Cours de C - ANSI
3ème partie (2 cours de 2 heures)
Bertrand LIAUDET

Qu'est-ce que la science ? La science c'est ce que le père enseigne à son fils. Qu'est-ce que la technologie? C'est ce que le fils enseigne à son père.

Michel Serres

SOMMAIRE DE LA TROISIEME PARTIE

CHAPITRE 14 : LES ENTREES SORTIES CONVERSATIONNELLES

14.1 Écrire à l'écran : printf

1 Définition

La fonction printf permet d'écrire à l'écran.

On a vu que le premier paramètre est la chaîne de caractères qu'on écrit à l'écran.

exemple :

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

écrit 3 si x vaut 3.

syntaxe :

```
printf (format, liste d'expressions);
```

2 Paramètrages

On peut paramètrer quatre éléments dans le format : le code de conversion (%d) et le gabarit, la précision et le cadrage de l'affichage.

Gabarit[1]

Par défaut les entiers sont affichés avec le nombre de caractères nécessaires. Les flottant sont affichés avec 6 chiffres après la virgule.

Un nombre placé entre le % et le code de conversion précise le gabarit d'affichage, c'est-à-dire un nombre minimal de caractères à utiliser.

```
printf(":%3d:",n);
```

```
n=3             :^^3:
n=-5200         :-5200:
```

```
printf(":%f:", x)
```

notation décimale, par défaut, 6 chiffres après la virgule

```
x = 1.2345      :1.234500:
x = 12.3456789 :12.345678:
```

1 Del97 p.55.

```
printf(":%10f",x)
```

notation décimale, gabarit de 10 minimum, par défaut, 6 chiffres après la virgule

```
x = 1.2345      :^^1.234500:
x = 12.345      :^12.345000:
x = 1.2345E5    :123450.000000:
```

```
printf(":%e",x)
```

notation décimale, gabarit de 10 minimum, par défaut, 6 chiffres après la virgule

```
x = 1.2345           :1.234500e+000:
x = 123.45           :1.234500e+002:
x = -123456789E8     :-1.234568e+010:
```

Précision

On peut limiter le nombre de chiffres après la virgule par un nombre précédé d'un point et situé entre le % et le code de conversion.

```
printf(":%10.3f",x)
```

```
x = 1.2345      :^^^^^^1235:
x = 1.2345E3    :^^1234.500:
```

```
printf(":%12.4e",x)
```

```
x = 1.2345           :^1.2345e+000:
x = 123.456789E8     :^1.2346e+010:
```

Cadrage

Le signe moins "-" placé immédiatement après le symbole % ("%-4d") demande de cadrer l'affichage à gauche et non pas à droite comme c'est le cas par défaut. Les éventuels caractères supplémentaires seront alors placés à droite.

RETENIR : `%-12.3f : gabarit et précision et cadrage`

Précision ou gabarit variable

Le caractère * figurant à la place d'un gabarit ou d'une précision signifie que la valeur est fournie dans la liste des argument du printf.

```
printf (":%8.*f:", n, x);
```

`n = 3, x = 1.2345, ^^^1.234` (trois chiffres après la virgules, gabarit 8).

3 Chaînes de caractères

Soit la chaîne "bonjour, monsieur" de 17 caractères. On peut y appliquer les mêmes principes de gabarit et de précisions. La précision c'est le nombre de caractères effectifs.

```
:%s:          :bonjour, monsieur:     (normal)
:%12s:        :bonjour, monsieur:     (au minimum 12, ici 17)
:%.12s:       :bonjour, mon:          (les 12 premiers)
:%20s:        :   bonjour, monsieur:  (minimum 20,cadré à
                                      droite)
:%-20s:       :bonjour, monsieur   :  (minimum 20,cadré à
                                      gauche)
:%20.12s:     :        bonjour, mon:  (minimum 20, 12 premiers,
                                      à droite)
:%-20.12s:    :bonjour, mon        :  (minimum 20, 12 premiers,
                                      à gauche)
```

RETENIR : :%20s: et :%-20s:

14.2 Lire au clavier : scanf

1 Définition

La fonction scanf permet de lire des valeurs au clavier.

exemples :

```
scanf ("%d", &n);
scanf("%d%d", &n, &m);
scanf("%d%c", &n, &c);
```

syntaxe :

```
scanf (format, liste de variables);
```

2 Liste des variables : mode de passage

On peut faire deux remarques concernant le passage des paramètres : l'une d'un point de vue algorithmique, l'autre d'un point de vue technique C.

point de vue algorithmique : paramètres en sortie

Bien sur, à la différence du printf, on a des variables et pas des expressions. Ces variables sont en sortie alors que pour le printf, elles étaient en entrée.

Lecture :

```
scanf ("%d", &n);
```

c'est la même chose que

```
lire(n);
```

ou encore

```
n <- clavier;
```

Ecriture :

```
printf ("%d \n", n)
```

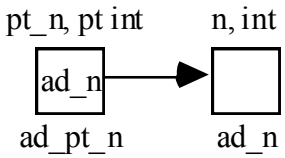
c'est la même chose que

```
écrire (n);
```

ou encore

```
écran <- n;
```

Les variables du scanf sont précédées d'un "&" : cela signifie qu'elles sont passées par adresse, c'est-à-dire qu'on ne passe pas la variable mais l'adresse de la variable. Dans l'exemple précédent, on ne passe donc pas un entier, mais on passe un pointeur d'entier.



C'est parce que la variable est en sortie de la fonction qu'on la passe par adresse.

14.3 Précisions sur Les E/S

1 Précisions sur le scanf

On peut paramètrer plusieurs éléments du format: le code de conversion (%d), le gabarit maximum et de qui concerne les régles de lecture.

Gabarit

On peut imposer le nombre de caractères maximum

scanf("%3d%3d", &n, &m)

12^25¬ -> 12 et 25

^^^^12345¬ -> 123 et 45

Règles de lecture

Cas des entiers et des réels

Si on a une série d'entiers ou de réels, chaque lecture doit être séparée par un ou plusieurs séparateurs (espace, saut de ligne (" \n"), tabulation horizontale (" \t"), tabulation verticale (" \v"), retour chariot (" \r"), saut de page (" \f")).

De plus, les espaces entre les formats sont ignorés.

```
scanf("%d%d", &n, &m);
```

```
12^25¬
^^^^^12^^^^^25^^^¬
12¬¬25¬
```

Cas des caractères

```
scanf("%d%c", &n, &c);
```

Si on a un caractère : l'espace comme le retour seront interprétés comme des caractères.

```
12a¬          : 12 et 'a'
12 a¬         : 12 et ' '
```

Cas où il y a un séparateur entre les deux formats :

```
scanf("%d %c", &n, &c);
```

```
12a¬          : 12 et 'a'
12^a¬         : 12 et 'a '
12^^^^^a¬     : 12 et 'a '
```

Caractères littéraux dans le format[2]

```
scanf("%d/%d/%d, &jour, &mois, &annee);
```

Avec un tel format, nous pouvons lire les dates de la forme jj / mm / aa

```
20/01/99      : 20 et 01 et 99
```

2 K&R91 p. 157.

2 Lecture et écriture

Nouvelles fonctions

Il existe en C des fonctions et de macros qui permettent de lire et d'écrire spécifiquement des caractères où des chaînes de caractères.

Le prototype de ces fonctions est défini dans <stdio.h>

Caractère

int getchar (void)

getchar permet de lire un caractère à la fois sur l'entrée standard (stdin) qui est normalement le clavier.

c = getchar () joue le même rôle que scanf("%c", &c)

int putchar (int c)

putchar affiche un caractère sur la sortie standard (stdout) qui est normalement l'écran.

putchar (c) joue le même rôle que printf("%c", c)

Chaîne de caractères

char * gets (char * s)

gets lit les des caractères sur l'entrée standard en s'interrompant à la rencontre d'une fin de ligne (\n) ou d'une fin de fichier.

int puts (char * s)

écrit une chaîne de caractères sur la sortie standard (stdout) qui est normalement l'écran.

Quelques pièges à éviter, retour sur le scanf

exemple 1

```
#include <stdio.h>
void main (void) {
    int n, p;
    printf("entrez n :");
    scanf ("%d", &n);
    printf ("n=<%d> \n",n);
    printf("entrez p :");
    scanf ("%d", &p);
    printf ("p=<%d> \n",p);
}
```

```
entrez n :12 25¬
n=12
entrez p :
p=25            SE
```

(En gras souligné : ce qu'on saisit). Le programme ne s'est pas arrêté sur le deuxième scanf.

En effet, ce qui a été saisi au premier scanf est traité ainsi : dès l'apparition d'un séparateur, le nombre saisi est affecté à n. Ce qui reste derrière (25) reste dans un tampon qui sera

utilisé par le scanf suivant. Celui trouve donc un entier (25) et l'affecte à p sans que l'on puisse refaire une saisie.

exemple 2

```
#include <stdio.h>
void main (void) {
    char c='a';
    while (c != 'q') {
        printf ("entrez c :");
        scanf ("%c", &c);
        printf ("c=<%c><%d> \n", c, c);
    }
}
```

```
entrez c :a¬
c=<a><97>
entrez c :c=<
><10>
entrez c:    SE
```

Quand on tape : "a¬" ça fait deux caractères.10 c'est le code ASCII de "¬". Il y a donc deux affichages.

```
entrez c :a   ¬
c=<a><97>
entrez c:c=< ><32>
entrez c:c=< ><32>
entrez c:c=< ><32>
entrez c:c=<
><10>
entrez c:    SE
```

Quand on tape "a ¬", "a" suivi de trois espaces et un entrée, cela fait 5 caractères. 32 c'est le code ASCII de " " (espace).

Rappelons la syntaxe de scanf :

int scanf (char *scanf, sorties…)

scanf lit les caractères sur l'entrée standard (le clavier) et les interprète selon les spécifications du format. Il place ensuite les résultats dans les arguments suivants. La fonction s'arrête quand elle a fini de parcourir le format ou lorsqu'une entrée ne correspond pas aux spécifications de format. La fonction retrourne le nombre de données correctement affectées en sortie.

La chaîne de caractères lue sur l'entrée standard et sur laquelle scanf travaille est appelée "tampon" (il absorbe tout ce qu'on lui rentre). Ce tampon n'est vidé qu'à la fin du scanf : le scanf s'arrête en fonction du format, pas en fonction du tampon. Ce tampon est exploré caractère par caractère en fonction du besoin, c'est-à-dire du format. Dès que les besoins sont satisfaits, ou qu'un besoin ne peut pas être satisfait, l'exploration s'arrête et la fonction aussi. Mais tout ce qui

n'a pas été utilisé est conservé pour la prochaine utilisation du scanf[3]. L'appel suivant du scanf reprendra l'interprétation du reste du tampon avec le nouveau format.

Dans notre exemple, chaque caractère lu, y compris les espaces et les fins de ligne, est interprété.

exemple 3

```
#include <stdio.h>
void main (void) {
    char c='a', s[10];
    s[0] = 'a';
    while (s[0] != 'q') {
        printf ("entrez ch :");
        scanf ("%s", s); /* s ou &s : c'est pareil */
        printf ("s=<%s> \n", s);
    }
}
```

```
entrez ch :bla¬
s=<bla>
entrez ch :¬
¬
¬
bla¬
s=<bla>
entrez ch :qla   ¬
s=<qla>
```

Quelle que soit la saisie, on obtient le même résultat et un bon résultat.

Notons qu'on ne peut pas saisir d'espace en fin de chaîne.

"Notons aussi que la norme prévoit que si l'on applique l'opérateur & à un nom de tableau, on obtient… l'adresse du tableau. Autrement dit, &s est équivalent à s."[4]

exemple 4

```
#include <stdio.h>
void main (void) {
    char c='a', s[10];
    while (c != 'q') {
        printf ("entrez c :");
        scanf ("%c", &c);
        printf ("c=<%c><%d> \n", c, c);
        printf ("entrez ch :");
        scanf ("%s", s);
        printf ("s=<%s> \n", s);
    }
}
```

```
entrez c :s¬
c=<s><115>
```

3 K&R91 p. 155, Del97 p. 60.

4 Del97 p. 159.

```
entrez ch :bla¬
s=<bla>
entrez c : c=<
><10>
entrez ch :bla ¬
entrez c : c=< ><32>
entrez ch :bla->¬
entrez c : c=<    ><9>
entrez ch :
etc…
```

A noter dans cet exemple qu'on ne peut pas quitter la boucle.

Explication et solutions : gets et sscanf[5]

gets et scanf n'ont pas exactement le même effet : avec gets, seule la fin de ligne sert de délimitateur (pas les espaces). De plus, et surtout, contrairement à ce qui se passe avec un scanf, le caractère de fin de la ligne est effectivement consommée : il ne sera pas pris en compte lors d'une nouvelle lecture.

Pour fiabiliser la lecture au clavier au peut associer le couple de fonctions : gets et sscanf.

sscanf (char * che, char * format, sorties …)

sscanf c'est la même chose qu'un scanf mais au lieu de traiter la chaîne saisie au clavier, sscanf traite la chaîne fournie en entrée (che). Elle renvoit le nombre d'objet convertis.

Exemple :

```
#include <stdio.h>
void main (void) {
    char c='a', s[10];
    int n, compte;
    do {
        printf ("entrez un entier et un caractère : ");
        gets (s);
        compte = sscanf (s,"%d %c", &n, &c);
    }
    while (compte != 2);
    printf ("compte=<%d> c=<%c> n=<%d>\n", c, n);
}
```

```
entrez un entier et un caractère : bof¬
entrez un entier et un caractère : a 9¬
entrez un entier et un caractère :   12 a  ¬
compte=<2> c=<a> n=<12>
```

5 Del97 pp. 158-161.

14.4 Nouvelles fonctions de lecture

1 int getchar()

On a déjà vu cette fonction.

Elle renvoit le caractère lu au clavier.

2 int getc(FILE* flot) ou int fgetc(FILE *flot)

Ces fonctions renvoient un caractère lu sur un flot : un fichier ou le clavier.

Ces deux fonctions sont équivalentes. La fonction getc est en réalité une macro équivalente à la fonction fgetc : donc elle est plus rapide.

int getchar() <-> int getc(stdin)

3 char * gets(char *)

C'est un équivalent de getc pour une chaîne de caractères, mais il vaut mieux éviter de l'utilier[6].

6 Bra98 p.527.

14.5 Nouvelles fonctions d'écriture

int putc(FILE*)

1 int putchar(int cara)

Cette fonction écrit le caractère en entrée à l'écran.

2 int putc(int cara, FILE* flot) ou int fputc(int cara, FILE *flot)

Ces fonctions écrivent le caractère en entrée sur un flot : un fichier ou l'écran.

Ces deux fonctions sont équivalentes. La fonction putc est en réalité une macro équivalente à la fonction fputc : donc elle est plus rapide.

int putchar(int) <-> int putc(int, stdout)

3 int puts(char *)

C'est un équivalent de putchar pour une chaîne de caractères, mais il vaut mieux éviter de l'utilier[7].

7 Bra98 p.527.

CHAPITRE 15 : LES FICHIERS[8]

15.1 Généralités

1 La mémoire

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

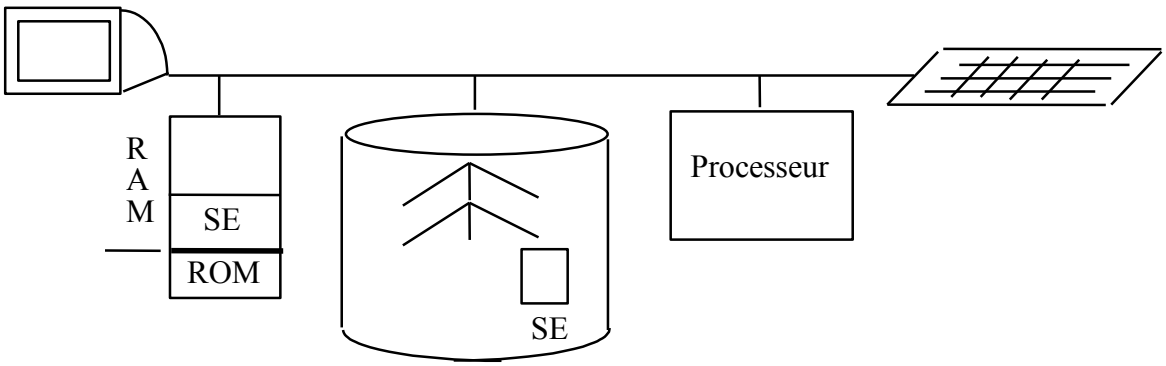
Il y a deux grandes dichotomies pour caractériser la mémoire :

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

Ca fait 3 types de mémoire :

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

2 Architecture des ordinateurs



Rappelons que quand l'ordinateur démarre, le petit programme codé dans la ROM va chercher le logiciel système d'exploitation (MsDos, Windows 98, Windows NT, UNIX, MacOS, etc. SE sur le schéma) sur le disque dur et le recopie sur la RAM. L'utilisateur communique alors avec le système d'exploitation. Le système d'exploitation offre trois grands types de fonctionnalité : la gestion de fichier (dir, cd, grep, etc.), la gestion des périphériques et le lancement des logiciels.

8 **Début du huitième cours.**

3 Les types de fichiers

Deux grandes dichotomies :

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

Ca fait quatre types de fichiers :

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

4 Organisation du rangement des fichiers

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

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

5 Fichier et variable en programmation

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

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

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

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

15.2 Algorithmique des fichiers

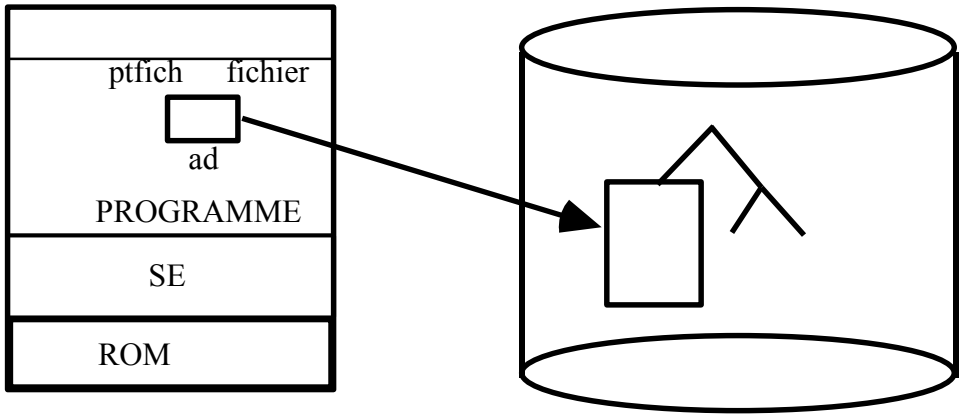
Avant de présenter les fonctions C de traitement de fichier, on va présenter une sorte d'axiomatique du traitement des fichiers, c'est-à-dire le jeu minimal des fonctions qui permet de faire tous les traitements possibles sur les fichiers.

1 Le type fichier

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

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

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



2 Créer un fichier

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

pt_fichier **<- creerFichier** (nom_du_fichier in) **;**

3 Ouvrir un fichier pour le consulter

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

pt_fichier **<- consulterFichier** (nom_du_fichier in) **;**

4 Ouvrir un fichier pour le modifier

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

pt_fichier **<- modifierFichier** (nom_du_fichier in) **;**

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

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

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

a <- **lireFichier** (pt_fichier inout)

6 Ecrire un élément et passer au suivant

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

ecrireFichier (pt_fichier inout, a out)

7 Se déplacer de n élément

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

deplaceFichier (pt_fichier out, n in)

8 Repérer la fin du fichier

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

oui_non = **EOF** (pt_fichier in)

9 Fermeture d'un fichier

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

fermeFichier (pt_fichier **in**) **;**

10 Exemples

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

récupérer dans un tableau tous les éléments d'un fichier

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

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

écrire dans un fichier tous les éléments d'un tableau

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

15.3 Les fichier binaires en langage C

On va maintenant présenter les fonctions du langage C qui permettent de manipuler des fichiers binaires.

1 Le type FILE *

Pointeur de fichier : pour pouvoir travailler sur un fichier, le C dispose d'un nouveau type : FILE * . Celui-ci permet de déclarer des variables qui contiennent en particulier un pointeur sur le fichier sur lequel on va travailler.

En C, on écrit : FILE * ptfich;

On déclare donc un pointeur.

2 Premier exemple : création d'un fichier d'entiers

```
#include <stdio.h>
void main (void)
{
    char nomfich[20];
    int n;
    FILE * ptfich;
    printf("nom du fichier :");
    scanf("%s", nomfich);
    ptfich=fopen(nomfich, "w");
    do{
        printf("entrez un entier :");
        scanf("%d", &n);
        if (n) {
            fwrite (&n, sizeof(int), 1, ptfich);
        }
    }
    while (n);
    fclose(ptfich);
}
```

3 L'ouverture des fichiers : fopen

```
#include <stdio.h>
FILE * fopen (char * nomfich, char* mode)
/*
    en entrée :
        mode :mode d'ouverture du fichier
        nomfich : nom du fichier à ouvrir
    en sortie standard :
        renvoie un pointeur sur le fichier ouvert.
        NULL en cas d'erreur
*/
```

lecture

"r" : lecture seule

le mode "r" interdit l'écriture dans le fichier. Le fichier doit exister. S'il n'existe pas la fonction renvoie NULL.

écriture : création

"w" : écriture seule

le mode "w" est celui qui permet d'écrire dans le fichier. Attention, si le fichier existait déjà, toutes les données seront perdu. Si le fichier n'existe pas il est créé.

écriture : ajout

"a" : ajout seul

le mode a force le positionnement du pointeur à la fin du fichier lors de chaque opération d'écriture. Si le fichier n'existe pas, il est créé (c'est un "w").

lecture et écriture : la mise à jour

r+ : lecture et mise à jour d'un fichier existant

w+ : création d'un fichier avec possibilité de lecture et d'écriture.

a+ : création d'un fichier avec possibilité de lecture et d'écriture, l'écriture se faisant à la fin.

Mode	**Accès**	**Positionnement en écriture**	**Comportement** si le fichier existe	**Comportement** si le fichier n'existe pas
r	Lecture	au début	-	erreur
w	Ecriture	au début	initialisation RAZ	création
a		à la fin	-	création
r+	Lecture	au début	-	erreur
w+	et	au début	initialisation RAZ	création
a+	Ecriture	à la fin	-	création

RESUME : r : lecture, w ecriture au début, a : écriture à la fin. rwa+ : lecture et écriture.

Attention, en cas d'ouverture en mode +, il est nécessaire d'effectuer un positionnement (fsetpos, **fseek** ou rewind) ou un vidage de buffer (**fflush**) entre une lecture et une écriture[9].

Le mode + a peu d'intérêt car on évite le plus souvent de travailler directement sur le fichier : on commence en général par le recopier dans une variable de la mémoire vive (en totalité ou en partie) et ensuite on travaille sur cette variable. Ca va plus vite et c'est plus souple dans la manipulation.

9 Del97 p. 241 (remarque et p. 231).

4 Ecriture dans un fichier : fwrite

```
  #include <stdio.h>
  size_t fwrite (void * elt, size_t taille, size_t nb, FILE *
ptfich)
  /*
      en entrée :
          elt :pointeur sur l'élément à écrire
          taille : taille de l'élément à écrire
          nb : nombre d'éléments à écrire
          ptfich : pointeur de fichier dans lequel on écrit
      en sortie :
          ptfich : pointeur de fichier dans lequel on écrit
      en sortie standard :
          le nombre d'élément effectivement écrit
  */
```

5 La fermeture des fichiers : close

```
  #include <stdio.h>
  int fclose (FILE * stream)
  /*
      en entrée :
          un pointeur sur le fichier ouvert.
      en sortie standard :
          renvoit EOF (-1) en cas d'erreur, 0 sinon
  */
```

6 Deuxième exemple : affichage d'un fichier d'entiers

```
  #include <stdio.h>
  void main (void)
  {
      char nomfich[20];
      int n;
      FILE * ptfich;
      printf("nom du fichier :");
      scanf("¨%s", nomfich);
      ptfich=fopen(nomfich, "r");
      while (fread (&n, sizeof(int), 1, ptfich),!feof (ptfich)){
             printf("%d\n", n);
      }
      fclose(ptfich);
  }
```

7 Lecture dans un fichier : fread

```
  #include <stdio.h>
  size_t fread (void * elt, size_t taille, size_t nb, FILE *
ptfich)
  /*
      en entrée :
          taille : taille de l'élément à écrire
          nb : nombre d'éléments à écrire
          ptfich : pointeur du fichier dans lequel on écrit
      en sortie :
          elt :pointeur sur l'élément lu
          ptfich : pointeur du fichier dans lequel on écrit
      en sortie standard :
          renvoie le nombre d'élément effectivement écrit
  */
```

8 Tester la fin du fichier : feof

```
  #include <stdio.h>
  int feof (FILE * ptfich)
      en entrée :
          ptfich : pointeur de fichier dans lequel on écrit
      en sortie standard :
          0 si la fin de fichier n'est pas atteinte
```

Attention :

Dans notre exemple, on fait le feof après le fread, dans la condition de la boucle. En effet si on fait un feof après avoir lu le dernier élément du fichier, feof n'est pas encore vrai. Il vaut encore 0. feof ne prendra la valeur vrai (1), qu'après une tentative de lecture au delà du fichier. Si le fread était dans le corps de la boucle, on aurait deux fois le traitement du dernier élément du fichier.

9 Se positionner dans le fichier : fseek

```
  #include <stdio.h>
  int fseek (FILE * ptfich, long décalage, int origine)
  /*
      en entrée :
          décalage : nombre d'octets de déplacement
          origine : origine du déplacement
          ptfich : pointeur de fichier
      en sortie :
          ptfich : pointeur de fichier positionné
      en sortie standard:
          une valeur non nulle en cas d'erreur
  */
```

Cette fonction permet de ce déplacer soit à partir du début, soit à partir de la fin, soit à partir de la position courante du fichier.

Il existe trois valeurs possibles pour l'origine, conservées dans des constantes :

SEEK_SET : origine au début du fichier

SEEK_CUR : origine à la position courante du fichier

SEEK_END : origine à la fin du fichier.

Pour calculer le nombre d'octets du déplacement, on utilisera des formules du type :
sizeof (type) * nb_elt

Exemple :

```
FILE * entree
…
printf("entrez le numero de l'entier cherche :");
scanf("%d", &num);
fseek(entree, sizeof(int)*(num-1),SEEK_SET);
fread(&n, sizeof(int), 1, entree);
printf("l'entier numero %d = %d\n", num, n);
```

15.4 Les fichiers texte en langage C

On vient de voir les fichiers binaires (fichier d'entiers, de flottants, de structures, etc).

Passons maintenant aux fichiers texte : fichier lisible avec un éditeur.

1 Déclaration

La déclaration est la même qu'avec les fichiers binaire : FILE * ptFich.

2 Ouverture et fermeture

L'ouverture et la fermeture se font avec les même fonctions : fopen et fclose.

3 Lecture et écriture

On peut reprendre toutes les fonctions d'entrée-sortie (printf, etc…), ajouter un "f" devant (pour « file ») et passer en premier paramètre le pointeur de fichier :

```
fscanf (FILE * ptfich, etc…)
fprintf (FILE * ptfich, etc…)
etc…
```

exemple :

```
#include <stdio.h>
void main (void)
{
    char nomfich[20], mot[20];
    int val;
    FILE * ptfich;
    printf("nom du fichier :");
    scanf("%s", nomfich);
    ptfich=fopen(nomfich, "w");
    fprintf(ptfich,"bonjour 3");
    fclose(ptfich);
    ptfich=fopen(nomfich, "r");
    fscanf(ptfich,"%s %d", mot, &val);
    printf("mot: <%s> val: <%d>", mot, val);
    fclose(ptfich);
}
```

4 La notion de flot ou flux[10]

Un flot c'est ce qui arrive (ou ce qui sort) par un canal d'entrée ou de sortie.

stdin

C'est le flot du canal d'entrée standard, c'est-à-dire le clavier.

Il y a d'autres canaux d'entrée possibles : les fichiers, la souris, etc.

10 Bra98 p.236.

stdout

C'est le flot du canal de sortie standard, c'est-à-dire l'écran.
Il y a d'autres canaux de sortie possibles : les fichiers, l'imprimante, etc.

stdin et stdout sont gérés comme des fichiers

Exemple

```
printf("bonjour");
```

c'est

```
fprintf(stdout, "bonjour");
```

stderr

Quand on veut gèrer les messages d'erreurs, on les envoie sur stderr :

```
fprintf (stderr, "erreur");
```

Par défaut, stderr c'est l'écran, mais on peut le redéfinir, particulièment comme un fichier.

5 Autres fonction de lecture et écriture

```
c=fgetc (FILE * ptfich)
fputc (char, FILE * ptfich)
fgets (char *, int lgmax, FILE * ptfich)
fputs (char * , FILE * ptfich)
```

6 Redirection et « pipe »

Redirection

On peut rediriger la sortie standard d'un programme (l 'écran en général) dans un fichier, ou rediriger un fichier dans l'entrée standard d'un programme (le clavier en général).

Le programme suivant compte les '\n ' (passage à la ligne) qu'on lui fournit au clavier. Il s'arrête dès qu'il rencontre un EOF (ctrl Z).

```
#include <stdio.h>
void main (void)
{
    int c, nl;
    nl=0;
    while((c = getchar() ) != EOF){ //ctrl Z au clavier
        if (c == '\n'){
            nl++;
        }
    }
    printf("nombre de lignes : %d\n", nl);
}                                                          main.c
```

Ce programme devient intéressant si on redirige l'entrée standard. Pour compter le nombre de lignes de main.c, il suffit d'écrire :

```
main.exe < main.c
```

Avec visual C ++, dans project/settings/debug/program arguments, on écrit : <main.c

Pipe

Le « pipe » (tuyau) permet de rediriger la sortie standard d'un programme dans l'entrée standard d'un autre programme.

Le programme suivant recopie à l'écran tous les caractères lus au clavier, jusqu'à rencontrer un EOF.

```
#include <stdio.h>
void main(void)
{
    int c;
    c=getchar();
    while(c != EOF) { //ctrl Z au clavier
        putchar(c);
        c=getchar();
    }
}                                                          main.c
```

Ce programme devient intéressant si on redirige l'entrée et la sortie standard :

```
C:> main.exe < main.c > copie.c
```

Ainsi, on a créer une copie de main.c.

On aurait aussi pu écrire :

```
C:> type main.c | main.exe > copie.c
```

Ce qui revient à envoyer la sortie standard de « type main.c » (affichage à l'écran de main.c) dans l'entrée standard de main.exe, puis à rediriger le résultat dans copie.c.

CHAPITRE 16 : METHODES[11]

16.1 La programmation structurée

1 Qu'est ce que la programmation structurée :

- Au premier niveau, c'est la programmation dans laquelle on utilise trois structures pour contrôler de l'ordre d'exécution des instructions : la séquence d'instructions, le test et la boucle.

  Cela permet d'écrire un code plus performant et plus clair.

  La programmation structurée n'utilise jamais de « goto ».

- Au deuxième niveau c'est la programmation qui utilise le principe de la division du programme en sous-programmes. Un programme, comme un sous-programme, quelle que soit sa complexité, peut être considéré comme un système traitant des informations fournies en entrée et produisant des informations à la sortie.

2 Objectifs de la programmation structurée

- **L'adéquation au besoin** : chaque programme ou sous programme doit répondre correctement aux exigences fonctionnelles et ergonomiques de son cahier des charges.
- **La fiabilité** : chaque programme ou sous programme doit pouvoir être vérifié de telle sorte qu'il garantisse une fiabilité qui soit la plus haute possible et qui soit mesurable.
- **La maintenabilité** : chaque programme ou sous-programme doit pouvoir être corrigé ou mis à jour le plus facilement possible. C'est, en principe, ce que permet une décomposition logique du programme en sous-programmes.

3 Principe de la programmation stucturée

Le principe de la programmation structurée consiste **à décomposer un problème complexe considéré comme un « tout » en sous-problèmes plus simples**.

Ce type d'analyse ou de programmation est appelé analyse ou **programmation descendante**, **top-down programming** en anglais.

Cette première analyse, c'est aussi ce qu'on appelle l'**analyse fonctionnelle** ou l'**analyse générale**.

Le texte suivant décrit les principes de ce type d'analyse :

> Au lieu de ce grand nombre de préceptes dont la logique est composée, je crus que j'aurais assez des **quatre** suivants, pourvu que je prisse une ferme et constante résolution de ne manquer pas une seule fois à les observer. Le **premier** était de ne recevoir jamais aucune chose pour vraie, que je ne la connusse évidemment être telle […] Le **second**, de diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre. Le **troisième**, de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusqu'à la connaissance des plus composés […] Et le **dernier**, de faire partout des dénombrements si entiers et des revues si générales, que je fusse assuré de ne rien omettre.
>
> **Discours de la méthode, 2ème partie,**
> **René Descartes, 1637.**

11 **Début du dixième cours.**

Ce texte reste toujours d'actualité :

Dans l'analyse d'un problème il faut commencer par lister toutes informations traitées par le problème (en listant d'abord tous les usages attendus), donc toutes les informations en entrées et en sortie : "*faire partout des dénombrements si entiers et des revues si générale, que je fusse assuré de ne rien omettre*".

Il faut ensuite ne jamais hésiter, dès que le problème semble un peu difficile, à le décomposer en sous-problèmes (fonctions) dont la solution est remise à plus tard : "*diviser chacune des difficultés que j'examinerais en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre*". Pour que cette décomposition soit possible de façon rigoureuse, il faut que les sous-problèmes soient présentés avec leur liste d'informations en entrée et en sortie.

Une fois les fonctions simples définies, il faut les écrire et les vérifier indépendamment les unes des autres : "*ne recevoir jamais aucune chose pour vraie, que je ne la connusse évidemment être telle*".

Les fonctions pouvant en appeler d'autres, la vérification se fera des fonctions les plus simples jusqu'au programme complet : "*conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusqu'à la connaissance des plus composés*".

4 Les dinstinctions du développement

Il y a trois distinctions capitales dans le développement d'un logiciel.

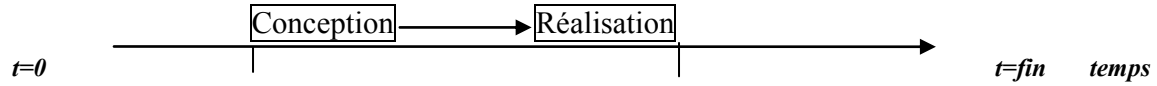
Première distinction : Développement = Conception + Réalisation

Le développement se compose de deux activités qu'on peut distinguer : la conception et la réalisation.

- **La conception** consiste à comprendre et prévoir ce qu'il a à faire.
- **La réalisation** consiste à faire concrètement ce qu'il y a à faire.

La distinction entre la conception et la réalisation est une façon d'organiser la division du travail.

Le premier principe de la méthode consiste à considérer ces deux activités comme deux étapes successives :

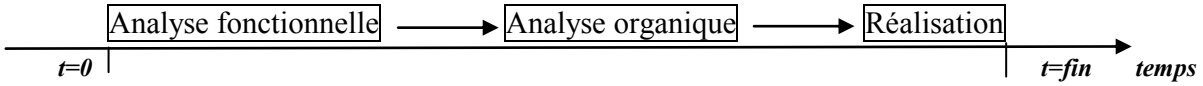


Le projet se déroule dans le temps : il commence avec la conception, il se termine avec la réalisation.

La division du travail consiste à mettre en évidence les étapes de la réalisation d'un logiciel.

La conception se divise en deux parties :

- L'analyse fonctionnelle d'abord. L'analyse fonctionnelle s'occupe des fonctions (ou des services) que le système offre à ses utilisateurs.
- L'analyse organique (ou architectonique[12]) ensuite. L'analyse organique s'occupe de la façon dont sera construit le système pour répondre aux attentes de l'analyse fonctionnelle.



Analyse fonctionnelle	**Analyse organique**
Point de vue de l'utilisateur	Point de vue de l'informaticien
Point de vue du maître d'ouvrage	Point de vue du maître d'œuvre
Point de vue de celui qui commande le logiciel	Point de vue de celui qui réalise le logiciel
Le QUOI	Le COMMENT
Externe	Interne
Build the right system	*Build the system right*
Construire le bon système	Construire bien le système

Avec cette distinction, on fait apparaître :

- le point de vue de l'utilisateur : le maître d'ouvrage (l'utilisateur, le client)
- le point de vue de l'informaticien : le maître d'œuvre.

Pour l'utilisateur, ce qui compte, c'est l'usage du système : les cas d'utilisation (vocabulaire UML).

Pour l'informaticien, ce qui compte c'est l'architecture interne du système.

L'analyse fonctionnelle a pour but de lister l'ensemble des cas d'utilisation : ce sont les fonctions du système. L'analyse fonctionnelle consiste à mettre au jour ces fonctions, avec leurs données en entrées et en sortie.

C'est ainsi qu'on garantit qu'on va construire le bon système : faire ce qui est demandé.

Au niveau de l'analyse fonctionnelle on précise, pour chaque fonction, son descriptif (ce qu'elle fait en une ligne) et son entête.

12 L'architectonique c'est la technique de la construction, mais aussi la structure ou l'organisation de la construction. Est architectonique ce qui est conforme à la technique de l'architecture.

L'analyse organique se divise en deux parties :

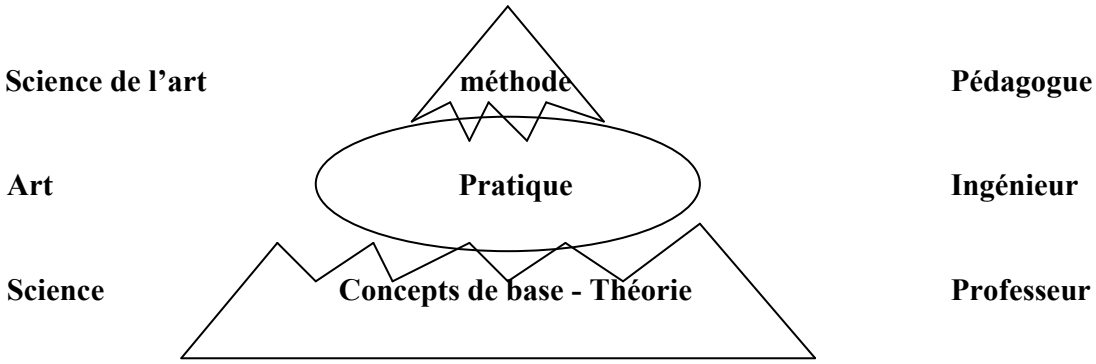
- L'architecture système (ou analyse organique générale): elle s'occupe de l'organisation des sous-systèmes logiciels et matériels du système complet. Tous les programmes abordés dans ce cours n'ont pas de sous-systèmes.
- L'analyse détaillée (ou analyse organique détaillée) : elle s'occupe du découpage en procédure et en fonctions informatiques de chacun des sous-systèmes logiciels. A ce niveau vont apparaître les en-têtes des fonctions, voir leurs pseudo-codes.

L'analyse organique garantit que ce qu'on va faire, on va bien le faire. Chaque fonction de l'analyse fonctionnelle donnera lieu à un algorithme et sera divisée en sous-fonctions, si nécessaire.

Pour chaque fonction, on précise :

- Descriptif
- Entête
- Usage
- Principe de résolution
- Algorithme

5 Schéma général



La méthode c'est en quelque sorte la théorie de la pratique.

6 Préceptes méthodologiques

Tout ceci nous amène à une série de préceptes méthodologiques

Précepte n°14 : Réfléchissez d'abord, vous programmerez plus tard

Précepte n°15: Définissez le problème complètement

Précepte n°16 : Utilisez l'étude descendante

Précepte n°17 : Méfiez-vous des autres études

Précepte n°18 : N'ayez pas peur de tout recommencer

Précepte n°19 : Fournissez une bonne documentation

16.2 La directive include

Nous allons maintenant présenter quelques outils du langage C qui permettent de mettre en œuvre correctement la méthode.

La directive include permet d'inclure dans un fichier source le contenu d'un autre fichier.

Les fichiers inclus contiennent en général, et dans cet ordre :

- des variables globales (à éviter)
- des constantes symboliques
- des définitions de types
- des prototypes de fonctions

Il y a deux formes du #include :

```
# include <nom_fichier>
```

La forme avec <…> recherche le fichier dans un emplacement (chemin, répertoire) défini par l'implémentation. C'est la directive d'inclusion des fichiers d'entêtes de la bibliothèque standard.

```
# include "nom_fichier"
```

La forme avec "…" recherche le fichier dans un emplacement précisé par le programmeur. C'est la directive d'inclusion des fichiers d'entêtes créés pour le programme.

On ne peut inclure qu'un seul fichier par directive. Il faut autant de directives que de fichiers à inclure.

Exemple :

Les valeurs limites qu'on trouve dans <limits.h> sont des constantes symboliques définie avec des #define :

Dans limits.h on trouve :

```
# define SHRT_MIN -32768
                                                    limits.h
```

Quand on écrit dans un programme C :

```
# include <limits.h>
                                                    Exemple.C
```

cela revient à inclure le fichier limits.h dans le programme C. C'est donc comme si on avait écrit directement dans le programme C :

```
# define SHRT_MIN -32768
                                                    Exemple.C
```

16.3 Méthodes de compilation

1 Compilation séparée

Les constituants d'un programme C

Les fichiers sources

Ce sont les fichiers " .C" (par exemple) qui contiennent le code des fonctions, dont le main.

Les fichiers d'entêtes

Ce sont les fichiers ".h" (par exemple) qui contiennent des déclarations de fonctions, de types et de variables globales.

Ces fichiers sont inclus dans les fichiers sources par le préprocesseur au moment de la compilation.

Les fichiers objets

Ce sont les fichiers ".o" ou " .obj" (en général). C'est le résultat de la compilation d'un fichier " .C".

Ce sont des fichiers qui jouent le rôle de bibliothèque de fonctions.

Le fichier exécutable

C'est le résultat de l'édition de lien de plusieurs fichiers objet.

Règles de bonne programmation

Pas d'instructions dans un fichier d'entête.

Les fichiers ne doivent pas être trop gros (200 lignes).

Il ne faut pas utiliser les include pour ajouter des fonctions complètes dans un fichier!

Pour vos projets : travaillez d'abord en mono-fichier.

Le makefile

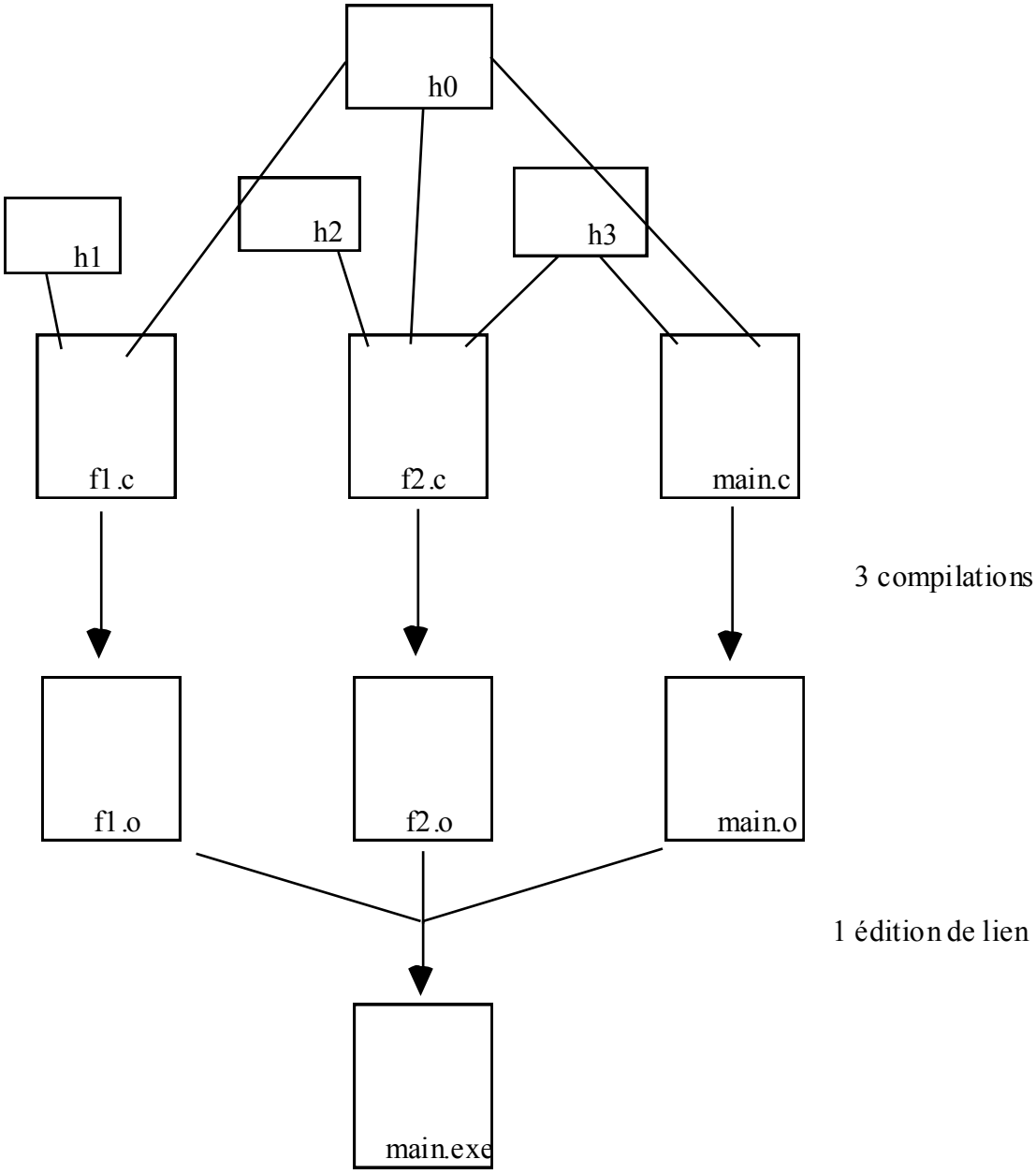
La notion de makefile est issue du monde UNIX.

Un makefile c'est un fichier texte créé automatiquement ou par l'utilisateur et qui contient les commandes de compilation.

Il dit où on va trouver les fichiers d'entête, les fichiers objets et les fichiers sources.

Il dit aussi quelles sont les dépendances entre ces fichiers, c'est-à-dire les recompilations nécessaires en fonction de la modifications des fichiers sources et des fichiers d'entêtes.

Exemple



Si h0 est modifié, il faut tout recompiler et refaire l'édition de liens.

Si h3 est modifié, il faut recompiler le main et f2 et refaire l'édition de liens.

Si le main est modifié, il faut recompiler le main et refaire l'édition de liens.

Utilité de la compilation séparée

La compilation séparée est utile pour le développement de gros logiciel. En effet, lorsqu'un logiciel est constitué de plusieurs centaines de procédures, tout recompiler peut prendre plusieurs minutes, dizaines de minutes ou heures. La compilation séparée permet de gagner du temps.

Elle permet aussi de développer des bibliothèques de fonctions qui sont stables et dont on utilisera que le .o.

2 Compilation conditionnelle

La compilation conditionnelle est une technique qui permet de choisir si des instructions seront compilées ou pas.

C'est très utile pour avoir des « printf » de mise au point, que l'on conserve et dont au évite l'usage grâce à la compilation conditionnelle.

C'est aussi utile pour adapter le programme à un type de matériel ou de compilateur ou à un autre.

Les directives de compilation conditionnelle se répartissent en deux catégories, suivant le type de condition invoquée :

Condition liée à la valeur d'une expression

```
#define DEBUG 1 // VRAI
#include <stdio.h>
void main (void) {
    #if DEBUG == 1 // VRAI
    printf ("debug\n") ;
    #endif
}
```

On peut aussi utiliser un #else un #elif

L'avantage de du #if par rapport à un « if » simple, c'est que le code ne sera compilé que si DEBUG est vrai. Ainsi, on évite d'alourdir le code exécutable du programme.

Condition liée à l'existence d'un symbole

```
#define DEBUG
#include <stdio.h>
void main (void) {
    #ifdef DEBUG
    printf ("debug 1\n") ;
    #endif

    #ifdef NODEBUG
    printf ("no debug 2\n") ;
    #endif
}
```

L'intérêt du #ifdef par rapport au #if c'est que l'on peut définir le symbole à la compilation.

Remarque

On peut aussi laisser les « printf » de debogage dans le programme et controler leur utilisation par un paramêtre passé directement au programme.

```
#include <stdio.h>
#include <string.h>
void main(int argc, char *argv[] )
{
    int DEBUG=0;
    printf("%d = <<%s>>\n", argc, argv[1]);
    if (argc == 2)
        if (strcmp(argv[1], "DEB")== 0)
```

```
            DEBUG=1 ;
    if (DEBUG == 1)
        printf ("debug\n") ;
}
```

On reverra ça avec l'utilisation de argv et argc.

3 Déclarations multi-fichiers et compilation séparée

Certaines caractéristiques des variables sont liées à la compilation séparée.

globales "extern"[13]

On distingue en C deux types de déclaration de variable : la déclaration-définition et la déclaration-référence.

La déclaration-défintion c'est la déclaration habituelle : elle définit la variable avec son nom et son type et elle engendre la **réservation de la mémoire** nécessaire pour les valeurs de la variable.

La déclaration-référence est une déclaration qui n'engendre pas de réservation mémoire. Les déclaration-référence sont des déclarations précédées du mot-clé "**extern**".

La déclaration-référence est utilisée en cas de compilation séparée. Elle permet de faire référence dans un premier fichier à une variable globale définie avec réservation de mémoire dans un autre fichier, et ainsi elle permet l'utilisation de la variable dans le premier fichier et sa compilation séparée.

globales "static"

Il est possible de limiter la portée d'une variable globale au fichier dans lequel elle est déclarée (et définie). Ceci se fait par adjonction du mot clé "**static**" à la déclaration de la variable.

Ce mot clé s'applique aussi à la déclaration des prototypes de fonctions. Ainsi les fonctions deviennent locales au fichier.

13 Del97 p. 117.

16.4 méthodes pour l'ihm : la vérification des types

En C, comme dans la plupart des langages, la saisie d'une mauvaise valeur risque d'engendrer soit l'arrêt brutal du programme (« plantage »), soit des résultats incohérents.

Par exemple, si on a :

```
scanf(« %f », &f)
```

et qu'on rentre un caractère, le programme peut « planter » ou continuer avec de mauvaises valeurs.

Il faudrait donc vérifier ce qu'on a saisi.

Pour cela, il faut dans tous les cas saisir une chaîne de caractères (ce qui n'engendre jamais d'erreurs) et faire des vérifications avant de transformer la chaîne dans le type souhaité.

Il existe trois jeux de fonctions pour effectuer ces conversions :

1 Les fonctions « a to ? »

```
#include <stdio.h>
int atoi (char * ch)
```

convertit ch en un int;

```
#include <stdio.h>
long atol (char *)
```

convertit ch en un long; équivaut à strtol (ch, NULL, 10) quand il n'y a pas d'erreur.

```
#include <stdio.h>
double atof (char *)
```

convertit ch en un double

2 sscanf

```
#include <stdio.h>
int sscanf (char * che, const char * format, …)
```

sscanf va permettre de transformer une chaine en entier ou en flottant.

Exemple :

```
#include <stdio.h>
void main (void) {
    char ch[20];
    int res, n1;
    float f;

    printf ("entrez 1 entier:") ;
    fflush(stdin);
    scanf("%s", &ch);
    printf("lu : <%s>\n", ch);
    res=sscanf(ch,"%d", &n1);
```

```
      printf("%d valeur%s décodé%s : <%d>\n", res, res>1?"s":"",
res>1?"s":"", n1);

      printf ("entrez 1 entier et un réel :") ;
      fflush(stdin);
      gets(ch);
      printf("lu : <%s>\n", ch);
      res=sscanf(ch,"%d %f", &n1, &f);
      printf("%d   valeur%s   décodé%s   :   <%d>   <%f>\n",   res,
res>1?"s":"", res>1?"s":"", n1, f);
  }
```

```
entrez un entier :2¬
lu : <2>
1 valeur décodé : <2>
entrez 1 entier et un réel : 2 4.3¬
lu : <2 4.3>
2 valeurs décodés : <2> <4.300000>
```

Remarques :

- Avec scanf on ne peut pas récupérer toute une chaîne mais on s'arrête au premier blanc.
- Avec gets on récupère bien toute la chaîne saisie (« 2 4.3 »), mais si on entrait : « 4.3 2 », on obtiendrait 4 et 0.3 ! => il vaut donc mieux traiter les éléments saisis un par un.

3 Les fonctions « str to ? »

```
#include <stdio.h>
long strtol (char * ch, char ** fin, int base)
```

long strtol (const char * che, char ** adfin, int base) : renvoie le nombre correspondant à la chaîne che(par exemple « 234 » renvoit 234), lu en base base. L'adresse de la fin de la chaîne peut être placé dans adfin. Si on parcourt toute la chaîne, on met adfin à NULL. L'usage classique est donc : res = strtol (ch, NULL, 10).

convertit ch en un long : ch est interprété en base "base". fin revoit l'adresse du caractère de ch sur lequel le décodage s'est arrêté. Si on arrive à la fin de la chaîne : fin donne l'adresse de "\0" de ch. Si aucune conversion n'a été effectué, fin vaut ch. Si on ne veut pas s'occuper de ce paramètre, il suffit de passer le pointeur NULL[14].

```
#include <stdio.h>
unsigned long strtoul (char * ch, char ** fin, int base)
```

convertit ch en un unsigned long.

```
#include <stdio.h>
double strtod (char * ch, char ** fin)
```

convertit ch en un double. Fonctionne comme strtol en base 10. On peut écrire les flottant sous forme décimal ou sous forme exponentielle. [15].

14 Bra98 p. 222.
15 Bra98 p. 226.

CHAPITRE 17 : DERNIERES NOTIONS

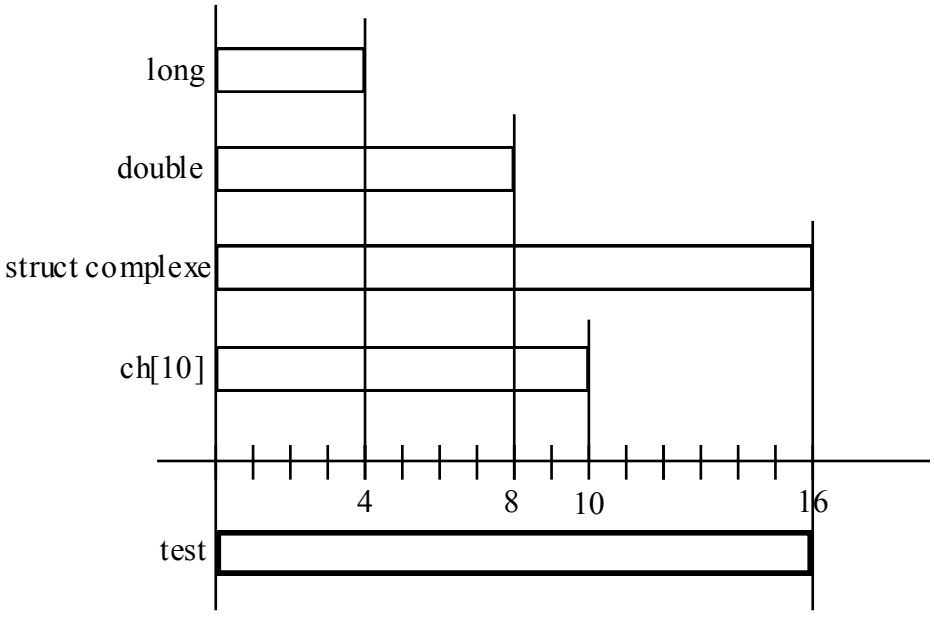
17.1 Les unions

Le langage C permet de partager un même emplacement de mémoire par des variables de types différents. C'est ce qu'on appelle les unions.

La syntaxe de l'union est la même que celle des struct :

```
struct complexe {
    double a;
    double b;
};
typedef union test {
    long entier;
    double reel;
    char ch[10];
    struct complexe complexe
} test;
test t1, t2;
```

t1 contient **soit** un entier, **soit** un double, **soit** une chaîne de 10, **soit** une structure.



D'un point de vue général, il vaut mieux éviter le type union : c'est un fourre tout! Il peut s'utiliser très occasionnellement dans des applications de bases de données.

17.2 Les énumérés

```
enum logique {NON, OUI};
```

Les énumérés ont peu d'intérêt[16].

17.3 Pointeurs génériques et pointeurs de fonction

Pointeur générique : void *. Exemple : void * malloc(int)

Les pointeurs de fonction permettent de rendre l'appel à la fonction variable. Cela permet de mettre ne œuvre les notions de polymorphisme et de surcharge de la programmation objet.

http://www.siteduzero.com/tutoriel-3-314203-les-pointeurs-sur-fonctions.html

17.4 La bibliothèque standard

Il y a **15** en-têtes standards définis dans la norme ISO.

Ces 15 entêtes permettent d'utiliser les **146** fonctions de la bibliothèque standard.

Pour connaître toutes les fonctions et tous les define standards, il faut se reporter soit à un manuel de référence, soit à l'aide du compilateur.

Il existe aussi des entêtes et des fonctions propres à chaque compilateur.

1 Les 4 entêtes les plus courantes

stdio.h

Pour les fonctions se rapportant aux entrées-sortie : affichage, lecture au clavier, fichier, etc.

stdlib.h

Pour les fonctions utilitaires : gestion de la mémoire, transformation de type, …

string.h

Pour les fonctions de traitement de chaînes.

ctype.h

Pour les fonctions destinées à tester les caractères.

2 Les 3 entêtes pour le calcul scientifique

math.h

Pour les fonctions mathématiques.

16 K&R91 p. 39.

float.h

Pour les constantes définissant les limites des flottants.

limits.h

Pour les constantes définissant les limites des entiers.

3 Les 8 autres

time.h

Pour les fonctions de traitement de la date et de l'heure.

errno.h

Pour la gestion des erreurs

assert.h

Pour la gestion des erreurs

signal.h

Pour envoyer des signaux entre des processus.

stdarg.h

Pour écrire des fonctions avec un nombre de paramêtres variables.

locale.h

Pour exploiter des spécificités locales dans l'affichage (alphabet, quantité monétaire, etc.)

setjmp.h

stddef.h

17.5 Les opérateurs de traitement des bits

Ils ne s'appliquent qu'aux entiers et aux caractères, soit char, short, int et long, signés ou non.
Ils sont au nombre de 6 :

&	ET bit à bit
|	OU inclusif bit à bit
^	OU exclusif bit à bit
<<	décalage à gauche

>> décalage à droite

~ complément à un (unaire)

Ces opérateurs correspondent à des opérations de bas niveau (non typées). On ne les expliquera pas plus dans cette partie du cours.

CHAPITRE 18 : GRAMMAIRE GENERALE DU LANGAGE C

18.1 L'alphabet

Dans tous les langages formels, dans les langages de programmation en particulier, on peut toujours ramener les symboles primitifs à une liste finie de symboles qui composent en quelque sorte l'alphabet du langage.

Pour pouvoir présenter clairement l'alphabet du langage de programmation C avec notre langage habituel, le français, il faut pouvoir distinguer les deux langages. En logique, on appelle le langage dont on parle (ici le langage de programmation) le langage objet, et le langage avec lequel on parle (ici le français) le métalangage (ou langage naturel). Pour distinguer les éléments du langage objet, le Pascal, des éléments du métalangage, le français, on présente généralement les éléments du langage objet entre guillemets.

L'alphabet du langage C comporte les caractères suivants :

- **26 lettres majuscules** : de « A » à « Z ».
- **26 lettres minuscules** : de « a » à « z ».
- **10 chiffres** : « 0 », « 1 », « 2 », « 3 », « 4 », « 5 », « 6 », « 7 », « 8 », « 9 ».
- **29 caractères graphiques :** « + », « - », « * », « / », « < », « > », « = », « ! », « . », « , », « ; », « : », « (», «) », « { », « } », « [», «] », « ' », « ^ », « _ », « # », « ~ », « \ », « ? », « % », « & », «/», « " ».
- **4 séparateurs :** l'espace, la tabulation, le saut de ligne, le saut de page.

18.2 Les mots

Avec cet alphabet, on peut construire les mots du langage. Ces mots, soit appartiennent à une liste prédéfinie, soit sont construits selon des **règles de formation** qui permettent de savoir si un mot appartient au langage ou pas.

Tous les mots sont construits avec des lettres, des chiffres ou des caractères graphiques.

Les mots sont séparés entre eux par des séparateurs ou par des opérateurs.

Il existe 6 sortes de mots dans le langage C :

- les 32 mots clés (ou mots réserves)
- les identificateurs
- les constantes
- les 45 opérateurs
- les commentaires
- les directives du préprocesseur

1 Les 32 mots clés

Cf. transparent III.2.1 (T10).

Il s'agit des mots prédéfinis par le langage C et qui ont une signification particulière. C'est une liste finie de 32 mots clés.

Un mot clé ne peut pas être utilisé comme identificateur.

On peut regrouper les mots clés en genres (qui ne sont qu'indicatifs) :

7 pour les types simples

"char", "double", "enum", "float", "int", "register", "void".

4 pour préciser les types simples

"long", "short", "signed", "unsigned".

3 pour les types complexes

"struct", "typedef", "union".

5 pour les variables

"auto", "const", "extern", "static", "volatile"

1 pour la mémoire

"sizeof".

5 pour les tests

"case", "default", "else", "if", "switch"

3 pour les boucles

"for", "while", "do"

4 pour les débranchements

"break", "continue", "goto", "return"

2 Les identificateurs

Un identificateur est un mot défini par l'utilisateur.

Les **règles de formation** de ces identificateurs sont les suivantes :

- Un identificateur est composé de lettres, majuscules ou minuscules, de chiffres et de soulignés.
- Il peut être de longueur quelconque mais seuls les 31 premiers caractères sont significatifs.
- Il ne peut pas commencer par un chiffre.
- Les lettres majuscules et minuscules sont différenciées.

Un identificateur étant composé à partir de 26 lettres, 10 chiffres et 1 souligné, soit 37 caractères, seuls les 31 premiers caractères étant significatifs, et le premier caractère ne pouvant pas être un chiffre, il y a : 27*(37 puissance 30) identificateurs possibles, soit environ 3 E+48!

3 Les constantes

Les constantes se divisent en :

- constantes entières,
- constantes flottantes,

- constantes énumérées,
- constantes caractères,
- constantes chaînes de caractères.

4 Les 45 opérateurs

Les opérateurs permettent de construire des expressions à partir de variables ou de constantes.

Il existe 45 opérateurs en C.

5 les commentaires

Les caractères « /* » marquent le début d'un commentaire qui se termine par les caractères « */ ». Les commentaires ne s'imbriquent pas et ne figurent pas à l'intérieur de constantes de type chaîne ou caractère.

6 les directives du préprocesseur

Il y a trois actions principales pour le préprocesseur :

- l'inclusion de fichier : #include
- la définition de constantes et de macros : #define
- la compilation conditionnelle : #ifdef, #else, #endif, #ifndef, #elif

18.3 Règles générales de formation d'un programme C

Un programme C a l'allure suivante :

```
# include <stdio.h>          /* les inclusion de fichiers */
# include <stdlib.h>
/* etc. */

# define N 5                                    /* les macros */
/* etc. */

int f1 (float,  int *);                      /* les prototypes */
char f2 (float );
/* etc. */

typedef                          / * les déclarations de types */
/* etc. */

int X;                /* les variables globales : à limiter ! */
/* etc. */

void main(int argc, char *argv[] )                / * le main */
{
    float a, b;
    /* les variables locales du main */
    instructions;
}
                                                   /* les fonctions */
/* ============================================
    f1 : en entrée x, float. Signification :
        en sortie y, char, signification
        un entier, signification
============================================*/
int f1 (float x,  char * y)
{
    float a, b;
    /* les variables locales de la fonction*/
    instructions;
}

/* ============================================
    f2 : en entrée x, float. Signification :
        en sortie un entier, signification
============================================*/
char f2 (float  x)
{
    int a, b;          /* les variables locales de la fonction*/
    instructions;
}

/* etc. */
```

equa1.c