées et réflexes...

4.3.1 Erreurs à la compilation

Elles sont souvent détectées par l'analyseur syntaxique. Mais qu'est-ce qu'il dit ce compilateur ? La ligne 7 est correcte !

```
$ gcc -ansi -pedantic  f.c
    f.c: In function ‘main’:
    f.c:7: parse error before ‘;’
    f.c:7: parse error before ‘)’

$ cat f.c
    #include <stdio.h>
    #define N 10;              /* ligne 2 */

    main () {
      int i;

      for (i = 1; i < N; i++)   /* ligne 7 */
      printf ("%d",i);
    }
```

La ligne 7 est en effet correctement écrite ! Mais le préprocesseur (voir la ligne 2) a remplacé toutes les occurrences de `N` (d'où l'intérêt de l'avoir écrit en majuscule) par `10;`. Par suite la ligne 7 devient : `for (i = 1; i < N;; i++)` et le compilateur détecte une erreur en ligne 7...

4.3.2 Erreurs à l'édition des liens

```
$ gcc -ansi -pedantic fich.c
    ld: Undefined symbol
    _fct
```

Vous utilisez une fonction nommée `fct` sans l'avoir définie dans `fich.c`. Revoir vos prototypes ou les directives d'inclusion ou bien ajouter la définition de `fct` dans `fich.c`.

4.3.3 Erreurs à l'exécution

En C, ce n'est pas rare... Il y a seulement un `printf` et un `scanf` et les résultats (s'il y en a) sont étranges ! N'y aurait-il pas des phrases du type :

```
  scanf ("Entrez la valeur de x %d \n", &x); /* scanf est fait pour
                                                lire, pas pour écrire  */

  float x;
  scanf ("%d", &x);                         /* format incompatible */
```

Le programme choisit tout le temps la même alternative (voire ne s'arrête pas) : revoir les expressions conditionnelles et en particulier celles du type : `(x == a)` et vérifiez que vous n'avez pas écrit : `(x = a)`. Le programme s'arrête prématurément et affiche un message contenant *Memory fault* ou *Bus error* : c'est vraisemblablement lié aux pointeurs.

```
/* Lire un entier */
int x;
scanf ("%d", x); /* scanf doit mettre un entier à l’adresse donnée
                    par le contenu de x */

/* Affectation illicite  (réel -> entier). */
int x = 10;
int * p;
*p = 1.5;   /* p pointe sur un entier */

/* Afficher les éléments d’un tableau. */
#define N 100
int t[N];
int i;
for (i=0; i<=N; i++)
  printf ("%d\n", t[i]);
/* c’était bien parti...
   mais lorsque i vaut N vous cherchez à écrire t[N]
   (dépassement des limites du tableau ! */

/* lire une chaine */
char * ch;    /* ch est un pointeur sur caractère non initialisé */
scanf ("%s", ch); /* incorrect ! */
  /* il faut au préalable allouer une zone mémoire pointée par ch
     par exemple : ch = malloc (sizeof (char)*10);
     alloue une zone de 10 caractères  */
printf ("%s\n", ch);

/* Affectation illicite (chaine -> chaine). */
char ch1 [5] = "ab";
char ch2 [5] = "c";
ch1 = ch2;
```

Dans le dernier exemple, `ch1` et `ch2` sont des constantes de pointeurs égales aux adresses de `ch1 [0]` et de `ch2 [0]`. si celles-ci sont respectivement `100` et `200` tout ce passe comme si vous essayez d’écrire `100 = 200`... il faut passer par la fonction `strcpy (ch1, ch2)`.
Conseil : définissez les chaînes avec des tableaux. Réservez l’utilisation des char * aux paramètres formel si nécessaire.

4.4 Résumé

– Règle 1. Ne jamais écrire un programme avant d’avoir écrit et validé l’algorithme correspondant.
– Règle 2. Travailler avec `emacs` ou `xemacs`(configuré pour C).
– Règle 3. Respecter les Recommandations de style. En particulier, indenter et commenter vos programmes.
– Règle 4. Lancer l’exécution de vos programmes lorsque vous n’avez plus aucun *warning*. Un *warning* n’empêche pas la génération d’un exécutable, mais les statistiques montrent qu’un tel programme a plus de chances de se planter à l’exécution.

5 Recommandations de style

L'objectif de cette section n'est donc pas de contraindre la créativité des développeurs mais de fournir quelques pistes pour améliorer la lisibilité des programmes. Que pensez vous de ces deux extraits de programme (le premier est écrit en PL/1 un langage où il n'y a pas de mots clés) ?

```
/* PL/1 */
IF THEN THEN THEN = ELSE ; ELSE ELSE = THEN

/* C */
#include <stdio.h>
main(){int x,y;int z; printf("Entrez x "); scanf("%d", &x);
printf("Entrez y "); scanf("%d", &y); z=x;x=y;y=z;
printf("x=%d\ty=%d\n", x, y); return 0;}
```

S'ils vous semblent lisibles, faites relire un de vos vieux programmes à un ami et regardez sa tête ! La désignation et la présentation des entités qui composent un programme indépendamment de l'ordonnancement des actions (qui lui est défini par l'algorithme) conditionne en grande partie la lisibilité des programmes (donc le nombre d'erreurs de programmation).

1. Utilisation des majuscules et des minuscules.
   – Les identificateurs de variables, de fonctions et de procédures s'écrivent en minuscules.
   – Les constantes s'écrivent en majuscules.
   – Les noms de type définis avec `typedef` commencent par une majuscule.
2. Choix des désignations.
   – Les noms doivent être évocateurs (pas nécessairement très longs). Bannir les désignations telles que `toto`...
   – Les articulations sont mises en valeur par le `_` : `creer_pile` ou alors par une majuscule : `creerPile`.
   – Lorsque deux noms ayant un lien possèdent des attributs communs, mettre en tête ce qui les distingue : `abscisse_point`, `ordonnee_point`.
   – Les noms de procédures (qui permettent d'exécuter des actions) doivent être dérivés de verbes sur le mode impératif (en anglais) ou infinitif (en français) : `write`, `afficher`, `creer_pile`.
   – Les fonctions et les attributs (qui décrivent des l'accès à des informations) sont désignés par un nom éventuellement suivi d'un qualificatif : `valeur`, `arbre_gauche`.
   – Dans le cas des fonctions et des attributs de type booléen, le nom pourra suggérer une interrogation : `est_vide`, `est_une_feuille` ou être désigné par un adjectif : `vide`, `ouvert`...
3. Commentaires.
   – Toute procédure ou fonction doit commencer par un commentaire d'en-tête significatif, clair et concis.

```
float distance_a_origine (Point p)
/* distance par rapport au point (0, 0) */
{
  /* ... */
}
```

   – Ne pas répéter une information évidente par la portion de source adjacente.
   – Éviter les mots ou phrases superflus : cette fonction retourne...
   – Les restrictions sur l'usage de la procédure ou de la fonction doivent être indiquées au niveau des prototypes.
   – Ne pas utiliser d'abréviations (un commentaire c'est fait pour expliquer).
   – Types de commentaires :
     – procédure : une phrase sur le mode impératif ;

  - fonction (non booléenne) : une expression nominale ;
  - fonction booléenne : une question.
- Les éclaircissements dans les endroits délicats font l’objet de commentaires libres.

4. Agencement du code.
   - Caractères suivis par un blanc qui ne sont pas précédés par un blanc : ’)’, ’,’, ’;’, ’.’.
   - Caractères précédés par un blanc qui ne sont pas suivis par un blanc : ’(’.
   - Les opérateurs relationnels, arithmétiques sont suivis et précédés par un blanc.
   - L’indentation joue un rôle essentiel dans la lisibilité. L’exemple suivant peut servir de *modèle*.

```
/* somme des ventes */
#include <stdio.h>
#define N 10     /* taille des tableaux */
/* lire_tableau (t, n) :
   prend en entrée le nombre d’éléments à lire n et retourne t */
void lire_tableau (float t[], int n);

/* somme (t, n) :
    retourne la somme des éléments d’un tableau t de taille n en entrée */
float somme (float t [], int n);

main() {
  float ventes [N] ;  /* tableau des ventes */
  float s = 0;    /* somme des ventes */

  printf ("Entrez les valeurs des ventes \n");
  lire_tableau (ventes, N); /* saisie des ventes */
  s = somme (ventes, N);
  printf ("La somme des ventes est de : %f\n", s);
}

/* lire_tableau (t, n):
    prend en entrée le nombre d’éléments à lire n et retourne t */
void lire_tableau (float t[], int n) {
  int i;

  for (i = 0; i < n; i++) {
    scanf ("%f", &t[i]);
  }
}

/* somme (t, n) :
   retourne la somme des éléments d’un tableau t de taille n en entrée */
float somme (float t [], int n) {
  float sc = 0 ; /* somme courante */
  for (i = 0; i < n; i++) {
    sc = sc + t[i];
  }
  return sc;
}
```

6 Correspondance langage algorithmique → langage C

types	Caractère, Entier, Réel	**char**, **int**, **double**
Définitions d'objets	objet1, objet2 : NomType	**NomType objet1, objet2;**
Actions simples	instruction	**expression;**
Actions composées (N>1)	instruction 1 . . . instruction N	**{** **expression 1;** **...** **expression N; }**
Affectation	var ← valeur	**var = valeur;**
Conditionnelle (1ère forme)	**si** condition **alors** action 1 **sinon** action 2 **fsi**	**if (condition)** **action 1** **else** **action 2**
Itération (forme canonique)	**tant que** condition **faire** action **ftq**	**while (condition)** **action**
Itération (variante)	**faire** action **tant que** (condition)	**do** **action** **while (condition);**
Itération (schéma **pour***)*	**pour** i = val1 : Naturel **à** val2 **pas** h action **fpour** i	**for (i=val1; i<=val2; i=i+h)** **action**
Deux opérateurs	**div** **mod**	**/ // opérandes de type entier** **%**
Deux actions simples	saisir (x) afficher (y)	**scanf ("%d", &x);** **printf ("%d ", y);**

Un exemple simple :

// Algorithme *racine* **fonction** principale ( ) r : Réel afficher ("Saisir une valeur >= 0 : ") saisir (r) **si** r < 0 **alors** afficher ("tricheur ! !") **sinon** afficher ("racine(", r, ") = ",racineCarrée(r)) **fsi** allerAlaLigne ( ) **retourner** OK **ffct** principale	**// compilation :** **// gcc -Wall -lm racine.c -o racine** **#include <stdlib.h> // pour EXIT_SUCCESS** **#include <stdio.h> // scanf et printf** **#include <math.h> // sqrt** **int main ( ) {** **double r;** **printf ("Saisir une valeur >= 0 : ");** **scanf ("%lf", &r);** **if (r < 0)** **printf ("tricheur ! !");** **else** **printf("racine(%f) = %f\n",r,sqrt(r));** **return EXIT_SUCCESS;** **}**

Et la suite :

Fonction	**fonction** nom(objet :Type) :TypeRésultat action // corps de nom **retourner** valeur **ffct** nom // Exemple d'appel : x ← nom (expression)	**TypeResultat nom(Type objet) {** **action // corps de nom** **return valeur;** **}** **// Exemple d'appel :** **x = nom (expression);**
Procédure	**procédure** verbe (**donnée** x : TypeX **donnée-résultat** y :TypeY **résultat** z : TypeZ) action sur x, y et z **fproc** verbe // Exemple d'appel : // a, b et c sont de types TypeX, TypeY, TypeZ verbe (a , b , c)	**// voir notes de bas de page** **void verbe (TypeX x,** **TypeY *y,** **TypeZ *z) {** **action sur x, *y et *z** **}** **// Exemple d'appel :** **verbe (a, &b, &c);**

Les notations ***y** et ***z** ne s'utilisent que si **TypeY** et **TypeZ** ne sont pas des types tableau (2.9.1).

Définitions d'objets de types composés	objet : **tableau**[1..N] **de** TypeElt objet : **structure** champ1 : Type1 . . . champN : TypeN **fstruct**	**TypeElt objet[N];** **struct nom { Type1 champ1;** **...** **TypeN champN;** **} objet;**
Définitions de types	**type** TypeTableau = **tableau**[1..N] **de** TypeElt **type** TypeStructure = **structure** champ1 : Type1 . . . champN : TypeN **fstruct**	**typedef TypeElt TypeTableau[N];** **typedef struct { Type1 champ1;** **...** **TypeN champN;** **} TypeStructure;**
Conditionnelle 2ème forme	**cas où** expression **vaut** valeur1 : action1 . . . valeurN : actionN autre : actionParDéfaut **fcas**	**switch (expression) {** **case valeur1 : action1** **break;** **...** **case valeurN : actionN** **break;** **default : actionParDefaut }**
Constantes	**constante** NB_ELEMENTS = 10	**#define NB_ELEMENTS 10**

Références

[Aho et al., 1974] Aho, A., Hopcroft, J., and J., U. (1974). *The design and analysis of computer algorithms*. Addison-Wesley.

[Aho et al., 1987] Aho, A., Hopcroft, J., and Ullman, J. (1987). *Structures de données et algorithmes*. InterEditions.

[Braquelaire, 2005] Braquelaire, A. (2005). *Méthodologie de la programmation en langage C*. Dunod.

[Cormen et al., 2002] Cormen, T., Leiserson, C., Rivest, R., and C., S. (2002). *Introduction à l'agorithmique*. Dunod.

[Cries, 1985] Cries, D. (1985). *The science of programming*. Springer Verlag.

[Drix, 1994] Drix, P. (1994). *Langage C. Norme ANSI*. Masson.

[Froidevaux et al., 1994] Froidevaux, C., Gandel, M., and Soria, M. (1994). *Types de données et algorithmes*. Ediscience int.

[Horowitz et al., 1993] Horowitz, E., Sahni, S., and Anderson-Freed, S. (1993). *L'essentiel des structures de données en C*. Dunod.

[Kernighan and Ritchie, 1992] Kernighan, B. and Ritchie, D. (1992). *Le langage C (ANSI)*. Masson.

[Rifflet, 1994] Rifflet, J. (1994). *La programmation sous Unix*. Ediscience International.

[Roberts, 1995] Roberts, E. (1995). *The Art and Science of C*. Addison Wesley.

[Sedgewick, 1991] Sedgewick, R. (1991). *Algorithmes en langage C*. InterEditions.

[Wirth, 1986] Wirth, N. (1986). *Algorithms and Data Structures*. Prentice Hall.

Index

D

E

F

P

Q

R

S

T

U

V

W

X

Z