Aide-mémoire de Langage C

Damien Mercier

6 février 2000

Table des matières

1 Eléments de syntaxe **4**
1.1 Structure générale d'un programme C 4
1.2 Un exemple complet 4

2 La compilation sous Unix **5**

3 Types, opérateurs, expressions **9**
3.1 Les types simples 9
3.2 Déclarations des variables 9
3.3 Types complexes, déclarations de type 11
3.4 Ecriture des constantes 12
3.5 Le transtypage (*cast*) 13
3.6 Les opérateurs 13
3.7 L'appel de fonction 15
3.8 Les fonctions *inline* 15
3.9 Les priorités 16

4 Tests et branchements **17**
4.1 La conditionnelle 17
4.2 Le *switch* 18
4.3 Les boucles 18
4.4 Les branchements 19

5 Tableaux et pointeurs **19**
5.1 Déclaration et utilisation des tableaux 19
5.2 Les pointeurs 20
5.3 Allocation dynamique de mémoire 21
5.4 Arithmétique sur les pointeurs 22
5.5 Les chaînes de caractères 23
5.6 Les pointeurs de fonction 24

6 Le préprocesseur **25**
6.1 L'inclusion de fichiers 25
6.2 Les macros (*#define*) 25
6.3 Compilation conditionnelle 26

7 Fonctions d'entrée/sortie **27**
7.1 Le passage d'arguments en ligne de commande 28
7.2 Entrée/sortie simples 28
7.3 Entrées/sorties avec format : *fprintf()*, *fscanf()* 29

8 Compléments : la bibliothèque standard et quelques fonctions annexes **29**
8.1 <stdio.h> : entrées-sorties 30
8.2 <ctype.h> : tests de caractères 34
8.3 <string.h> : chaînes de caractères et blocs mémoire 35
8.4 <math.h> : fonctions mathématiques 36
8.5 <stdlib.h> : fonctions utilitaires diverses 36
8.6 <assert.h> : vérifications à l'exécution 38
8.7 <limits.h> , <float.h> : constantes limites 38
8.8 <stdarg.h> : fonctions à nombre variable d'arguments 38

Introduction

Ce document n'est pas un manuel de programmation, ni un support de cours concernant le langage C. Il s'agit simplement d'un aide-mémoire qui devrait permettre à tout élève ayant suivi le cours de C d'écrire facilement des programmes. Il a été écrit à partir de l'excellent livre de référence de B. W. Kernighan et D. M. Ritchie intitulé *Le langage C, C ANSI*.

Le langage C dont il est question ici est celui défini par la norme ANSI de 1985 (et sa révision de 1989). Néanmoins, quelques ajouts seront faits, comme par exemple le type **long long**, car ils sont supportés par la plupart des compilateurs. De plus certains de ses ajouts seront intégrés dans la prochaine révision de la norme ANSI, prévue pour 2000 à notre connaissance.

Comme il est très difficile de présenter indépendamment les différents éléments d'un langage, il y aura parfois des références à des sections ultérieures dans le cours du texte. Cet aide-mémoire est un document de travail et il ne faut pas hésiter à aller et venir de page en page.

Pour une utilisation régulière et efficace du langage, il ne saurait être trop recommandé de compléter ce petit aide-mémoire par une lecture du livre de référence cité ci-dessus, en particulier la partie *Annexe A: manuel de référence* de la norme ANSI et *Annexe B : La bibliothèque standard*, dont un résumé succinct est présenté à la fin de ce document.

Le C a été développé au début par Dennis Ritchie, puis Brian Kernighan, au AT&T Bell Labs à partir de 1972, pour une machine de l'époque, le PDP-11. Le but était d'en faire un langage adapté pour écrire un système d'exploitation portable. Le langage fut appelé C car il héritait de fonctionnalités d'un précédent langage nommé B et était influencé par un autre langage nommé BCPL. Le C a servi immédiatement (en 1972-1974 par Ken Thompson et Dennis Ritchie) à réimplanter le système **Unix** (dont le début du développement date de 1970). Les lois anti-trust empêcheront AT&T de commercialiser **Unix** et c'est ainsi que le C et **Unix** se répandront via les universités dans les années 1980. Le C est un langage très utilisé du fait de sa relative facilité d'utilisation, de son efficacité et de son adéquation au développement des systèmes d'exploitations (grâce en particulier à l'arithmétique sur les pointeurs et au typage faible). On dit parfois avec humour que c'est *un langage qui combine l'élégance et la puissance de l'assembleur avec la lisibilité et la facilité de maintenance de l'assembleur* !

Certaines des faiblesses du langage, en particulier des différences de plus en plus marquées entre ses nombreuses implantations et la nécessité d'une boîte à outils portable seront corrigées par la norme ANSI de 1985. Afin d'adapter le langage aux exigences de développement de grande envergure, son successeur, restant autant que possible compatible au niveau source, sera le C++, développé par Bjarne Stroustrup, avec une philosophie assez différente, mais en conservant la syntaxe.

1 Eléments de syntaxe

1.1 Structure générale d'un programme C

Pour respecter la tradition, voici le programme le plus simple écrit en C :

```
#include <stdio.h>
main() {
      printf("Je vous salue, O maitre.");
}
```

Un programme C est constitué de :

- **Directives du préprocesseur** qui commencent par un # et dont nous reparlerons par la suite.
  Ici `#include <stdio.h>` permet de disposer des fonctions d'entrée-sortie standard *STanDard Input/Output Header*, en particulier de la fonction `printf()` qui permet l'affichage à l'écran.
- **Déclarations globales** de type et de variables, ainsi que des **prototypes** de fonction, c'est-à-dire des déclarations de fonctions où ne figurent que le nom de la fonction, les paramètres et le type de valeur de retour, mais sans le corps de la fonction. Il n'y en a pas dans l'exemple ci-dessus.
- **Fonctions**. Tous les sous-programmes en C sont des fonctions. Ce que l'on appelle habituellement procédure (en Pascal par exemple) est ici une fonction qui ne retourne rien, ce qui se dit `void` en C. Le compilateur reconnaît une fonction à la présence des parenthèses qui suivent le nom de la fonction et à l'intérieur desquelles se trouve la liste des paramètres. Cette liste peut être vide comme c'est le cas ci-dessus pour `main()`.
- **Une fonction particulière** dont le nom est obligatoirement **main** qui est la fonction appelée par le système d'exploitation lors du lancement du programme. C'est la seule fonction dans cet exemple.

1.2 Un exemple complet

Voici un exemple plus complet montrant la structure générale, avec une variable globale `iglob` visible de partout, les prototypes des fonctions (indispensable [1] pour la fonction `somme()` dans le cas présent) et les fonctions. Ce programme calcule $5!$ d'une façon un peu détournée...

```
#include <stdio.h>

/* variables globales */
int iglob;

/* prototypes des fonctions */
int somme(int a, int b);
int produit(int a, int b);

/* fonctions */
int produit(int a, int b) {
     int sortie;

     if (a>1) {
       sortie = somme(produit(a-1, b) , b);
     } else {
       sortie = b;
     }
     return sortie;
}

int somme(int a, int b) {
     return a+b;
}
```

1. En fait, en l'absence de prototype, et si le texte de la fonction n'a pas été rencontré jusqu'alors, le compilateur considère qu'elle retourne un *int*, ce qui fonctionne ici... mais n'est pas recommandé.

```
/* programme principal */
main() {
    int j;
    j=1;
    for (iglob=1;iglob<=5;iglob++) {
        /* appel de la fonction produit() */
        j = produit(iglob, j);
     }
     printf("%d\n",j);
}
```

On remarquera dès à présent qu'une fonction est formée d'*un type*, du *nom de la fonction* suivi de la *liste des paramètres entre parenthèses* puis d'un **bloc** formant le corps de la fonction. On appelle bloc en C tout ensemble délimité par des accolades {} et comportant des déclarations de variables (il peut ne pas y en avoir) suivi d'expressions[2]. Il est toléré de ne pas faire figurer de type de retour pour une fonction (c'est le cas du `main()` ici), mais c'est une pratique fortement déconseillée.

Un **prototype** a la même écriture que la fonction, à cela près que le bloc du corps de la fonction est remplacé par un point-virgule. Lorsqu'il sera fait référence à des fonctions provenant d'une biliothèque dans la suite de ce document, celles-ci seront présentées par leur prototype.

2 La compilation sous Unix

Exemple élémentaire

Le compilateur C sous **Unix** fourni avec la machine s'appelle généralement **cc**. Néanmoins on utilise souvent **gcc** car il est présent partout, il supporte la norme ANSI ainsi que quelques extensions bien pratiques, il est libre et très efficace sur toutes les machines. C'est par ailleurs le seul compilateur C digne de ce nom disponible sur les systèmes d'exploitation libres (Linux, FreeBSD, NetBSD).

La compilation se fait dans le cas le plus simple par :

```
Idefix> gcc <source.c> -o <executable>
```

où il faut bien entendu remplacer les chaînes entre `<>` par les bons noms de fichiers.

Si des bibliothèques sont nécessaires, par exemple la bibliothèque mathématique, on ajoute à la ligne de commande précédente `-l<bibliotheque>`, (`-lm` pour la bibliothèque mathématique).

Options du compilateur *gcc*

gcc accepte de très nombreuses options (voir la page de manuel **man gcc**). En voici quelques unes fort utiles :

-g Compilation avec les informations utiles pour le déverminage (debug). Avec **gcc** on peut ainsi utiliser l'excellent déboggueur **gdb** (ou la version avec interface graphique **xxgdb**).

-ansi Force le compilateur à être conforme à la norme ANSI. Les extensions comme les **long long** sont néanmoins acceptées.

-pedantic Le compilateur refuse tout ce qui n'est pas strictement conforme à la norme ANSI. Ceci, combiné avec **-ansi** permet de vérifier qu'un programme est strictement conforme à la norme et donc en théorie portable.

-Wall Force l'affichage de tous les messages d'alerte lors de la compilation.

-O Optimisation de la compilation. Il vaut mieux ne l'utiliser que lorsque le programme fonctionne déjà de manière correcte. Des optimisations encore plus poussées sont possibles avec **-On**, où *n* est un entier entre 1 et 4 en général.
L'optimisation conduisant parfois à des déplacements d'instructions, il est déconseillé d'exécuter au dévermineur des programmes optimisés.

-I<rep> Précise le(s) répertoire(s) où il faut chercher les fichiers de **header** (**include**).

-D<var=val> Permet de définir des variables du préprocesseur. `-DDEBUG=2` est équivalent à mettre au début du source la ligne `#define DEBUG 2`. On peut également écrire `-DMONOPTION` qui est équivalent à `#define MONOPTION`[3].

2. C'est l'équivalent du **begin...end** du Pascal.

3. Il s'agit de directives du préprocesseur qui seront présentées en partie 6.

-S Le compilateur génère uniquement le listing assembleur (et non du code objet ou un exécutable), avec par défaut le suffixe **.s**. C'est utile pour optimiser efficacement de très petits morceaux de code critiques (ou par simple curiosité !).

-M Permet d'extraire des règles de dépendance (envoyées sur la sortie standard) pour le programme **make**, de façon similaire à **makedepend** (cf. page 8).

-p L'exécutable généré intègre des fonctions de mesure de temps passé, nombre d'appels... (*profiling*). Lors du lancement de cet exécutable, un fichier **gmon.out** est généré, et les résultats sont consultables ultérieurement avec **gprof** `<nom_de_l'exécutable>`.

De nombreuses autres options d'optimisation sont disponibles et peuvent influencer sensiblement la rapidité d'exécution dans le cas de programmes très calculatoires.

Compilation séparée

La programmation modulaire consiste à ne pas regrouper dans un seul fichier tout le code source d'un projet, mais au contraire à le répartir dans plusieurs fichiers et à compiler ces derniers séparément. Ceci permet une réutilisation simple du code car un ensemble de fonctions d'usage général peut être isolé dans un fichier et réutilisé facilement dans d'autres programmes. De plus, pour les gros développements logiciels, la compilation est une étape longue à chaque modification; il est donc souhaitable de séparer les sources en plusieurs petits fichiers, ainsi seuls les fichiers effectivement modifiés devront être recompilés à chaque fois.

Afin de rendre le source d'un projet le plus lisible possible, on adopte généralement quelques conventions simples. On distingue ainsi trois types de fichiers sources :

- Des fichiers contenant des fonctions qu'on nommera avec une extension **.c**, mais ne contenant en aucun cas une fonction `main()`.
- Des fichiers contenant les déclarations de type, les prototypes des fonctions (voir plus loin) et éventuellement des macros, correspondant aux sources précédentes. On les nommera comme les sources auxquelles ils correspondent mais avec un suffixe **.h**, et ils seront inclus par des directives `#include` dans les sources des fichiers **.c**. On appelle ces fichiers des **headers** (en-têtes).
- Des fichiers avec une extension **.c**, contenant un `main()` et utilisant les fonctions des autres fichiers (qui sont déclarées par des inclusions des **.h**). Chaque fichier de ce type donnera un exécutable du projet final. Il est recommandé d'adopter une convention claire de nommage de ces fichiers afin de les distinguer des premiers (par exemple les nommer `m_<nom_exécutable>.c` ou `main.c` s'il n'y en a qu'un).

La compilation d'un fichier source **.c** en un fichier objet ayant le même nom de base mais le suffixe **.o** se fait par une commande du type :

```
Idefix> gcc -c <source.c>
```

Rappelons qu'un fichier objet est en fait un fichier contenant le code compilé du source correspondant, mais où figurent également (entre autre) une table des variables et des fonctions exportées définies dans le source, ainsi qu'une table des variables et des fonctions qui sont utilisées mais non définies.

Les fichiers objets (**.o**) sont alors liés ensemble, c'est la phase *d'édition de liens*, où sont également ajoutées les bibliothèques (ensemble de fonctions précompilées). Dans cette phase, les tables de tous les fichiers objets sont utilisées pour remplacer les références aux variables et aux fonctions inconnues par les vrais appels. Le résultat est alors un fichier exécutable. Ceci se fait par une commmande du type :

```
Idefix> gcc <obj1.o> <obj2.o> ... <objn.o> -o <executable> -lm
```

Il convient de bien faire la différence entre la ligne `#include <math.h>` et l'option `-lm` sur la ligne de commande. En effet, le header **math.h** contient tous les prototypes des fonctions mathématiques, mais ne contient pas leur code exécutable. Le code permettant de calculer le logarithme par exemple se trouve en fait dans une bibliothèque précompilée. C'est l'option d'édition de lien `-lm` qui fournit la fonction elle-même en ajoutant au programme le code de calcul des fonctions de la bibliothèque mathématique.

> Les bibliothèques précompilées sont stockées sous **Unix** comme des fichiers *archive* portant l'extension **.a**[a]. Lors de l'utilisation de `-l<nom>`, le fichier recherché s'appelle en fait **lib**<**nom**>**.a**, et se trouve généralement dans **/usr/lib**. Il est donc possible mais fortement déconseillé de remplacer la ligne
>
> ```
> Idefix> gcc <obj1.o> <obj2.o> ... <objn.o> -o <executable> -lm
> ```
>
> par
>
> ```
> Idefix> gcc <obj1.o> <obj2.o> ... <objn.o> /usr/lib/libm.a -o <executable>
> ```

[a] Il existe également d'autre types de bibliothèques et le mécanisme réel d'édition de lien est un peu plus subtil que nous le présentons ici, néanmoins le comportement qualitatif reste le même.

Makefile

Garder trace des dépendances[4] des fichiers entre eux et réaliser manuellement les phases de compilation serait fastidieux. Un utilitaire générique pour résoudre ce genre de problèmes de dépendances existe : **make**. Son utilisation dépasse largement la compilation de programmes en C. La version utilisée ici est GNU make, accessible sous le nom **make** sous **Linux** et sous le nom **gmake** sous **SUN Solaris**.

make fonctionne en suivant des règles qui définissent les actions à effectuer. Il cherche par défaut ces règles dans un fichier nommé **Makefile** dans le répertoire courant. Voici un exemple simple qui permet de compiler les fichiers `fich.c` et `m_tst.c` en un exécutable **tst** :

```
## Makefile
# L'exécutable dépend des deux fichiers objets
tst : fich.o m_tst.o
	gcc fich.o m_tst.o -o tst     # ligne de compilation à exécuter

## Chaque fichier objet dépend du source correspondant
# mais également du header contenant les prototypes des fonctions
fich.o : fich.c fich.h
	gcc -c fich.c

m_tst.o : m_tst.c fich.h
	gcc -c m_tst.c
```

La première ligne d'une règle indique le nom du fichier concerné (appelé fichier cible) suivi de deux-points (`:`) et de la liste des fichiers dont il dépend. Si l'un de ces fichiers est plus récent[5] que le fichier cible, alors les commandes situées sur les lignes suivant la règle sont exécutées. Ainsi, si **tst** doit être fabriqué, la première règle nous dit que `fich.o` et `m_tst.o` doivent d'abord être à jour. Pour cela, **make** tente alors de fabriquer ces deux fichiers. Une fois cette tache effectuée (ce sont les deux autres règles qui s'en chargent), si le fichier nommé **tst** est plus récent que `fich.o` et `m_tst.o`, c'est terminé, dans le cas contraire, **tst** est refabriqué par la commande qui suit la règle de dépendance : `gcc fich.o m_tst.o -o tst`.

> **Attention** le premier caractère d'une ligne de commande dans un **Makefile** est un TAB (caractère de tabulation) et non des espaces.

Avec le **Makefile** ci-dessus, il suffit ainsi de taper **make tst** (ou même **make**, car **tst** est la première règle rencontrée) pour fabriquer l'exécutable **tst**. **make** n'exécute que les commandes de compilation nécessaires grâce au procédé de résolution des dépendances explicitée ci-dessus.

make peut servir à la génération automatique de fichiers d'impression à partir de sources LaTeX... Il peut permettre d'automatiser à peu près n'importe quelle séquence d'action systématique sur des fichiers. L'envoi par courrier électronique du projet en C est un exemple (à aménager si vous souhaitez l'utiliser) :

```
# Envoi automatique de mail pour le projet
# A remplacer par le bon nom
NOM=monnom
PROF=gueydan

# Les sources du projet
SRCS= Makefile interface.c calcul.c \
      fichiers.c m_projet.c

mail.txt: explication.txt $(NOM).tar.gz.uu
      cat explication.txt $(NOM).tar.gz.uu > mail.txt

mail: mail.txt
      mail -s IN203 gueydan < mail.txt
```

4. On dit qu'un fichier dépend d'un autre, si le premier doit être reconstruit lorsque le second change. Ainsi un fichier objet dépend de son source et de tous les headers inclus dans le source.

5. Les dépendances sont vérifiées à partir de la date de dernière modification de chaque fichier.

```
$(NOM).tar.gz.uu: $(SRCS)
      tar cvf $(NOM).tar $(SRCS)
      gzip -c $(NOM).tar > $(NOM).tar.gz
      uuencode $(NOM).tar.gz $(NOM).tar.gz > $(NOM).tar.gz.uu
```

Voici enfin un exemple générique complet pour compiler les sources `interface.c`, `calcul.c`, `fichiers.c` et `m_projet.c`. Les premières lignes de l'exemple sont des commentaires (commençant par #), et expliquent le rôle du projet.

Il est fait usage ici de l'utilitaire **makedepend** pour fabriquer automatiquement les dépendances dues aux directives `#include`. En effet, chaque fichier objet dépend d'un fichier source mais aussi de nombreux **headers** dont il est fastidieux de garder la liste à jour manuellement. **makedepend** parcourt le source à la recherche de ces dépendances et ajoute automatiquement les règles correspondantes à la fin du fichier *Makefile* (les règles ainsi ajoutées à la fin ne sont pas reproduites ci-dessous). **makedepend** est un outil faisant partie de X11, **gcc -M** peut jouer le même rôle mais son utilisation est un peu plus complexe.

L'usage de variables (*CFLAGS*, *LDFLAGS*...) permet de regrouper les différentes options au début du fichier *Makefile*. La variable contenant la liste des fichiers objets est obtenue automatiquement par remplacement des extensions *.c* par *.o*.

Des caractères spéciaux dans les règles permettent d'écrire des règles génériques, comme ici la génération d'un fichier objet ayant l'extension **.o** à partir du source correspondant. Un certain nombre de ces règles sont connues par défaut par **make** si elles ne figurent pas dans le fichier **Makefile**.

```
##############################################################################
# ECOLE    : ENSTA
# PROJET   : Cours IN261
# FONCTION : Compilation du projet
# CIBLE    : UNIX / make ou GNU make
# AUTEUR   : MERCIER Damien
# CREATION : Thu Sep 23 10:02:31 MET DST 1997 sur Obiwan
# MAJ      : Tue Jan 12 20:18:28 MET 1999 sur Idefix
# Version  : 1.02
##############################################################################
# Le compilateur utilisé
CC=gcc
# Options de compilation
CFLAGS=-I. -ansi -g -Wall
# Options d'édition de liens
LDFLAGS=-lm

# Le nom de l'exécutable
PROG = projet
# La liste des noms des sources (le caractere \ permet de passer
# a la ligne sans terminer la liste)
SRCS = interface.c calcul.c \
      fichiers.c m_projet.c

# La liste des objets
OBJS = $(SRCS:.c=.o)

##############################################################################
# Règles de compilation
##############################################################################
# Regle par defaut : d'abord les dependances, puis le projet
all: dep $(PROG)

# Ajout automatique des dependances entre les .c et les .h
#  a la fin de ce Makefile
dep:
```

```
      makedepend -- $(CFLAGS) -- $(SRCS)

# Comment compile t'on un .c en un .o ? (Regle generique)
%.o : %.c
      $(CC) $(CFLAGS) -c $*.c

# L'edition de liens
$(PROG) : $(OBJS)
      $(CC) $(CFLAGS) $(OBJS)  -o $(PROG) $(LDFLAGS)

# 'make clean' fait un nettoyage par le vide
clean:
      \rm -f $(PROG) $(OBJS) Makefile.bak *~ #* core
```

3 Types, opérateurs, expressions

3.1 Les types simples

Les types de base en C sont les nombres entiers (**int**), les nombres réels (**float**) et les caractères (**char**). Certains de ces types existent en plusieurs tailles :

char représente un caractère codé sur un octet (8 bits) ;

int désigne un entier codé sur la taille d'un mot machine, généralement 32 bits sur les machines actuelles ;

short (on peut aussi écrire **short int**) désigne un entier sur 16 bits ;

long (ou **long int**) désigne un entier sur 32 bits ;

float est un nombre flottant (réel) simple précision, généralement codé sur 32 bits ;

double est un flottant double précision codé le plus souvent sur 64 bit ;

long double est un flottant quadruple précision. Il n'est pas garanti que sa gamme soit supérieure à un **double**. Cela dépend de la machine.

Le mot clé **unsigned** placé avant le type permet de déclarer des nombres non-signés. De la même façon, **signed** permet de préciser que le nombre est signé. En l'absence de spécification, tous les types sont signés, à l'exception de **char** qui est parfois non signé par défaut[6]. Il convient de noter que le type **char** peut servir à stocker des nombres entiers et peut servir comme tel dans des calculs. La valeur numérique correspondant à un caractère donné est en fait son code ASCII.

Les tailles citées ci-dessus sont valables sur une machine *standard* dont le mot élémentaire est 16 ou 32 bits et pour un compilateur *standard* : autant dire qu'il est préférable de vérifier avant...

3.2 Déclarations des variables

Une déclaration de variable se fait systématiquement **au début d'un bloc** avant toute autre expression[7].

La déclaration s'écrit `<type> <nom>;` où `<type>` est le type de la variable et `<nom>` est son nom. On peut regrouper plusieurs variables de même type en une seule déclaration en écrivant les variables séparées par des virgules : `int a, b, i, res;` par exemple. Remarquer que ceci est différent des paramètres d'une fonction, qui doivent tous avoir un type explicite (on écrit `int somme(int a, int b);`).

Toute variable déclarée dans un bloc est locale à ce bloc, c'est-à-dire qu'elle n'est pas visible en dehors de ce bloc. De plus, elle masque toute variable venant d'un bloc qui l'englobe. Ceci peut se voir sur un exemple :

```
#include <stdio.h>
int a, b;

dummy(int a) {
    /* la variable a globale est ici masquee
       par le parametre a, en revanche b est globale */
```

6. Avec **gcc**, le type **char** est signé ou non selon les options de compilation utilisées sur la ligne de commande : **-fsigned-char** ou **-funsigned-char**.

7. Cette limitation devrait être levée dans la future norme ANSI.

```
    /* on a a=3 a l'appel 1 (provient du c du main()), b=2 (globale) */
    /* et a=5 a l'appel 2 (provient du d du sous-bloc), b=2 (globale) */

    /* c et d ne sont pas visibles ici */
    printf("%d %d\n",a,b);
}

main() {
    int c, d;
    a=1; b=2; c=3; d=4;
    dummy(c); /* appel 1 */
    /* on a toujours a=1, b=2 globales
      et c=3, d=4 locales */
    {         /* sous-bloc */
     int d, b;
     d=5; /* d est locale a ce bloc et masque le d de main() */
     b=7; /* b est locale a ce bloc */
     dummy(d); /* appel 2 */
    }
    printf("%d\n",d); /* on revient a d=4 du bloc main() */
}
```

Ce programme affiche dans l'ordre :

```
3  2
5  2
4
```

Attention, la simple déclaration d'une variable ne provoque généralement aucune initialisation de sa valeur. Si cela est néanmoins nécessaire, on peut écrire l'initialisation avec une valeur initiale sous la forme `int a=3;`.

Les classes de stockage

On appelle de ce nom barbare les options qui peuvent figurer en tête de la déclaration d'une variable. Les valeurs possibles sont les suivantes :

- auto : Il est *facultatif* et désigne des variables locales au bloc, allouées automatiquement au début du bloc et libérées en fin de bloc. On ne l'écrit quasiment jamais.
- register : Cette déclaration est similaire à `auto`, mais pour un objet accédé très fréquemment. Cela demande au compilateur d'utiliser si possible un registre du processeur pour cette variable. Il est à noter que l'on ne peut demander l'adresse mémoire d'une variable (par l'opérateur & présenté en partie 5.2) ayant cette spécification, celle-ci n'existant pas.
- static : Les variables non globales ainsi déclarées sont allouées de façon statique en mémoire, et conservent donc leur valeur lorsqu'on sort des fonctions et des blocs et que l'on y entre à nouveau. Dans le cas des variables globales et des fonctions, l'effet est différent : les variables globales déclarées statiques ne peuvent pas être accédées depuis les autres fichiers source du programme.
- extern : Lorsqu'un source fait appel à une variable globale déclarée dans un autre source, il doit déclarer celle-ci **extern**, afin de préciser que la vraie déclaration est ailleurs. En fait une variable globale doit être déclarée sans rien dans un source, et déclarée **extern** dans tous les autres sources où elle est utilisée. Ceci peut se faire en utilisant le préprocesseur (cf. 6).

En règle générale, il est préférable d'éviter l'utilisation de la classe **register** et laisser au compilateur le soin du choix des variables placées en registre.

Afin de simplifier la tâche de développement et pour limiter les risques d'erreurs, il est recommandé de déclarer avec le mot clé `static` en tête les variables et fonctions qui sont globales au sein de leur fichier, mais qui ne doivent pas être utilisées directement dans les autres sources.

En revanche, utiliser **static** pour des variables locales permet de conserver une information d'un appel d'une fonction à l'appel suivant de la même fonction sans nécessiter de variable globale. Ceci est à réserver à quelques cas très spéciaux : ne pas en abuser !

3.3 Types complexes, déclarations de type

Le C est un langage dont la vocation est d'écrire du code de très bas niveau (manipulation de bits), comme cela est nécessaire par exemple pour développer un système d'exploitation [8], mais il dispose également de fonctionnalités de haut niveau comme les définitions de type complexe.

Trois familles de types viennent s'ajouter ainsi aux types de base, il s'agit des **types énumérés**, des **structures** et des **unions**. Ces trois types se déclarent de façon similaire.

Le type énuméré est en fait un type entier déguisé. Il permet en particulier de laisser des noms littéraux dans un programme pour en faciliter la lecture. Les valeurs possibles d'un type énuméré font partie d'un ensemble de constantes portant un nom, que l'on appelle des énumérateurs. L'exemple suivant définit un type énuméré correspondant à des booléens, noté **bool**, et déclare en même temps une variable de ce type nommée **bvar** :

```
enum bool {FAUX, VRAI} bvar;
```

Après cette déclaration il est possible de déclarer d'autres variables de ce même type avec la syntaxe `enum bool autrevar;`.

> Dans la première déclaration, il est permis d'omettre le nom du type énuméré (dans ce cas seule la variable **bvar** aura ce type) ou bien de ne pas déclarer de variable sur la ligne de déclaration d'énumération.

Dans une énumération, si aucune valeur entière n'est précisée (comme c'est le cas dans l'exemple précédent), la première constante vaut zéro, la deuxième 1 et ainsi de suite. La ligne ci-dessus est donc équivalente à (sans nommage du type)

```
enum {FAUX=0, VRAI=1} bvar;
```

Lorsque'elles sont précisées, les valeurs peuvent apparaître dans n'importe quel ordre :

```
enum {lun=1,ven=5,sam=6,mar=2,mer=3,jeu=4,dim=7} jour;
```

Une structure est un objet composé de plusieurs membres de types divers, portant des noms. Elle permet de regrouper dans une seule entité plusieurs objets utilisables indépendemment. Le but est de regrouper sous un seul nom plusieurs éléments de types différents afin de faciliter l'écriture et la compréhension du source.

Une structure **pixel** contenant les coordonnées d'un pixel et ses composantes de couleur pourra par exemple être :

```
struct pixel {
   unsigned char r,v,b;
   int x,y;
};
```

Par la suite, la déclaration d'une variable **p1** contenant une structure **pixel** se fera par `struct pixel p1;`.

> La remarque précédente concernant les possibilités de déclaration des énumérations s'applique également aux structures et aux unions.

Les accès aux champs d'une structure se font par l'opérateur **point** (`.`) . Rendre le point **p1** blanc se fera ainsi par `p1.r = 255; p1.v = 255; p1.b = 255;`

Une union est un objet qui contient, selon le moment, un de ses membres et un seul. Sa déclaration est similaire à celle des structures :

```
union conteneur {
   int vali;
   float valf;
   char valc;
};
union conteneur test;
```

En fait, il s'agit d'un objet pouvant avoir plusieurs types selon les circonstances.

On peut ainsi utiliser `test.vali` comme un **int**, `test.valf` comme un **float** ou `test.valc` comme un **char** après la déclaration précédente, mais un seul à la fois !

La variable **test** est juste un espace mémoire assez grand pour contenir le membre le plus large de l'union, et tous les membres pointent sur la même adresse mémoire. Il incombe donc au programmeur de savoir quel type est stocké dans l'union lorsqu'il l'utilise : c'est une des raisons qui rendent sont usage peu fréquent.

8. C à été développé au départ comme le langage principal du système **Unix**.

> **Attention**, il est fortement déconseillé d'utiliser la valeur d'un membre d'une union après avoir écrit un autre membre de la même union (écrire un **int** et lire un **char** par exemple) car le résultat peut différer d'une machine à l'autre (et présente en général peu d'intérêt, sauf si l'on veut savoir l'ordre que la machine utilise pour stocker les octets dans un **long**...).

Déclaration de type

Celle-ci se fait au moyen du mot-clé **typedef**. Un nouveau type ainsi défini peut servir par la suite pour déclarer des variables exactement comme les types simples. On peut ainsi écrire

```
typedef unsigned char uchar;
```

afin d'utiliser **uchar** à la place de `unsigned char` dans les déclarations de variables...

La déclaration de type est très utile pour les énumérations, les structures et les unions. Dans notre exemple précédent, il est possible d'ajouter

```
typedef struct pixel pix;
```

pour ensuite déclarer `pix p1,p2;`. Une version abrégée existe également dans ce cas et permet en une seule fois de déclarer la sructure **pixel** et le type **pix** :

```
typedef struct pixel {
   unsigned char r,v,b;
   int x,y;
} pix;
```

Le comportement est alors exactement le même que ci-dessus : les déclarations `struct pixel p1;` et `pix p1;` sont équivalentes.

Il est possible aussi de supprimer complètement le nom de structure dans ce cas en écrivant :

```
typedef struct {
   unsigned char r,v,b;
   int x,y;
} pix;
```

Comme précédemment, les déclarations de variables ultérieures se feront par `pix p1;`.

> Mais **attention**, cette dernière écriture ne permet pas de réaliser les structures **autoréférentielles** qui sont présentées au 5.2.

3.4 Ecriture des constantes

Les constantes entières sont écrites sous forme décimale, octale (base 8) ou hexadécimale (base 16). L'écriture décimale se fait de façon naturelle, l'écriture octale s'obtient en faisant précéder le nombre d'un **0** (par exemple 013 vaut en décimal 11), et l'écriture hexadécimale en faisant précéder le nombre du préfixe **0x**.

De plus, il est possible d'ajouter un suffixe **U** pour **unsigned** et/ou **L** pour **long** dans le cas où la valeur de la constante peut prêter à confusion. Ainsi **0x1AUL** est une constante de type **unsigned long** et valant 26.

Les constantes de type caractère peuvent être entrées sous forme numérique (par leur code ASCII, car du point de vue du compilateur ce sont des entiers sur un octet), il est aussi possible de les écrire entre apostrophes (e.g. **'A'**). Cette dernière écriture accepte également les combinaisons spéciales permettant d'entrer des caractères non-imprimables sans avoir à mémoriser leur code ASCII (voir au paragraphe 5.5). Le caractère `'\0'` vaut la **valeur** 0 (appelée NUL en ASCII) et sert de marqueur spécial[9].

Il faut bien remarquer que le type caractère correspond à un caractère unique (même si certains caractères sont représentés par des chaînes d'échappements comme `'\n'`) et non à une chaîne de caractères.

Les constantes flottantes (qui correspondent à dex nombres réels) peuvent utiliser une syntaxe décimale ou une notation scientifique avec la lettre `e` pour marquer l'exposant.

Il est possible d'affecter une valeur à une variable lors de sa déclaration. Ceci se fait le plus simplement possible par `int a=3;`. Dans le cas où cette variable est en fait une constante (c'est un comble !), c'est-à-dire qu'elle ne peut et ne doit pas être modifiée dans le corps du programme, on peut préciser **const** devant la déclaration : `const float e2 = 7.3890561`.

9. Le chiffre '0' vaut en fait en ASCII 48

Cette même syntaxe d'affectation initiale peut se faire pour les structures, les énumérations et les tableaux. Pour les structures, les valeurs initiales doivent être écrites entre accolades, séparées par des virgules et dans l'ordre de la structure. Il est possible d'omettre les derniers champs de la structure, qui prennent alors par défaut la valeur zéro :

```
typedef struct {
   unsigned char r,v,b;
   int x,y;
} pix;

pix p1={2, 3, 4, 5, 6}; /* r=2 v=3 b=4 x=5 y=6 */
pix p2={1, 2};          /* r=1 v=2 b=0 x=0 y=0 */
```

const sert également à préciser que certains arguments de fonction (dans la déclaration, et s'il s'agit de pointeurs, cf. 5.2) désignent des objets que la fonction ne modifiera pas [10].

3.5 Le transtypage (*cast*)

Lors de calculs d'expressions numériques, les types flottants sont dits **absorbants** car la présence d'un unique nombre flottant dans une expression entraîne une évaluation complète sous forme de flottant.

Il est fréquent qu'une variable ait besoin d'être utilisée avec un autre type que son type naturel. Cela arrive par exemple si l'on veut calculer le quotient de deux entiers et stocker le résultat dans un réel (voir le paragraphe suivant).

Ce transtypage s'obtient en faisant précéder l'expression que l'on souhaite utiliser avec un autre type par ce type entre parenthèses. C'est par exemple le seul moyen de convertir un nombre flottant en entier, car une fonction `floor()` est bien définie dans la bibliothèque mathématique, mais elle retourne un nombre flottant ! On écrira ainsi `i = (int)floor(f);`.

> Le transtypage d'entier en flottant et réciproquement comme présenté ci-dessus est un cas très particulier, qui pourrait laisser croire que toutes les conversions sont possibles par ce biais. Attention donc, **dans la grande majorité des cas**, le transtypage conduit juste à interpréter le contenu de la mémoire d'une autre façon : c'est le cas par exemple si l'on essaie de transformer une chaîne de caractère `char s[10]` en un entier par `(int)s`, et le résultat n'a pas d'intérêt (en fait cela revient à lire l'adresse mémoire à laquelle est stockée `s` !). Pour effectuer les conversions, il convient d'utiliser les fonctions adéquates de la bibliothèque standard.

3.6 Les opérateurs

Les opérations arithmétiques en C sont d'écriture classique : `+, -, *, /` pour l'addition, la soustration, la multiplication et la division. Il est à noter que la division appliquée à des entiers donne le quotient de la division euclidienne. Pour forcer un résultat en nombre flottant (et donc une division réelle), il faut convertir l'un des opérandes en nombre flottant. Par exemple `3/2` vaut **1** et `3/2.0` ou `3/(float)2` vaut [11] **1.5**.

Les affectations s'écrivent avec le signe `=`. Une chose importante est à noter : une affectation peut elle-même être utilisée comme un opérande dans une autre opération, la valeur de cette affectation est alors le résultat du calcul du membre de droite. Ainsi on pourra par exemple écrire `a=(b=c+1)+2;` dont l'action est de calculer `c+1`, d'affecter le résultat à `b`, d'y ajouter 2 puis d'affecter le nouveau résultat à `a`. Il est à noter que ce type d'écriture, sauf cas particuliers, rend la lecture du programme plus difficile et doit donc être si possible évité.

Les opérations booléennes permettent de faire des tests. Le C considère le `0` comme la valeur **fausse** et **toutes** les autres valeurs comme vraies. Pour combiner des valeurs en tant qu'expression booléenne, on utilise le OU logique noté `||` et le ET logique noté `&&`. La négation d'une valeur booléenne s'obtient par `!` placé avant la valeur. Il est important de retenir également que *les opérateurs logiques* `&&` *et* `||` *arrêtent l'évaluation de leurs arguments dès que cela est possible* [12]. Cette propriété peut être intéressante, par exemple dans des conditions de fin de boucle du type SI *mon indice n'a pas dépassé les bornes* ET *quelque chose qui dépend de mon indice* ALORS *instructions*. En effet dans ce cas le calcul qui dépend de l'indice n'EST PAS effectué si l'indice a dépassé les bornes, ce qui est préférable !

Il reste à engendrer des grandeurs booléennes qui représentent quelque chose ! Ceci ce fait avec **les opérateurs de comparaison**. Le test d'égalité se fait par `==` (deux signes égal), le test d'inégalité (différent de) se fait par `!=`, la relation d'ordre par `>`, . `>=`, `<` et `<=`.

10. L'usage de **const** est particulièrement important pour les systèmes informatiques embarqués, où *le code exécutable et les constantes* peuvent être stockés en mémoire morte (ROM).

11. Il s'agit ici d'un transtypage (*cast*).

12. Ceci s'appelle de *l'évaluation paresseuse* (lazy evaluation).

Attention une erreur classique est d'écrire un seul signe égal dans un test. Comme le résultat d'une affectation est la valeur affectée et comme un nombre non nul est un booléen vrai, écrire :

```
if (a == 2) {
      printf("a vaut 2\n");
}
```

ou écrire :

```
if (a = 2) {
      printf("a vaut 2\n");
}
```

est très différent ! (la deuxième version est un test toujours vrai, et a vaut 2 après l'exécution).

Les opérations d'incrémentation et de décrémentation étant fréquentes sur les entiers, il est possible (et même recommandé) d'écrire `i++` plutôt que `i=i+1`. Comme dans le cas des affectations, l'opération d'incrémentation à une valeur, qui est la valeur avant incrémentation si le `++` est à droite de la variable, et est la valeur incrémentée si c'est `++i`. Un opérateur de décrémentation noté `--` existe également. En résumé :

```
            /* valeurs apres la ligne */
i = 2;      /* i=2 */
a = ++i;    /* i=3, a=3 */
b = a--;    /* b=3, a=2 */
b++;        /* b=4 */
```

On remarquera que, bien que l'opérateur ait une valeur de retour, celle-ci peut être ignorée. Ceci est vrai également pour les fonctions. Par exemple `printf()` que nous avons utilisé tout au début retourne normalement un `int`, mais le plus souvent on ignore cette valeur de retour.

Dans la lignée des opérateurs d'incrémentation, il existe tout une famille **d'opérateurs de calcul-affectation** qui modifient une variable en fonction d'une expression. Leur syntaxe est de la forme `<var.> R= <expr.>` où `R` est une opération parmi `+`, `-`, `*`, `/`, `%`, `>>`, `<<`, `&`, `|` et `^`. L'écriture précédente est alors équivalente à :
`<var.> = <var.> R <expr.>`.

Quelques opérateurs spéciaux viennent s'ajouter à cela : nous avons déjà parlé des fonctions, **l'appel de fonction** est un opérateur (c'est même l'opérateur de priorité maximale). Il y a également l'opérateur **conditionnel** dont la valeur résultat provient d'une des deux expressions qu'il contient selon une condition :
`<Condition> ? <Expression si vraie> : <Expression si fausse>`
La condition est une expression booléenne. Elle est systématiquement évaluée (avec un arrêt dès que le résultat est certain comme nous l'avons vu précédemment) et selon sa valeur, une des deux expressions et une seulement est évaluée. Un exemple d'usage est la construction d'une fonction MIN :

```
int min(int a, int b) {
  return ( (a>b) ? b : a );
}
```

Un dernier opérateur souvent passé sous silence est **l'opérateur virgule**. Il permet de séparer une suite d'expressions qui seront nécessairement évaluées l'une après l'autre de la gauche vers la droite, le résultat final de cette séquence d'expression étant le résultat de la dernière évaluation (expression la plus à droite).

Les opérateurs concernant les pointeurs et les tableaux seront présentés en partie 5.

Les opérateurs sur les bits d'un mot sont extrêmement utiles en programmation bas-niveau et permettent généralement d'éviter la programmation en langage machine. Les opérations agissent bit à bit sur la représentation binaire de leur(s) opérande(s). On dispose ainsi de la complémentation (`~`) de tous les bits d'un mot, du ET bit à bit, du OU et du XOR (OU eXclusif) notés respectivement `&`, `|` et `^`. Les opérateurs de décalage à gauche (équivalent à une multiplication par deux sur des entiers non signés) et à droite (division par 2) se notent `<<` et `>>`. Quelques exemples :

```
unsigned char c, d;
c=0x5f;      /* en binaire 0101 1111 */
d = ~c;      /* d = 0xA0   1010 0000 */
d = d | 0x13; /*    0x13 = 0001 0011   -> d = 0xB3  1011 0011 */
c = d ^ (c << 2);  /* c decale a gauche 2 fois :    0111 1100 */
                                  resultat c= 0xCF 1100 1111 */
```

Attention une autre erreur classique consiste à oublier un signe et à écrire `cond1 & cond2` au lieu de `cond1 && cond2` dans un test. Le compilateur ne peut bien sûr rien remarquer et l'erreur est généralement très difficile à trouver.

3.7 L'appel de fonction

En C, l'appel de fonction se fait par le nom de la fonction suivi de la liste des paramètres entre parenthèses. Une particularité très importante du langage est que les paramètres passés en argument de la fonction sont recopiés dans des variables locales à la fonction avant le début de son exécution. On appelle cela un **passage par valeur**. Ainsi, sans utiliser les pointeurs [13], le seul résultat qui puisse sortir d'un appel de fonction est sa valeur de retour.

La fonction se termine lorsque l'exécution atteint l'accolade fermante du bloc, et dans ce cas il n'y a pas de valeur de retour (la fonction doit être déclarée de type `void`). Si la fonction doit retourner une valeur, celle-ci doit être précisée par l'instruction `return` [14]. L'exécution de cette instruction termine la fonction. Il est possible d'avoir plusieurs instructions `return` dans une même fonction, par exemple dans le cas de tests.

Dans le cas d'une fonction retournant `void` (sans valeur de retour), `return ;` permet de terminer la fonction.

Voici un exemple :

```
#include <stdio.h>
int test(int a, int b) {  /* lors de l'appel, a=2 et b=7 */
   if (a==0) return b;
   /* la suite n'est execute que si a != 0 */
   a=b/a;
   b=a;
   return a;
}
main() {
   int x, y, z;
   x=2; y=7;
   z=test(x,y);  /* z=3 mais x et y ne sont pas modifies */
                 /* seule leur valeur a ete copiee dans a et b de test() */
   printf("%d %d %d %d\n",x,y,z,test(0,z));  /* affiche 2 7 3 3 */
}
```

Les fonctions peuvent bien sûr être récursives, comme le montre l'exemple du 1.2.

3.8 Les fonctions *inline*

L'écriture de programmes lisibles passe par le découpage des algorithmes en de nombreuses fonctions, chacune formant une unité logique. Certaines de ces fonctions ne servent qu'une fois, mais on préfère ne pas intégrer leur code à l'endroit où elles sont appelées. A l'inverse, d'autres fonctions sont très simples (quelques lignes de programme, comme par exemple le calcul du minimum de deux valeurs) et sont utilisées un grand nombre de fois.

Dans les deux cas, chaque appel de fonction est une perte de temps par rapport à écrire le code directement à l'endroit de l'appel (le compilateur sauvegarde le contexte au début de l'appel de fonction, place éventuellement les arguments sur la pile, puis appelle la fonction) le retour de l'appel est également assez coûteux.

Pour éviter ce gâchis de puissance de calcul, la plupart des compilateurs essayent d'optimiser en plaçant le code de certaines fonctions directement à la place de leur appel. Ceci a de nombreuses limitations : il faut compiler avec des optimisations fortes et les fonctions concernées doivent être dans le même fichier source [15]

Pour aider le compilateur dans le choix des fonctions à optimiser ainsi, il est possible de préciser le mot-clef **inline** au début de la déclaration de la fonction. Ceci prend alors effet même si les optimisations ne sont pas demandées (ce n'est qu'une indication, comme la classe *register* pour les variables).

Ce mot-clef **inline** ne fait pas partie de la norme ANSI actuelle, mais est supporté par **gcc** et de nombreux autres compilateurs, et devrait faire partie de la nouvelle norme.

13. Voir 5.2. Ceux-ci permettent quelque chose de similaire au passage par variable (ou par référence) du Pascal.

14. On peut indifféremment écrire `return n;` ou `return(n);`, c'est une affaire de goût. La deuxième écriture ayant l'avantage de donner un petit air de fonction à `return`.

15. Ceci n'empêche pas la compilation séparée, mais tout appel de fonction entre des fichiers objets différents est obligatoirement effectué sous la forme d'un vrai appel de fonction. A cette fin, le compilateur conserve toujours une version de la fonction appelable *normalement.*

Pour permettre l'appel de fonction depuis d'autres fichiers objet[a], le compilateur conserve tout de même une version sous forme de fonction classique, sauf si la fonction est de plus déclarée **static inline**.

[a] C'est également nécessaire si la fonction est appelée quelque part par pointeur de fonction, cf. 5.6.

On pourra avec avantage déclarer ainsi une fonction minimum de deux entiers sous la forme de fonction inline (plutôt que de macro, ce qui conduit à de fréquentes erreurs) :

```
static inline int min(int a, int b) {
  return ( (a>b) ? b : a );
}
```

Ces fonctions déclarées **static inline** sont les seules fonctions que l'on pourra écrire dans les fichiers **.h** afin de pouvoir les utiliser comme on le fait avec des macros.

3.9 Les priorités

Voici un tableau récapitulatif donnant tous les opérateurs et leur priorité (la première ligne du tableau est la plus prioritaire, la dernière est la moins prioritaire), les opérateurs ayant des priorités égales figurent dans une même ligne et sont évalués dans le sens indiqué dans la colonne associativité :

Opérateurs	Fonction	Associativité	priorité
`()`	appel de fonction	→	maximale
`[]`	élément de tableau	→	
`.`	champ de structure ou d'union	→	
`->`	champ désigné par pointeur	→	
`!`	négation booléen	←	
`~`	complémentation binaire	←	
`-`	opposé	←	
`++`	incrémentation	←	
`--`	décrémentation	←	
`&`	adresse	←	
`*`	déréférenciation de pointeur	←	
`(type)`	transtypage	←	
`*`	multiplication	→	
`/`	division	→	
`%`	reste euclidien	→	
`+`	addition	→	
`-`	soustraction	→	
`<<`	décalage à gauche	→	
`>>`	décalage à droite	→	
`<`	strictement inférieur	→	
`<=`	inférieur ou égal	→	
`>`	strictement supérieur	→	
`>=`	supérieur ou égal	→	
`==`	égal	→	
`!=`	différent	→	
`&`	ET bit à bit	→	
`^`	OU eXclusif (XOR) bit à bit	→	
`|`	OU bit à bit	→	
`&&`	ET booléen	→	
`||`	OU booléen	→	
`? :`	conditionnelle	←	
`=`	affectation	←	
`*= /= %=`	affectations avec calcul	←	
`+= -= <<=`	affectations avec calcul	←	
`>>= &= |= ^=`	affectations avec calcul	←	
`,`	séquence d'expressions	→	minimale

Les opérateurs concernant les pointeurs sont présentés au 5.

Ce tableau est très important, et son ignorance peut conduire parfois à des erreurs difficiles à détecter. Citons par example `a=*p++` : les opérateurs `++` et `*` sont de même priorité, donc l'expression est évaluée dans l'ordre d'associativité (de droite à gauche), et donc p est incrémenté, et l'ancienne valeur de p est déréférencée, le résultat étant stocké dans a. Si l'on veut incrémenter la valeur pointée par p (et non p lui-même), il faudra écrire `a=(*p)++`. Le résultat stocké dans a est le même dans les deux cas.

En cas de doute, il ne faut pas hésiter à mettre quelques parenthèses (sans abuser toutefois) car cela peut aider à la compréhension. On évitera ainsi d'écrire : `a=++*p>b|c+3` !
On évitera aussi : `a=((++(*p)>b)|(c+3))` et on préfèrera quelque chose comme : `a = ++(*p)>b | (c+3)`.

4 Tests et branchements

4.1 La conditionnelle

Nous avons vu lors de l'examen des opérateurs l'expression conditionnelle. Il existe également bien sûr une structure de contrôle qui réalise l'exécution conditionnelle. Sa syntaxe est la suivante :

```
if (<condition>) {
   <instructions ...>
}
```

La condition est testée comme un booléen, c'est-à-dire qu'elle est considérée comme vraie si son évaluation n'est pas nulle. Dans le cas où une seule instruction est à exécuter quand la condition est vraie, on trouvera souvent cette instruction toute seule à la place du bloc entre accolades.
La version avec alternative :

```
if (<condition>) {
   <instructions ...>
} else {
   <instructions ...>
}
```

Là encore il est possible de remplacer l'un ou l'autre (ou les deux) blocs entre accolades par une unique instruction. Attention dans ce cas, il faut terminer cette instruction par un point-virgule.
Avec de multiples alternatives :

```
if (<cond1>) {
   <instructions ...> /* Si cond1 est vraie */
} else if (<cond2>) {
          <instructions ...> /* si cond1 fausse et cond2 vraie */
      } else {
          <instructions ...> /* si cond1 et cond2 fausses */
      }
}
```

De nombreux programmes C utilisent des tests sans instruction de comparaison explicite, voire des tests qui sont le résultat d'opérations d'affectation, par exemple :

```
if (a = b - i++) i--; else a=b=i++;
```

Ceci utilise le fait qu'une valeur non nulle est considérée comme vraie. C'est assez difficile à lire, aussi il sera préférable [16] d'écrire cette version équivalente :

```
i++;
a= b-i;
if (a != 0) {
     i--;
} else {
     i++;
     b = i;
     a = b;
}
```

16. A moins que l'on souhaite battre le record de l'écriture du jeu de tetris en un minimum de lignes !

4.2 Le *switch*

```
switch (<expression>) {
   case <val 1>:  <instructions ...>
   case <val 2>:  <instructions ...>
   ...
   case <val n>:  <instructions ...>

   default:  <instructions ...>
}
```

L'instruction `switch` permet de diriger le programme vers une séquence d'instructions choisie parmi plusieurs, suivant la valeur d'une expression qui doit être de type entier.

Le branchement se fait à l'étiquette `case` dont la valeur est égale à l'expression de tête (il ne doit y avoir au plus qu'une étiquette qui correspond). Si aucune étiquette ne correspond, le saut se fait à l'étiquette `default` si elle existe.

Attention, une fois le branchement effectué, le programme continue de s'exécuter dans l'ordre, y compris les `case` qui peuvent suivre. Pour empêcher ce comportement, on utilise souvent `break` à la fin des instructions qui suivent le `case` : son effet est de sauter directement après le bloc du `switch` (Voir 4.4).

Un exemple complet d'utilisation standard :

```
int val,c;
c=1;
val = mafonctiondecalcul(); /* une fonction définie ailleurs */
switch (val) {
    case 0:
    case 1: printf("Résultat 0 ou 1\n");
            c=2;
            break;
    case 2: printf("Résultat 2\n");
            break;
    case 4: printf("Résultat 4\n");
            c=0;
            break;
    default: printf("Cas non prévu\n");
}
```

4.3 Les boucles

La boucle `for`

C'est une boucle beaucoup plus générale que dans d'autres langages. Son rôle ne se limite pas à la simple répétition en incrémentant un entier. Son utilisation est la suivante :

```
for ( <inst. initiale> ; <cond. de poursuite> ; <inst. d'avancement> ) {
    <instructions ...>
}
```

L'*instruction initiale* est exécutée avant le début de la boucle, la *condition de poursuite* est testée à chaque début de boucle et l'*instruction d'avancement* est exécutée en dernière instruction à chaque tour de boucle.

L'équivalent de la boucle d'incrémentation d'une variable entière des autres langages s'écrira par exemple :

```
for (i=0; i<100; i++) {
    <instructions...> /* exécutées pour i=0, i=1 ... i=99 */
}
```

On peut omettre n'importe laquelle des trois expressions de la boucle **for**. Si c'est la condition de poursuite qui est omise, la boucle se comporte comme si cette condition était toujours vraie. Ainsi une boucle infinie[17] peut s'écrire :

```
for (;;) {
    <instructions ...>
}
```

17. Qui ne se termine jamais si l'on utilise pas les instructions de branchement qui seront présentées au 4.4.

La boucle `while`

```
while (<condition de poursuite>) {
    <instruction ...>
}
```

qui est complètement équivalente à (remarquer que les expressions de la boucle **for** sont facultatives) :

```
for (;<condition de poursuite>;) {
    <instructions ...>
}
```

De façon similaire, il est possible de réécrire une boucle `for` générique sous la forme d'une boucle `while`. L'écriture générique précédente est équivalente à :

```
<inst. initiale>;
while (<cond. de poursuite>) {
    <instructions...>
    <inst. d'avancement>;
}
```

La boucle `do...while`

Ici le test est effectué en fin de boucle. Cette boucle est donc, contrairement aux deux boucles précédentes, exécutée au moins une fois, même si la condition est fausse dès le début de la boucle. Voici la syntaxe :

```
do {
    <instructions ...>
} while (<condition de poursuite>);
```

4.4 Les branchements

Ce sont des instructions qui modifient le déroulement d'un programme sans condition.

L'instruction **continue** ne peut être présente qu'à l'intérieur d'une boucle. Elle conduit à un branchement vers la fin de la plus petite boucle qui l'entoure, mais ne sort pas de la boucle (ainsi l'*instruction d'avancement* si elle existe est effectuée, puis la boucle reprend).

L'instruction **break** peut apparaître dans une boucle ou dans une instruction `switch`. Elle effectue un saut immédiatement après la fin de la boucle ou du `switch`. Son usage est indispensable dans le cas des boucles infinies.

L'instruction **goto** <**label**> quant à elle peut apparaître n'importe où. Elle effectue un saut à l'instruction qui suit l'étiquette `<label>:` (cette étiquette se note par le label suivi de deux-points).

L'usage du **goto** est fortement déconseillé car il conduit fréquemment à un source illisible. Il peut dans quelques rares cas rendre service en évitant d'immenses conditions, ou de nombreuses répétitions d'un même petit morceau de code.

L'exemple suivant montre l'utilisation de **break** et de **continue** au sein d'une boucle `for`. On trouvera plus fréquemment l'instruction **break** dans des boucles infinies.

```
for (i=0; i<100; i++) {
    if (i<5) continue; /* si i<5 la fin de la boucle n'est pas exécutée */
    if (i==10) break; /* si i vaut 10 la boucle se termine */
    printf("%d\n",i);
}
```

Le résultat de cette boucle est l'affichage des nombres **5, 6, 7, 8, 9**. La condition de poursuite de la boucle `for` est en fait totalement inutile ici.

5 Tableaux et pointeurs

5.1 Déclaration et utilisation des tableaux

En C, un tableau à une dimension de 10 entiers sera déclaré par la syntaxe **int x[10]**. Il s'agit d'une zone de mémoire de longueur dix entiers qui est allouée automatiquement, `x` étant comme expliqué ci-dessous un pointeur constant vers le premier entier de cette zone.

Les dix cellules du tableau sont alors accessibles par **x[0]** à **x[9]**. Il convient de noter qu'**aucune vérification n'est faite sur la validité du numéro de la case demandée** du tableau. L'accès à une case au delà du dernier élément du tableau (ou avant le premier, par exemple **x[-1]**) peut conduire à une erreur, mais il se peut aussi que cela accède à d'autres variables du programme et dans ce cas, l'erreur est très difficile à détecter...

Les tableaux de plusieurs dimensions existent également, par exemple `char pix[128][256]`. L'accès aux cases de ce tableau se fait alors par `pix[i][j]`, `i` et `j` étant des entiers. Il faut voir ce type d'écriture comme l'accès à un tableau à une seul dimension de 128 éléments qui sont chacuns des tableaux de 256 caractères.

L'affectation de valeurs initiales dans un tableau à une dimension se fait au moyen d'une liste de valeurs entre accolades :

```
int a[10]={1,2,3,4};
```

Dans cet exemple, `a[0]`, `a[1]`, `a[2]` et `a[3]` sont initialisés, `a[4]` à `a[9]` ne le sont pas. On peut omettre d'écrire la taille du tableau, le compilateur la calcule alors en comptant le nombre d'éléments fournis pour l'initialisation : `int a[]={1,2,3,4}` conduit alors à un tableau de quatre éléments seulement. Pour les tableaux de plusieurs dimensions, le dernier indice correspond au nombre de colonnes du tableau :

```
int b[2][4]={{1,2,3,4}, {5,6,5,6}}; /* 2 lignes, 4 colonnes */
```

Dans ce cas, seul le premier indice peut être omis, le compilateur comptant qu'il y a deux lignes, mais ne comptant pas les colonnes sur chaque ligne. Il est même possible dans ce cas de ne pas assigner tous les éléments de chaque ligne ! Ainsi

```
int b[][4]={{1,2}, {5,6,5}};
/* 2 lignes (comptees par le compilateur), 4 colonnes */
```

conduit à un tableau de deux lignes et quatre colonnes dont seuls les deux premiers éléments de la première ligne sont affectés, ainsi que les trois premiers éléments de la deuxième ligne [18].

5.2 Les pointeurs

Un pointeur est en fait une variable spéciale dans laquelle on ne stocke pas la valeur que l'on désire conserver, mais seulement son adresse en mémoire. En C un *pointeur* sur un objet de type `typ` est une variable de type noté `typ *`. Ainsi la déclaration `int *x;` définit **x** comme un *pointeur* sur une valeur de type **int**, c'est-à-dire comme l'adresse mémoire d'un entier.

L'accès au contenu d'un pointeur se fait par déréférenciation avec l'opérateur étoile. La valeur pointée par `x` peut donc être accédée par l'expression ***x**.

Un pointeur est une variable qui a sa propre adresse, et qui contient l'adresse de la variable sur laquelle il pointe. Ainsi, la déclaration `int *x;` alloue la case mémoire correspondant au pointeur `x`, mais n'initialise pas sa valeur. Or cette valeur doit être l'adresse de l'entier pointé par `*x`. En conséquence, la déréférenciation de `x` provoquera probablement une erreur (ou un résultat stupide...) tant que l'on n'a pas écrit explicitement dans le pointeur une adresse valide.

Pour initialiser la valeur du pointeur `x`, il faut ainsi utiliser une adresse mémoire valide, par exemple, celle d'une autre variable. A cet effet, l'adresse d'une variable s'obtient par l'opérateur `&`. Ainsi si l'on a déclaré `int y,*x;` on peut écrire `x = &y;`. Le pointeur `x` pointe alors sur la case mémoire correspondant à la valeur de la variable `y`. Tant que le pointeur `x` n'est pas modifié, `*x` et `y` ont toujours la même valeur (c'est la même case mémoire). On peut voir `*` comme l'opérateur inverse de `&`. On a ainsi `*(&y)` qui vaut `y`, et de même `&(*x)` qui vaut `x`.

> Un pointeur est en fait une variable presque comme les autres. Il s'agit d'un entier de la taille d'une adresse mémoire de la machine (le plus souvent 32 bits). La variable pointeur `x` de notre exemple précédent est un entier de signification spéciale. Il est donc possible de prendre l'adresse d'un pointeur ! Le pointeur `&x` est alors un pointeur vers un pointeur sur un entier, il contient l'adresse de l'adresse de l'entier. Son type est `int **`.

Ces possibilités sont tout particulièrement intéressantes pour les appels de fonction. En effet, si on passe en argument d'une fonction l'adresse d'une variable, la fonction va pouvoir modifier la variable locale par l'intermédiaire de son adresse. L'adresse est recopiée dans un pointeur local, on ne peut donc pas la modifier, mais en revanche la case mémoire pointée est la même et peut donc être modifiée. Voici un exemple :

```
int exemple(int x, int *y)
```

18. Les variables tableaux globales sont par défaut initialisées à zéro, ainsi que les variables locales déclarées `static`, en revanche les variables tableaux locales *normales* (non statiques) NE sont PAS initialisées.

```
{
   x += 3;
   *y += x;
   return(x);
}

main()
{
   int a,b,c;

   a = 2;
   b = 1;
   c = exemple(a,&b);
   printf("a=%d b=%d c=%d\n");
   exit(0);
}
```

Dans cet exemple, le programme affiche `a=2 b=6 c=5`. La variable **b** a été modifiée par l'appel de fonction : on parle alors d'*effet de bord*.

> Pour écrire une fonction qui modifie la valeur d'un pointeur (et pas seulement l'élément pointé comme ci-dessus), il faudra passer en paramètre à cette fonction l'adresse du pointeur à modifier, c'est-à-dire un pointeur sur le pointeur.

Lorsque l'on définit des pointeurs sur des structures, il existe un raccourci particulièrement utile (`st` est ici un pointeur sur une structure contenant un champ nommé `champ`) :
`(*st).champ` est équivalent à `st->champ`, l'opérateur flêche s'obtenant par la succession des signes `-` et `>`.

> Les pointeurs sont également utiles dans les appels de fonction même si la fonction n'a pas à modifier la valeur pointée : il est en effet beaucoup plus efficace de passer une adresse à une fonction que de recopier un gros bloc de mémoire (on évite ainsi généralement de passer des structures entières en paramètre, on préfère un pointeur sur la structure). L'accès aux champs d'une structure référencée par un pointeur est ainsi le cas d'écriture le plus général, d'où la syntaxe spéciale.
>
> Lorsque la fonction ne modifie pas l'objet pointé, on utilisera avantageusement le mot clé **const** qui permet au compilateur de vérifier qu'aucune écriture n'est faite directement dans l'objet pointé (ce n'est qu'une vérification rudimentaire).
>
> Une dernière utilisation commune des pointeurs est le cas de fonctions pouvant prendre en paramètre des zones mémoires dont le type n'est pas imposé. On fait cela au moyen du pointeur générique de type `void *`, qui peut être affecté avec n'importe quel autre type de pointeur. Un exemple typique de cette utilisation est la fonction **memset()** (définie dans `<string.h>`) qui permet d'initialiser un tableau de données de type quelconque.

5.3 Allocation dynamique de mémoire

Pour pouvoir stocker en mémoire une valeur entière, il faut donc explicitement demander une adresse disponible. Il existe heureusement un autre moyen d'obtenir des adresses mémoire valides, c'est le rôle de la fonction `malloc()`. Son prototype est `void *malloc(size_t size);`. Le type `size_t` est un type spécial, on pourra retenir qu'il s'agit en fait d'un entier codant un nombre d'octets mémoire. La taille d'un objet de type donné s'obtient par l'opérateur `sizeof()`, appliqué à la variable à stocker ou à son type.

Ainsi on pourra écrire

```
int *x;
   /* on peut aussi écrire x=(int *)malloc(sizeof(*x)); */
x = (int *)malloc(sizeof(int));
*x = 2324;
```

ce qui :

1. Déclare la variable **x** comme un *pointeur* sur une valeur de type **int**.
2. Affecte à **x** une adresse dans la mémoire qui est réservée pour le stockage d'une donnée de taille **sizeof(int)**, donc de type **int**.
3. Place la valeur 2324 à l'adresse **x**.

L'appel **malloc(s)** réserve en mémoire un bloc de taille **s**. On remarque que la fonction `malloc()` retourne un pointeur de type `void *` : on appelle cela un pointeur générique (c'est le type de pointeur que manipulent les fonctions qui doivent travailler indépendamment de l'objet pointé). Il faut bien entendu le convertir dans le type du pointeur que l'on désire modifier et ceci se fait par transtypage.

La mémoire ainsi allouée le reste jusqu'à la fin du programme. C'est pour cela que lorsqu'on n'a plus besoin de cette mémoire, il convient de préciser qu'elle peut à nouveau être utilisée. Ceci se fait par la fonction `free()` dont l'argument doit être un pointeur générique. La taille n'a pas besoin d'être précisée car elle est mémorisée lors de l'appel à `malloc()`. Dans le cas présent, la libération de la mémoire se ferait par `free((void *)x);`.

Du fait des méthodes utilisées pour gérer cette allocation dynamique, il est préférable d'éviter de faire de très nombreuses allocations de très petits blocs de mémoire. Si cela est nécessaire, on écrira sa propre fonction de gestion mémoire, en faisant les allocations et libérations avec `malloc()` et `free()` par gros blocs [19].

5.4 Arithmétique sur les pointeurs

Un pointeur est en fait un entier un peu spécial puisqu'il contient une adresse mémoire. Certaines opérations arithmétiques sont disponibles afin de permettre la manipulation aisée de blocs de mémoire contenant plusieurs éléments consécutifs du même type.

On dispose ainsi de l'addition d'un pointeur `p` et d'un entier `n` qui donne un pointeur de même type que `p`, pointant sur le n-ième élément du bloc commençant à l'adresse pointée par `p`. Par exemple `p+0` vaut `p` et pointe donc sur l'élément numéro zéro, `p+1` pointe sur l'élément suivant... Ceci fonctionne quelle que soit la taille de l'élément pointé par `p`, `p+n` provoquant ainsi un décalage d'adresses correspondant en octets à `n` fois la taille de l'objet pointé par `p`.

Il est possible aussi d'effectuer la soustraction de deux pointeurs de même type, et cela donne un résultat entier qui correspond au nombre d'éléments qui peuvent être placés entre ces deux pointeurs. L'exemple suivant montre une façon d'initialiser un bloc mémoire (ce n'est pas la méthode la plus simple pour y arriver, mais elle est efficace) :

```
#define TAILLE 1000
typedef struct pixel {
   unsigned char r,v,b;
   int x,y;
} pix;

main() {
   pix *p;
   pix *q;

   p=(pix *)malloc(sizeof(pix)*TAILLE);

   for (q=p;q-p<TAILLE;q++) {
       q->r=q->v=q->b=0;
       q->x=q->y=0;
   }
    ...
}
```

Il existe des similarités entre les tableaux et les pointeurs. En particulier, un tableau à une dimension est en fait un pointeur constant (il n'est pas possible de modifier l'adresse sur laquelle il pointe) vers la première case du tableau. Ainsi il y a une équivalence totale entre les écritures `p[i]` et `*(p+i)` lorsque `p` est un pointeur ou un tableau monodimensionnel et `i` un entier (*char*, *int* ou *long*) [20].

19. L'inefficacité de la gestion par petits blocs vient entre autre de l'ensemble des paramètres stockés pour chaque malloc() effectué, qui peut atteindre 16 octets ! D'où, pour allouer un bloc de 1024*1024 éléments de type char, environ 1 Mo si cela se fait par un appel *malloc(1024*1024)* et environ 17 Mo si cela est fait en 1024*1024 appels à malloc(1)...

20. On a même équivalence avec `i[p]` bien que cette dernière écriture soit à éviter !

Attention, il n'y a pas équivalence simple entre une déréférenciation de pointeur et un accès à un tableau à plusieurs dimensions. Il faut en fait voir un tableau à deux dimensions `int tab[3][5]` comme un tableau de trois éléments qui sont des tableaux de cinq entiers (et non l'inverse). Ces trois éléments sont stockés les uns à la suite des autres en mémoire.

Les écritures `tab[i][j]`, `*(tab[i]+j)`, `*(*(tab+i) +j)` et `*((int *)tab+5*i+j)` sont alors équivalentes, mais il reste fortement conseillé pour la lisibilité de se limiter à la première.

De façon similaire, le passage d'un tableau en paramètre à une fonction peut se faire de façon équivalente sous la forme d'un pointeur (`void test(int *a)`) et sous la forme d'un tableau sans taille (`void test(int a[])`). Dans le cas de tableaux à plusieurs dimensions... la deuxième solution est très fortement déconseillé, on vous aura prévenu ! Si le type est un simple pointeur vers le début du tableau à deux dimensions, il faudra donc fournir également à la fonction la taille du tableau (tout ceci peut se mettre dans une jolie structure, comme on le fait par exemple pour des images).

5.5 Les chaînes de caractères

Les chaînes de caractères sont représentées en C sous la forme d'un pointeur sur une zone contenant des caractères. La fin de la chaîne est marquée par le caractère spécial `'\0'`. Il est possible d'utiliser la déclaration sous forme de tableau : **char ch[256];** définit une chaîne dont la longueur maximale est ici de 256 caractères, en comptant le caractère de terminaison. En fait, par convention, toute chaîne de caractère se termine par le caractère **nul** (pas le chiffre `'0'`, mais l'entier 0 noté souvent pour cette utilisation `'\0'`). La chaîne de caractère peut aussi se déclarer sous la forme d'un pointeur `char *`, vers une zone qu'il faut allouer avec `malloc()`.

On peut écrire des chaînes constantes délimitées par des guillemets. L'expression `"chaine\n"` est donc une chaîne comportant le mot `chaine` suivi d'un retour chariot, et comme toujours terminée par un caractère `'\0'`.

La fonction `strlen(s)` (déclarée dans <string.h> dans la bibliothèque standard) retourne la longueur de la chaîne `s` jusqu'au premier code `'\0'`[21]. La fonction `strcmp(s1,s2)` permet de comparer deux chaînes. Elle retourne -1 si lexicographiquement `s1`<`s2`, 0 si les deux chaînes sont identiques, et +1 si `s1`>`s2`.

Pour l'anecdote, cette fonction est équivalente à `strlen()` (!) :

```
int mystrlen(char *s) {
       int longueur;
       for (longueur=0; *s++;longueur++) ;
       return longueur;
}
```

Les structures autoréférentielles[22]

Ce sont des structures dont un des membres est un pointeur vers une structure du même type. On les utilise couramment pour programmer des listes chaînées et pour les arbres.

Voici un exemple simple mais complet, dans lequel la liste initiale est allouée en un bloc consécutif et chaînée, puis ensuite utilisée comme une liste chaînée quelconque :

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

typedef struct liste {
   struct liste *suivant;
   int valeur;
} tliste;


main() {
   tliste *maliste;
   tliste *temp;
   int i,som;
```

21. Il s'agit du nombre de caractères effectivement utiles dans la chaîne et non de la taille du bloc mémoire alloué comme le retournerait `sizeof()`.

22. On dit aussi parfois **structures récursives** par abus de langage.

```
   maliste=(liste *)malloc(100*sizeof(liste));
   /* chainage des elements de la liste */
   for(i=0;i<99;i++) {
       (maliste+i)->suivant=(maliste+i+1);
       (maliste+i)->valeur=i;
   };
   (maliste+99)->valeur=99;
   (maliste+99)->suivant=NULL;

   som=0;
   /* somme des elements de la liste */
   /* fonctionne par le chainage et non par les positions en memoire */
   for (temp=maliste; temp; temp=temp->suivant) {
      som += temp->valeur;
   }
   printf("somme: %d\n",som);

   exit(0);
}
```

Dans la déclaration d'une structure autoréférentielle, il n'est pas possible d'utiliser le type **tliste** défini par le **typedef** comme on le fait pour les déclarations de variables du **main()**, il est possible en revanche d'utiliser le type structuré **struct liste** en cours de déclaration.

5.6 Les pointeurs de fonction

Il est parfois souhaitable qu'une fonction générique ait besoin de pouvoir effectuer le même traitement à peu de choses près. C'est par exemple le cas d'une fonction de tri générique, qui aura besoin d'une fonction de comparaison, différente selon l'application [23].

Pour passer une fonction en paramètre à une autre fonction, il existe des pointeurs de fonctions, qui sont en fait des pointeurs contenant l'adresse mémoire de début de la fonction [24]. Afin que le compilateur connaisse les arguments de la fonction pointée, ceux-ci font partie de la déclaration du pointeur. On aura par exemple :

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

/* fonction qui effectue la derivation numerique partielle */
/* par difference finie d'une fonction de x et y selon x */
/* fonc est le nom local de la fonction a deriver */
/* x,y est le point de derivation et h est l'intervalle */
float numderiv(float (*fonc) (float, float), float x, float y, float h) {
   return (fonc(x+h,y)-fonc(x-h,y))/2/h;
   /* on pourrait aussi ecrire : */
   /*   return ((*fonc)(x+h,y)-(*fonc)(x-h,y))/2/h; */
   /*   mais attention aux parentheses ! */
}

/* une fonction de deux variables */
float f1(float x, float y) {
   return x*y*y;
}
/* La meme fonction inversee */
float f2(float x, float y) {
   return f1(y,x);
}
/* une autre fonction */
```

23. C'est comme cela que fonctionne la fonction de tri `qsort()`.
24. Dans le cas où la fonction a été déclarée **inline**, c'est la version sous forme de fonction standard qui est utilisée.

```
float f3(float x, float y) {
   return 3*x*x;
}

main() {
   float x,y;
   x=3.0; y=2.0;
   printf("x:%f y:%f  f(x,y):%f    df/dx(x,y):%f\n",x,y,f1(x,y),
                                        numderiv(f1, x, y, 0.1));
               /* on pourrait aussi ecrire numderiv( &f1 , x, y, 0.1)); */
   printf("x:%f y:%f  f(x,y):%f    df/dy(x,y):%f\n",x,y,f2(y,x),
                                        numderiv(f2, y, x, 0.1));
   printf("x:%f y:%f f3(x,y):%f   df3/dx(x,y):%f\n",x,y,f3(x,y),
                                        numderiv(f3, x, y, 0.1));
   exit(0);
}
```

Qui affiche :

```
x:3.000000 y:2.000000  f(x,y):12.000000    df/dx(x,y):3.999996
x:3.000000 y:2.000000  f(x,y):12.000000    df/dy(x,y):11.999994
x:3.000000 y:2.000000 f3(x,y):27.000000   df3/dx(x,y):17.999982
```

Il est possible de passer une fonction en argument (donc un pointeur de fonction) à une autre fonction indifféremment par `f1` ou `&f1`. Ceci car le compilateur sait qu'il s'agit nécessairement d'un pointeur, la fonction n'étant pas une variable. De la même façon, lors de son utilisation, les syntaxes `fonc()` et `(*fonc) ()` sont équivalentes (noter les parenthèses dans la deuxième écriture, nécessaires du fait des priorités des opérateurs).

> Pour les amateurs de programmes illisibles, il est possible d'écrire les appels de fonctions normaux de plusieurs façon aussi (en utilisant **gcc**, ceci n'est pas valable sur tous les compilateurs) : `f1(y,x)` est équivalent à `(*f1)(y,x)`, et même à `(* (* (&f1))) (y,x)`.

6 Le préprocesseur

Lors de la compilation d'un fichier source en C, il se déroule en fait plusieurs opérations successives. La première phase est une passe de remplacement textuel uniquement et est effectuée par le préprocesseur. Il est possible de voir le code source en sortie du préprocesseur avec la commande `gcc -E`.

Les lignes que le préprocesseur interprète commencent par un dièze (#) éventuellement précédé de caractères d'espacement. Les commmandes du préprocesseur s'étendent chacune sur une ligne entière et une seule; la seule façon d'écrire une commande du préprocesseur sur plusieurs lignes est d'empêcher l'interprétation du caractère de fin de ligne en le faisant précéder d'un caractère \ (tout à fait à la fin de la ligne).

Trois types d'opérations sont effectuées par le préprocesseur : **l'inclusion de fichiers**, **la substitution des macros** et **la compilation conditionnelle**.

6.1 L'inclusion de fichiers

La commande `#include <nom de fichier>` permet d'inclure le texte du fichier référencé à la place de la commande. Deux syntaxes existent pour cette commande : `#include <toto.h>` et `#include "toto.h"` qui diffèrent par les répertoires dans lesquels le préprocesseur va chercher `toto.h`.
On retiendra en général `#include <stdio.h>` pour les *headers* du système (le compilateur ne va pas chercher dans le répertoire courant, mais uniquement dans les répertoires standards du système), et `#include "toto.h"` pour les *headers* du projet, cette dernière syntaxe forçant le compilateur à chercher le fichier d'abord dans le répertoire courant.

6.2 Les macros (*#define*)

Une macro permet un remplacement d'un mot particulier dans tout le texte qui suit. Par exemple, la définition suivante remplace dans toute la suite le mot `test` par *printf("Ceci est un test\n")* :

```
#define test printf("Ceci est un test\n");
```

Il n'est pas possible dans la suite du code de redéfinir de la même façon `test`, il faut commencer pour cela par supprimer la définition courante par `#undef test`. Appliquer `#undef` à un identificateur inconnu n'est pas une erreur, c'est simplement sans effet.

Les macros permettent également de définir des remplacements plus intelligents, avec des arguments variables. Ainsi la fonction MIN décrite précédemment (paragraphe sur les opérateurs) peut s'écrire sous forme de macro :

```
#define MIN(a,b) (a)>(b)?(b):(a)
```

Tout écriture dans la suite du texte de `MIN(expr1,expr2)` fera le remplacement textuel en écrivant l'expression de remplacement après avoir substitué à `a` le texte de `expr1` et à `b` le texte de `expr2`. La présence de nombreuses parenthèses est chaudement recommandée car dans certains cas la priorité des opérations effectuées dans la macro peut être supérieure à celle des opérations faites dans ses arguments. Par exemple, supposons que l'on définisse une macro `PRODUIT` ainsi :

```
#define PRODUIT(a,b) a*b
```

Alors une instance notée `PRODUIT(2+3, 7);` donnera $2 + 3 * 7$ c'est à dire 23, ce qui n'est peut-être pas l'effet recherché...

D'autres fonctionnalités de remplacement plus fines sont possibles, mais sortent du cadre de cet aide-mémoire.

Le préprocesseur tient également à jour des macros spéciales qui contiennent la date, l'heure, le numéro de la ligne en cours et le nom de fichier en cours : `__DATE__`, `__TIME__`, `__LINE__` et `__FILE__`, qui sont utiles dans le cas d'affichage de messages d'erreur.

6.3 Compilation conditionnelle

On souhaite parfois compiler ou non certaines parties d'un programme, par exemple s'il doit fonctionner sur des machines utilisant des systèmes d'expoitation différents, mais aussi pour inclure ou non du code de déverminage par exemple. Pour cela le préprocesseur permet d'inclure des parties de code en fonction de conditions. La syntaxe est :

```
#if <condition>
       <code C a compiler si la condition est vraie>
#endif
```

Lorsqu'une alternative est possible, on peut écrire :

```
#if <condition>
       <code C a compiler si la condition est vraie>
#else
       <code C a compiler si la condition est fausse>
#endif
```

Dans le cas d'alternatives multiples, on a même :

```
#if <condition 1>
       <code C a compiler si la condition 1 est vraie>
#elif  <condition 2>
       <code C a compiler si la condition 1 est fausse et 2 vraie>
#else
       <code C a compiler si les deux conditions sont fausses>
#endif
```

Les conditions sont des expressions C booléennes ne faisant intervenir que des identificateurs du préprocesseur. Les expressions de la forme `defined(<identificateur>)` sont remplacées par 1 si l'identificateur est connu du préprocesseur et par 0 sinon. Comme il est fréquent d'écrire `#if defined(<id>)` ou `#if !defined(<id>)`, des formes abrégées existent et se notent respectivement `#ifdef <id>` et `#ifndef <id>`.

Citons deux autres utilisations importantes des remplacements par des macros, qui sont la compilation avec du code de déverminage (*debug*) optionnel, et la gestion des variables globales. Le petit morceau de code suivant effectue un appel à **printf()** pour afficher un message, mais uniquement si le programme a été compilé en définissant DEBUG. Ceci peut avoir été fait dans un des fichiers *header* (.h) inclus systématiquement, ou bien par l'ajout de l'option **-DDEBUG** pour **gcc** sur la ligne de compilation.

```
#include <stdio.h>

#ifdef DEBUG
#define DEB(_x_) _x_
#else
```

```
#define DEB(_x_)
#endif

main() {
  DEB(printf("Ceci est un message de debug\n"));
}
```

La macro **DEB** est ici remplacée, soit par ce qui est entre parenthèses si DEBUG est défini, soit par rien dans le cas contraire. C'est plus court et plus lisible que :

```
main() {
#ifdef DEBUG
  printf("Ceci est un message de debug\n");
#endif
}
```

L'autre utilisation importante concerne les variables globales. En effet, celles-ci doivent être déclarées dans UN SEUL fichier source (si possible celui qui contient la fonction **main()**), et doivent avoir une déclaration précédée de **extern** dans tous les autres sources. Ceci se fait simplement en déclarant cette variable avec une macro EXT dans un fichier header (.h) inclus dans tous les sources. La macro est ensuite définie comme valant rien du tout si l'inclusion est faite dans le fichier contenant le **main()**, et comme **extern** partout ailleurs. Voyons un petit exemple :

Ici un fichier *general.h*, inclus dans TOUS les sources :

```
/* EXT sera remplace par extern partout sauf si MAIN est defini */
#ifdef MAIN
#  define EXT
#else
#  define EXT extern
#endif

EXT int monglob1;
EXT char *glob2;
```

Voici le début de *calcul.c* :

```
#include "general.h"

int trescomplique(int a) {
  return a+monglob1;
}
```

Et le début de *main.c* :

```
#define MAIN
#include "general.h"
#include <stdio.h >

main() {
  monglob1=3;
  printf("Resultat: %d\n",trescomplique(2));
}
```

7 Fonctions d'entrée/sortie

De nombreuses fonctions d'entrée-sortie existent dans la bibliothèque `stdio.h`. Seules certaines sont décrites dans cette partie, le reste est présenté dans l'annexe. Une description plus détaillée de ces fonctions se trouve à la fin du manuel de référence de Kernighan et Ritchie, ou dans le *man* en ligne sous **Unix**.

7.1 Le passage d'arguments en ligne de commande

La fonction `main()` d'un programme peut en fait prendre des arguments. La déclaration complète est (les types des arguments sont imposés) :

```
int main(int argc, char **argv)
```

On écrit aussi fréquemment : `int main(int argc, char *argv[])` qui a le mérite de rappeler que `argv` est un tableau. Ces deux paramètres correspondent au nombre d'arguments de la ligne de commande *en comptant le nom de l'exécutable* (`argc`) et à la liste de ces arguments sous forme de chaînes de caractères (`argv[1]`, `argv[2]`, ..., `argv[argc-1]`). La première chaîne `argv[0]` contient le nom d'appel du programme.

Il est ainsi possible de passer des arguments en ligne de commande à un programme. Ces arguments peuvent être convertis avec les fonctions de conversion de la bibliothèque standard (`#include <stdlib.h>` : cf. Annexe) :

```
double atof(char *s);
int atoi(char *s);
long atol(char *s);
```

Ou grâce aux fonctions d'entrée mise en forme que nous allons voir maintenant.

7.2 Entrée/sortie simples

Tout ce qui suit est subordonné à l'inclusion du header <stdio.h> qui déclare les fonctions d'entrée-sortie ainsi que la structure **FILE**. Les deux sections qui suivent donnent un aperçu des fonctions d'entrée-sortie sans les détailler. On se reportera à l'annexe qui suit pour de plus amples précisions.

Les fonctions d'entrée-sortie opèrent sur des flots (de type *FILE*), qui doivent avoir été déclarés au préalable. L'entrée standard, la sortie standard et la sortie d'erreur sont automatiquement ouverts et se nomment respectivement *stdin*, *stdout* et *stderr*[25]. Par défaut l'entrée standard est le clavier, et les sorties sont à l'écran. Les opérateurs de redirection du *shell* permettent néanmoins d'altérer ceci en redirigeant ces accès vers/depuis un fichier/autre programme.

Pour des flots associés à des fichiers, il faut au préalable les ouvrir avec la fonction **fopen()**. Une bonne règle à respecter est que tout flot ouvert par un programme doit être fermé par **fclose()** dès qu'il ne sert plus.

Les flots sont des abstractions qui permettent de ne pas avoir à repréciser à chaque utilisation d'un fichier tous les paramètres : une fois le fichier ouvert, toutes les opérations ultérieures font référence à ce fichier par le flot, ce qui est beaucoup plus rapide car la bibliothèque standard (et aussi le système d'exploitation) retient l'état courant du fichier (position dans le fichier, emplacement sur le disque...). Cette abstraction permet également de travailler sur d'autres objets séquentiels en utilisant les mêmes fonctions, c'est le cas par exemple pour les **tuyaux** entre processus, pour les *sockets* réseau...

Les fonctions **fread()** et **fwrite()** permettent de lire des blocs de données dans un flot sans aucune interprétation (Cf. Annexe). Si l'on souhaite plutôt lire un fichier ligne par ligne, il est préférable d'utiliser **fgets()** qui offre l'avantage de s'arrêter aux fins de ligne (caractère `'\n'`), et garantit tout de même que la longueur de la chaîne de stockage ne peut pas être dépassée.

Dans les cas où le but est de traiter les caractères un à un sans attendre, il faut utiliser **getc()** et **putc()** (ou leurs équivalents **getchar()** et **putchar()** pour les flots d'entrée-sortie standards). La fonction **ungetc()** qui permet d'implanter simplement un *look-ahead*[26] de façon simple peut également rendre service.

> Les fonctions **fopen()**, **fread()**, **fprintf()**, **fflush()**... sont des fonctions présentes dans la bibliothèque standard, et sont donc normalement disponibles sur tous les systèmes d'exploitation. Ces fonctions sont écrites en C et précompilées dans un fichier d'archive qui forme la bibliothèque standard, et qui est ajouté automatiquement lors de la compilation. Ecrire **fopen()** est en fait un simple appel de fonction vers cette bibliothèque, dont le code exécutable fait partie du programme. A son tour, ces fonctions (**fopen()** par exemple) utilisent des services fournis par le système d'exploitation pour effectuer leur tâche. Ces services sont sollicités via un ou plusieurs **appels système** qui demandent au noyau du système d'exploitation d'ouvrir le fichier voulu. Sous **Unix** l'appel système (dont l'utilisation est similaire à un appel de fonction) utilisé ainsi est **open()**, qui retourne un entier appelé **descripteur de fichier Unix**. Cette valeur de retour est stockée dans l'objet *FILE* afin d'en disposer pour les manipulations ultérieures. Quelques autres appels système concernant les fichiers sous **Unix** sont **read()**, **write()**, **close()**...
>
> Il est fortement recommandé de ne pas mélanger pour un même flot des appels aux fonctions de la bibliothèque standard et des appels système **Unix** car il n'y aurait dans ce cas aucune garantie que l'ordre des opérations soit respecté (à cause des tampons en mémoire maintenus par la bibliothèque standard).

25. En fait, cette propriété n'est assurée que pour un système d'exploitation **Unix**, même si la plupart des compilateurs C fonctionnant dans d'autres environnements tentent de simuler ce comportement.

26. Littéralement *regarder en avant*, c'est une méthode de décision du type de lexème qui suit à partir de la lecture du premier caractère uniquement.

7.3 Entrées/sorties avec format : *fprintf(), fscanf()*

Pour obtenir des résultats joliment mis en forme, la commande à utiliser est **fprintf(flot, format, variables...)**. Lorsque *flot* est la sortie standard, un raccourci existe : **printf(format, variables...)**. Le *format* est une chaîne de caractères qui décrit la façon d'afficher les variables qui suivent ; par exemple `printf("x=%d y=%f\n",x,y);` affiche **x=** suivi de la valeur de la variable entière **x**, puis un espace puis **y=** suivi de la valeur de la variable flottante **y** et un retour chariot. La fonction **printf** est une fonction particulière qui prend un nombre variable d'arguments. Les codes à utiliser dans le format indiquent le type de la variable et sont présentés en annexe. Ainsi pour afficher un nombre complexe, on écrira par exemple `printf("z=%f+i%f\n",Re(z),Im(z));`.

La valeur de retour de **fprintf()** est **le nombre de caractères** effectivement écrits, ou une valeur négative en cas d'erreur.

La lecture de valeur se fait par la fonction **fscanf(flot, format, adresses des variables...)**. Le principe est similaire à **fprintf()**, excepté que les adresses des variables doivent être données afin que **fscanf()** puisse aller modifier leur valeur. Lorsque *flot* est l'entrée standard, on peut utiliser **scanf(format, adresses des variables...)**. Ainsi pour lire en entrée une variable flottante **float x;** suivie d'une chaîne de caractères **char cha[1024];**, on écrira `scanf("%f%s",&x, cha);` La variable **x** est passée par adresse, et la chaîne aussi puisqu'il s'agit d'un pointeur, donc d'une adresse mémoire.

La fonction **fscanf()** retourne **le nombre de champs** qu'elle a pu lire, ou la constante spéciale *EOF* en cas d'erreur ou de fin de fichier.

> Les caractères autres que des caractères de formatage et les espaces présents dans le format de **scanf()** doivent correspondre **exactement** aux caractères entrés au clavier. Il faut en particulier éviter de mettre un `\n` dans ce format. Les caractères d'espacement sont considérés par **fscanf()** comme des séparateurs de champs. Il n'est donc pas possible de récupérer d'un coup une chaîne de caractères comportant des blancs avec cette fonction. C'est pourquoi on utilisera plutôt `char *fgets(char *s, int n, FILE *f)` qui lit une ligne jusqu'au retour chariot ou jusqu'à ce qu'il y ait `n` caractères et stocke le résultat dans *s* (elle retourne *s* également s'il n'y a pas eu d'erreur).

Il ne faut pas perdre de vue que le système de fichiers d'**Unix** est bufferisé. Ainsi pour l'entrée standard, celle-ci n'est prise en compte que ligne par ligne après un retour chariot [27]. De même pour la sortie standard [28]. Lorsqu'on écrit dans un fichier, ou si la sortie standard est redirigée, la bufferisation est encore plus importante puisque la taille du tampon peut atteindre plusieurs kilo-octets. Pour forcer l'écriture, on utilise **fflush()**. Cette fonction est utile, car si un programme n'arrive pas à son terme (arrêt intempestif de la machine, ou erreur à l'exécution par exemple) les buffers sont irrémédiablement perdus.

Enfin la fonction **fclose()** permet de fermer un fichier, en vidant auparavant son buffer.

8 Compléments : la bibliothèque standard et quelques fonctions annexes

La bibliothèque standard ne fait pas partie à proprement parler du langage C mais elle a été standardisée par la norme ANSI et offre donc un ensemble de déclarations de fonctions, de déclarations de types et de macros qui sont communes à tous les compilateurs qui respectent la norme, quel que soit le système d'exploitation. Quelques exemples présentés ici sont néanmoins des extensions de cette bibliothèque standard spécifiques au monde **Unix**. De nombreuses fonctions de la bibliothèque standard reposent sur des appels systèmes **Unix** qui leur ressemblent, comme **open()**, **close()**, **read()** ou **lseek()** par exemple. Si la portabilité vers des environnements non-unix n'est pas de mise, il pourra être intéressant dans certains cas d'utiliser plutôt ces dernières fonctions.

Toutes les déclarations de la bibliothèque standard figurent dans des fichiers *include* :

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

Ces fichiers doivent être inclus avant toute utilisation d'une fonction de la bibliothèque par une directive du type `#include <fichier_include.h>`. Ils peuvent être inclus dans un ordre quelconque, voire plusieurs fois dans

27. Il est possible, mais ceci est spécifique **Unix** de reparamétrer le terminal afin de recevoir les caractères un à un sans attendre un retour chariot.
28. Là un **fflush()** permet de forcer l'écriture n'importe quand. Il reste néanmoins préférable d'utiliser **stderr** pour les sorties d'erreurs non-bufferisées.

un même source. Ils contiennent des déclarations de fonctions que le compilateur connaît par défaut pour la plupart, excepté pour <math.h> qui déclare des fonctions de la bibliothèque mathématique qu'il faut indiquer à l'édition de lien de façon explicite (Voir 2).

Les identificateurs masqués à l'utilisateur à l'intérieur de cette bibliothèque ont des noms commençant par un caractère de soulignement _. Il est donc recommandé d'éviter d'utiliser ce type de dénomination pour éviter d'éventuels conflits.

Seules les fonctions principales sont présentées ici.

Pour des informations plus détaillées, on se reportera au pages de manuel en ligne sous **Unix** accessibles par **man fgets** par exemple pour la fonction **fgets()**. Mais une fonction comme **open()** est également le nom d'une commande **Unix** aussi **man open** ne retournera pas la page voulue. Dans ce cas, on précisera la section du manuel correspondante, ici la section 2 (pour les appels système) ou 3 (pour les fonctions de la bibliothèque). Ceci s'écrit **man 2 open** ou **man 3 fprintf**.

8.1 <stdio.h> : entrées-sorties

Il s'agit de la partie la plus importante de la bibliothèque. Sans cela un programme ne peut afficher ou lire des données de façon portable. La source ou la destination des données est décrite par un objet de type FILE, appelé *flot*. Ce type structuré, dont il n'est pas utile de connaître les différents champs (ils peuvent d'ailleurs différer entre les versions de la bibliothèque) est associé à un fichier sur disque ou à un autre périphérique comme le clavier ou l'écran. Il existe des flots binaires et des flots de texte, mais sous **Unix** ceux-ci sont identiques[29].

Un *flot* est associé à un fichier par ouverture, ceci retourne un pointeur sur une structure FILE. A la fin des opérations sur le flot, il faut le fermer.

Au début de l'exécution du programme, trois flots spéciaux, nommés *stdin* pour l'entrée standard, *stdout* pour la sortie standard et *stderr* pour la sortie d'erreur sont déjà ouverts. En l'absence d'opérations spéciales, *stdin* correspond au clavier, *stdout* et *stderr* correspondent à l'écran.

Ouverture-fermeture de fichiers

`FILE *fopen(const char *nomfich, const char *mode)`

Ouvre le fichier indiqué (le nom est absolu ou relatif au répertoire courant, mais sans remplacement des caractères spéciaux comme ~) et retourne un flot, ou `NULL` en cas d'erreur. L'argument *mode* est une chaîne de caractères parmi :

`r`	lecture seule, la lecture commence au début du fichier.
`w`	écriture seule. Si le fichier n'existait pas, il est créé ; s'il existait, il est écrasé.
`a`	(*append*) aimilaire à **w** mais écrit à la fin du fichier sans l'écraser s'il existait.
`r+`	lecture et écriture. Le flot est positionné au début du fichier.
`w+`	Le fichier est crée (ou écrasé s'il existait), et ouvert en lecture et écriture ; le flot est positionné au début.
`a+`	similaire à **r+**, mais le flot est initialement positionné à la fin du fichier.

Dans le cas des modes lecture-écriture, il faut appeler **fflush()** ou une fonction de déplacement entre les opérations de lecture et d'écriture. L'ajout de `b` à la fin du mode indique un fichier binaire, c'est-à-dire sans conversion automatique des fins de ligne (cela n'a aucun effet sous **Unix**, voir la note 29).

`FILE *freopen(const char *nomfich, const char *mode, FILE *flot)`

Cette fonction sert à associer aux flots prédéfinis *stdin*, *stdout* ou *stderr* un fichier.

`int fflush(FILE *flot)`

Cette fonction n'est à appeler que sur un flot de sortie (ou d'entrée-sortie). Elle provoque l'écriture des données mises en mémoire tampon. Elle retourne zéro en l'absence d'erreur, `EOF` sinon.

`int fclose(FILE *flot)`

Provoque l'écriture des données en attente, puis ferme le flot. La valeur de retour est zéro en l'absence d'erreur, `EOF` sinon.

`FILE *fdopen(int fd, const char *mode)`

Ouvre un flot correspondant au descripteur de fichier **Unix** *fd*. Nécessite un descripteur de fichier valide (par exemple

29. Sous DOS, les flots texte convertissent les séquences CR-LF en LF seul (codes \r\n en \n seul), sous MacOS, ils convertissent le CR en LF (code \r en \n), sous **Unix**, ils convertissent les LF en LF !

issu de **open()**). Le descripteur n'est pas dupliqué, aussi il faut au choix soit terminer par **close(fd)**, soit par **fclose(flot)**, mais surtout pas les deux. Ne fait pas partie de la norme ANSI.

`int fileno( FILE *flot)`
Retourne l'entier descripteur de fichier **Unix** associé au flot. Ne fait pas partie de la norme ANSI.

Positionnement et erreurs

`int fseek(FILE *flot, long decalage, int origine)`
Positionne la *tête de lecture* dans le flot *flot*. L'entier *origine* peut valoir *SEEK_SET*, *SEEK_CUR* ou *SEEK_END* selon que la position est donnée respectivement à partir du début du fichier, de la position courante, ou de la fin. *decalage* représente le décalage demandé en nombre d'octets (attention au signe, une valeur négative faisant revenir en arrière).

fseek() retourne une valeur non nulle en cas d'erreur, zéro en cas de succès.

`long ftell(FILE *flot)`
Donne la valeur de la position courante dans le fichier (en nombre d'octets depuis le début du fichier), ou la valeur *-1L* (-1 sous forme de **long**) en cas d'erreur.

`void rewind(FILE *flot)`
Retourne au début du fichier. Equivalent à **(void)fseek(flot, 0L, SEEK_SET)**.

`int feof(FILE *flot)`
Retourne une valeur non nulle si l'indicateur de fin de fichier est actif, c'est-à-dire si la dernière opération sur le flot a atteint la fin du fichier.

`int ferror(FILE *flot)`
Retourne une valeur non nulle si l'indicateur d'erreur du flot est actif.

Toutes les fonctions qui prennent en argument un flot (de type **FILE ***) nécessitent que ce flot soit valide. En particulier, il ne faut pas faire `f=fopen(...)` puis appeler immédiatement `ferror(f)`, car **f** peut valoir **NULL** si l'ouverture du fichier a échoué. la solution est alors :

```
FILE *f;

f=fopen("mon_fichier","r");
if (f==NULL) {
   perror("Impossible d'ouvrir le fichier\n");
   exit(EXIT_FAILURE);
}
```

`void clearerr(FILE *flot)`
Remet à zéro les indicateurs de fin de fichier et d'erreur.

`void perror(const char *s)`
Affiche sur *stderr* la chaîne *s* et un message d'erreur correspondant à l'entier contenu dans la variable globale *errno*, donc issu de la dernière erreur rencontrée.

perror(s) est équivalent à `(void)fprintf(stderr, "%s : %s\n", strerror(errno) , s)`

`int fgetpos(FILE flot, fpos_t *pos)`
`int fsetpos(FILE *flot, fpos_t *pos)`
Ce sont d'autres fonctions permettant de relire la position dans le fichier (comme **ftell(flot)**) ou de l'imposer (comme **fseek(flot,SEEK_SET)**). Cette position est stockée dans un objet de type **fpos_t** (sous Unix, c'est un *long*, mais ce peut être une structure complexe sur d'autres systèmes d'exploitation).

Entrée-sortie directe

size_t est un type (correspondant à un entier) qui exprime une taille mémoire (en octet, issue par exemple de **sizeof()**, comme pour **malloc()**) ou un nombre d'éléments.

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *flot)`
Lit sur le flot *nobj* objets de taille *size* et les écrit à l'emplacement mémoire pointé par *ptr*. La valeur de retour est le nombre d'arguments lus avec succès, qui peut être inférieur à *nobj*. Pour déterminer s'il y a eu erreur ou si la fin du fichier a été atteinte, il faut utiliser **feof()** ou **ferror()**.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *flot)`
Ecrit sur le flot *nobj* objets de taille *size* pris à l'emplacement pointé par *ptr*. Retourne le nombre d'objets écrits.

Entrée-sortie de caractères

`int fgetc(FILE *flot)`
Lit un caractère sur le flot (*unsigned char* converti en *int*) et le retourne. Retourne *EOF* en cas de fin de fichier ou d'erreur (*EOF* est une constante entière définie par *stdio.h*, valant généralement -1).

`int fputc(int c, FILE *flot)`
Ecrit le caractère *c* (converti en *unsigned char*) sur le flot. Retourne ce caractère ou *EOF* en cas d'erreur.

`int getc(FILE *flot)`
Equivalent à **fgetc()**, mais éventuellement implanté sous forme de macro. Plus efficace que **fgetc** en général. Attention, l'évaluation de *flot* ne doit provoquer d'effet de bord (utiliser une variable pour *flot* et non le retour d'une fonction).

`int putc(int c, FILE *flot)`
Equivalent à **fputc()**. Même remarque que pour **getc()**.

`int getchar(void)`
Equivalent à **getc(stdin)**.

`int putchar(int c)`
Equivalent à **putc(c, stdout)**.

`int ungetc(int c, FILE *flot)`
Remet le caractère *c* (converti en *unsigned char*) sur le flot, où il sera retrouvé à la prochaine lecture. Un seul caractère peut être ainsi remis, et cela ne peut pas être *EOF*. Retourne le caractère en question ou *EOF* en cas d'erreur.

`char *fgets(char *s, int n, FILE *flot)`
Lit une ligne complète de caractères (jusqu'au caractère de fin de ligne compris) d'au plus $n-1$ caractères et place cette chaîne à partir de la case mémoire pointée par *s*. Si un caractère de fin de ligne est lu, il est stocké dans la chaîne. Le caractère `'\0'` est alors ajouté à la chaîne précédemment lue. Retourne *s* si tout s'est bien passé, ou bien *NULL* en cas d'erreur ou si la fin du fichier est atteinte sans aucun caractère disponible le précédant.

`int fputs(char *s, FILE *flot)`
Ecrit la chaîne de caractères *s* sur le flot (n'écrit pas le caractère `'\0'` de fin de chaîne). Retourne une valeur positive ou nulle, ou bien *EOF* en cas d'erreur.

Il existe une fonction **gets()** similaire à **fgets()**, mais son usage est fortement déconseillé.

Entrée-sortie mises en forme

`int fprintf(FILE *flot, const char *format, ...)`
Est la commande de sortie mise en forme. Il s'agit d'une fonction à nombre variable d'arguments (Voir 8.8). La chaîne *format* décrit le texte à écrire, dans lequel des suites de caractères spéciaux commençant par % sont remplacées par les valeurs des arguments qui suivent dans l'ordre. La valeur entière de retour est *le nombre de caractères écrits*, ou en cas d'erreur un nombre négatif.

La chaîne de caractères *format* contient des caractères ordinaires (sans %) qui sont copiés tels quels sur le flot de sortie, et des directives de conversion dont chacune peut provoquer la conversion dans l'ordre d'un argument de **fprintf()** suivant la chaîne de format.

Chaque directive de conversion commence par le caractère % et se termine par un caractère de conversion parmi ceux présentés dans le tableau qui suit. Entre le % et le caractère de conversion peuvent apparaître dans l'ordre :

– Des drapeaux, parmi :
  - # format de sortie alternatif.
  - – cadrage à gauche du champ.
  - + ajout systématique du signe aux nombres.
  - ' ' (espace) ajout d'un caractère pour le signe des nombres, soit l'espace pour les valeurs positives ou le signe `'-'` pour les valeurs négatives.
  - 0 (zéro) complète le début du champ par des zéros au lieu d'espaces pour les nombres.

– Un nombre précisant la largeur minimale du champ d'impression en nombre de caractères.

– Une précision sous la forme d'un point suivi d'un nombre (la valeur de cette précision est considérée comme zéro en l'absence de ce champ). Il s'agit du nombre maximal de caractères à imprimer pour une chaîne de caractères, ou du nombre minimal de chiffres pour les conversions **d**, **i**, **o**, **u**, **x** et **X**. Pour les conversions **e**, **E** et **f** cela donne le nombre de chiffres après le point décimal, et sinon cela correspond au nombre maximal de chiffres significatifs pour les conversions **g** et **G**.

– Enfin une lettre **h**, **l** ou **L** qui modifie le type de l'argument converti. **h** spécifie un argument *short*, **l** spécifie un argument *long*, et **L** un argument *long double* (ou *long long* pour les conversions entières, bien que ceci ne fasse pas partie de la norme ANSI).

Voici enfin les caractères de conversion :

d, i, o, u, x, X	Variante d'*int*. **d**:décimal. **u**:décimal non signé. **o**:octal. **x**:hexadécimal non signé utilisant *abcdef*. **X**:hexadécimal non signé utilisant *ABCDEF*.
e, E	*double* écrit en décimal sous la forme `[-]m.dddddd`Edd. La précision donne le nombre de chiffre après le point de la mantisse, la valeur par défaut est 6. **e** utilise la lettre *e* et **E** utilise *E*.
f	*double* écrit sous la forme `[-]ddd.ddd`. La précision donne le nombre de chiffre après le point décimal, sa valeur par défaut est 6. Si la précision vaut 0, le point décimal est supprimé.
g, G	*double* écrit en décimal comme par une conversion par **f** ou **e** selon la valeur de l'argument.
c	*int* converti en *unsigned char* et affiché comme un caractère ASCII.
s	*char* *. Chaîne de caractères.
p	*void* *. Pointeur écrit en hexadécimal (comme avec %#`lx`).
n	*int* *. Ne convertit pas d'argument, mais écrit dans l'argument le nombre de caractères affichés jusqu'à présent.
%	Affiche simplement un % sans aucune conversion.

Chaque directive de conversion doit correspondre exactement à un argument de type correct dans le même ordre.

> Les erreurs sur les types des arguments ou leur nombre ne sont généralement pas détectés par le compilateur et peuvent provoquer des résultats surprenants à l'exécution.

`int fscanf(FILE *flot, const char *format, ...)`

Est la commande d'entrée mise en forme. Il s'agit aussi d'une fonction à nombre variable d'arguments. La chaîne *format* décrit les objets à lire et leur type, et ceux-ci sont stockés dans les arguments qui suivent *qui doivent être obligatoirement des pointeurs*. La valeur entière de retour est le nombre d'éléments d'entrée correctement assignés (compris entre 0 et le nombre de directives de conversion). Si la fin de fichier est atteinte avant toute lecture, la valeur EOF est renvoyée. Si la fin de fichier est atteinte en cours de lecture, la fonction retourne le nombre d'éléments lus sans erreur.

La chaîne de caractères *format* contient des caractères ordinaires (sans %) qui doivent correspondre exactement aux caractères lus sur le flot d'entrée (exceptés les espaces et les tabulations qui sont ignorés). Elle contient également des directives de conversion formées dans l'ordre par :

– %,

– un caractère facultatif `'*'`, qui, lorsqu'il est présent, empêche l'affectation de la conversion à un argument (le champ est lu et sa valeur ignorée, aucun pointeur n'est utilisé dans les arguments de la fonction),

– un caractère facultatif **h**, **l** ou **L** qui spécifie une variante de taille de type (voir ci-dessous),

– un caractère de conversion parmi ceux présentés dans le tableau qui suit :

d	Variante d'*int **, lu sous forme décimale.
i	Variante d'*int **, la lecture est en octal si l'entier commence par 0, en hexadécimal s'il commence par 0x ou 0X, en décimal sinon.
o	Variante d'*int **, lu sous forme octale, qu'il commence ou non par un 0.
u	Variante d'*unsigned int **, lu sous forme décimale.
x	Variante d'*unsigned int **, lu sous forme hexadécimale.
f, e, g	*float ** (ou *double **). Les trois caractères sont équivalents.
c	*char **. Lit les caractères suivants et les écrit à partir de l'adresse indiquée jusqu'à la largeur du champ si celle-ci est précisée. En l'absence de largeur de champ, lit 1 caractère. N'ajoute pas de `'\0'`. Les caractères d'espacement sont lus comme des caractères normaux et ne sont pas sautés.
s	*char **. Chaîne de caractères lue jusqu'au prochain caractère d'espacement, en s'arrêtant éventuellement à la largeur du champ si elle est précisée. Ajoute un `'\0'` à la fin.
p	*void **. Pointeur écrit en hexadécimal.
n	*int **. Ne convertit pas de caractères, mais écrit dans l'argument le nombre de caractères lus jusqu'à présent. Ne compte pas dans le nombre d'objets convertis retourné par **fscanf()**.
[…]	*char **. La plus longue chaîne de caractère non vide composée exclusivement de caractères figurant entre les crochets. Un `'\0'` est ajouté à la fin. Pour inclure] dans cet ensemble, il faut le mettre en premier []…].
[^…]	*char **. La plus longue chaîne de caractère non vide composée exclusivement de caractères NE figurant PAS entre les crochets. Un `'\0'` est ajouté à la fin. Pour inclure] dans cet ensemble, il faut le mettre en premier [^]…].
%	Lit le caractère %. Ne réalise aucune affectation.

Les variantes de taille de type sont plus complètes que celles pour **fprintf()** : **h** précise un *short* et **l** un *long* pour les conversions *d, i, n, o, u, x*. **l** précise un *double* et **L** un *long double* pour les conversions *e, f, g*. Ainsi pour lire un *double* on utilise `scanf{"%lg",&d)` mais pour l'imprimer on utilise `printf("%g",d)`.

Un champ en entrée est défini comme une chaîne de caractères différents des caractères d'espacement ; il s'étend soit jusqu'au caractère d'espacement suivant, soit jusqu'à ce que la largeur du champ soit atteinte si elle est précisée. En particulier, **scanf()** peut lire au-delà des limites de lignes si nécessaire, le caractère de fin de ligne étant un caractère comme les autres.

Chaque directive de conversion doit correspondre exactement à un pointeur vers un argument de type correct dans le même ordre.

> Même remarque que pour **fprintf()**. De plus, si les pointeurs de destination n'ont pas le bon type (ou pire encore s'il ne sont même pas des pointeurs), le résultat à l'exécution risque d'être déroutant...

`int printf(const char *format, ...)`
Est équivalent à **fprintf(stdout, format, ...)**.

`int scanf(const char *format, ...)`
Est équivalent à **fscanf(stdin, format, ...)**.

`int sprintf(char *s, const char *format, ...)`
Est similaire à **fprintf()** excepté que l'écriture se fait dans la chaîne de caractères *s* plutôt que dans un flux. Attention, la taille de la chaîne *s* doit être suffisante pour stocker le résultat (y compris le dernier caractère `'\0'` invisible de fin de chaîne). Dans le cas contraire, on risque d'avoir des erreurs surprenantes à l'exécution.

`int sscanf(char *s, const char *format, ...)`
Est similaire à **fscanf()** excepté que la lecture se fait dans la chaîne de caractères *s* plutôt que dans un flux.

8.2 <ctype.h> : tests de caractères

Ces fonctions retournent une valeur non nulle si l'argument remplit la condition, et zéro sinon. Toutes ces fonctions ont comme prototype `int isalnum(int c)`, et *c*, bien qu'étant un entier doit contenir une valeur représentable

comme un *unsigned char*.

`isalnum()`	isalpha() ou isdigit() est vrai.
`isalpha()`	isupper() ou islower() est vrai.
`iscntrl()`	caractère de contrôle.
`isgraph()`	caractère imprimable sauf l'espace.
`islower()`	lettre minuscule.
`isprint()`	caractère imprimable y compris l'espace.
`ispunct()`	caractère imprimable différent de l'espace, des lettres et chiffres.
`isspace()`	espace, saut de page, de ligne, retour chariot, tabulation.
`isupper()`	lettre majuscule.
`isxdigit()`	chiffre hexadécimal.

8.3 <string.h> : chaînes de caractères et blocs mémoire

Chaînes de caractères

`char *strcpy(char *s, const char *ct)`
Copie la chaîne *ct*, y compris le caractère `'\0'` dans le chaîne *s*, retourne *s*.

`char *strncpy(char *s, const char *ct, size_t n)`
Copie au plus *n* caractères de *ct* dans *s*, en complétant par des `'\0'` si *ct* comporte moins de *n* caractères.

`char *strcat(char *s, const char *ct)`
Concatène *ct* à la suite de *s*, retourne *s*.

`char *strncat(char *s, const char *ct, size_t n)`
Concatène au plus *n* caractères de *ct* à la suite de *s*, termine la chaîne résultat par `'\0'`, retourne *s*.

`char *strcmp(const char *cs, const char *ct)`
Compare lexicographiquement *cs* et *ct*. Retourne une valeur négative si *cs*<*ct*, nulle si *cs*=*ct*, positive sinon.

`char *strncmp(char *s, const char *ct, size_t n)`
Comme **strcmp()** mais limite la comparaison aux *n* premiers caractères.

`char *strchr(char *s, int c)`
Retourne un pointeur sur la première occurrence de *c* (converti en *char*) dans *cs*, ou `NULL` si il n'y en a aucune.

`char *strrchr(char *s, int c)`
Comme **strchr** mais pour la dernière occurrence.

`char *strstr(const char *cs,const char *ct)`
Retourne un pointeur sur la première occurrence de la chaîne *ct* dans *cs*, ou `NULL` si il n'y en a aucune.

`size_t strlen(const char *cs)`
Retourne la longueur de *cs*, sans compter le caractère de terminaison `'\0'`.

`char *strerror(int n)`
Retourne un pointeur sur la chaîne correspondant à l'erreur *n*.

`char *strtok(char *s, const char *ct)`
Décomposition de *s* en lexèmes séparés par des caractères de *ct*.

Le premier appel à la fonction se fait en précisant dans *s* la chaîne de caractères à décomposer. Il retourne le premier lexème de *s* composé de caractères n'appartenant pas à *ct*, et remplace par `'\0'` le caractère qui suit ce lexème.

Chaque appel ultérieur à **strtok()** avec `NULL` comme premier argument retourne alors le lexème suivant de *s* selon le même principe (le pointeur sur la chaîne est mémorisé d'un appel au suivant). La fonction retourne `NULL` lorsqu'elle ne trouve plus de lexème.

> Cette fonction, d'usage peu orthodoxe (utilisant des variables locales statiques...) est extrêmement efficace car elle est généralement écrite directement en assembleur.

Blocs de mémoire

`void *memcpy(void *s, const void *ct, size_t n)`
Copie *n* octets de *ct* dans *s* et retourne *s*. Attention, les zones manipulées ne doivent pas se chevaucher.

`void *memmove(void *s, const void *ct, size_t n)`
Même rôle que la fonction **memcpy()**, mais fonctionne également pour des zones qui se chevauchent. **memmove()** est plus lente que **memcpy()** lorsque les zones ne se chevauchent pas.

`int memcmp(const void *cs, const void *ct, size_t n)`
Compare les zones mémoire pointées par *cs* et *ct* à concurrence de *n* octets. Les valeurs de retour suivent la même convention que pour **strcmp()**.

`void *memset(void *s, int c, size_t n)`
Ecrit l'octet *c* (un entier convertit en *char*) dans les *n* premiers octets de la zone pointée par *s*. Retourne *s*.

8.4 <math.h> : fonctions mathématiques

La bibliothèque correspondant à ces fonctions n'est pas ajoutée automatiquement par le compilateur et doit donc être précisée lors de l'édition de liens par l'option *-lm*.

Lorsqu'une fonction devrait retourner une valeur supérieure au maximum représentable sous la forme d'un double, la valeur retournée est *HUGE_VAL* et la variable globale d'erreur *errno* reçoit la valeur *ERANGE*. Si les arguments sont en dehors du domaine de définition de la fonction, *errno* reçoit *EDOM* et la valeur de retour est non significative. Afin de pouvoir tester ces erreurs, il faut aussi inclure <errno.h>.

Toutes les fonctions trigonométriques prennent leur argument ou retournent leur résultat en radians. Les valeurs de retour sont toutes des *double*, ainsi que les arguments des fonctions du tableau suivant :

`sin()`	sinus.
`cos()`	cosinus.
`tan()`	tangente.
`asin()`	arc sinus.
`acos()`	arc cosinus.
`atan()`	arc tangente (dans $[-\pi/2,\pi/2]$).
`atan2(y,x)`	pseudo arc tangente de y/x, tenant compte des signes, dans $[-\pi,\pi]$.
`sinh()`	sinus hyperbolique.
`cosh()`	cosinus hyperbolique.
`tanh()`	tangente hyperbolique.
`exp()`	exponentielle.
`log()`	logarithme népérien.
`log10()`	logarithme à base 10.
`pow(x,y)`	x à la puissance y. Attention au domaine : ($x > 0$) ou ($x = 0, y > 0$) ou ($x < 0$ et y entier).
`sqrt()`	racine carrée.
`floor()`	Partie entière exprimée en *double*.
`ceil()`	Partie entière plus 1 exprimée en *double*.
`fabs()`	Valeur absolue.

Pour obtenir la partie entière d'un *float f* sous la forme d'un entier, on utilise le transtypage en écrivant par exemple : `(int)floor( (double)f )`.

8.5 <stdlib.h> : fonctions utilitaires diverses

`double atof(const char *s)`
Convertit *s* en un *double*.

`int atoi(const char *s)`
Convertit *s* en un *int*.

`long atol(const char *s)`
Convertit *s* en un *long*.

`void srand(unsigned int graine)`
Donne *graine* comme nouvelle amorce du générateur pseudo-aléatoire. L'amorce par défaut vaut 1.

`int rand(void)`
Retourne un entier pseudo-aléatoire compris entre 0 et *RAND_MAX*, qui vaut au minimum 32767.

`void *calloc(size_t nobj, size_t taille)`
Allocation d'une zone mémoire correspondant à un tableau de *nobj* de taille *taille*. Retourne un pointeur sur la zone ou `NULL` en cas d'erreur. La mémoire allouée est initialisée par des zéros.

`void *malloc(size_t taille)`
Allocation d'une zone mémoire de taille *taille* (issu de **sizeof()** par exemple). Retourne un pointeur sur la zone ou `NULL` en cas d'erreur. La mémoire allouée n'est pas initialisée.

`void *realloc(void *p, size_t taille)`
Change en *taille* la taille de l'objet pointé par *p*. Si la nouvelle taille est plus petite que l'ancienne, seul le début de la zone est conservé. Si la nouvelle taille est plus grande, le contenu de l'objet est conservé et la zone supplémentaire n'est pas initialisée. En cas d'échec, **realloc()** retourne un pointeur sur le nouvel espace mémoire, qui peut être différent de l'ancien, ou bien `NULL` en cas d'erreur et dans ce cas le contenu pointé par *p* n'est pas modifié.

Si *p* vaut `NULL` lors de l'appel, le fonctionnement est équivalent à **malloc(taille)**.

`void free(void *p)`
Libère l'espace mémoire pointé par *p*. Ne fait rien si *p* vaut `NULL` . *p* doit avoir été alloué par **calloc()**, **malloc()** ou **realloc()**.

`void exit(int status)`
Provoque l'arrêt du programme. Les fonctions **atexit()** sont appelées dans l'ordre inverse de leur enregistrement, les flot restant ouverts sont fermés. La valeur de *status* est retourné à l'appelant [30]. Une valeur nulle de *status* indique que le programme s'est terminé avec succès. On peut utiliser les valeurs *EXIT_SUCCESS* et *EXIT_FAILURE* pour indiquer la réussite ou l'échec.

`void abort(void)`
Provoque un arrêt anormal du programme.

`int atexit(void (*fcn)(void))`
Enregistre que la fonction **fcn()** devra être appelée lors de l'arrêt du programme. Retourne une valeur non nulle en cas d'erreur.

`int system(const char *s)`
Sous **Unix** provoque l'évaluation de la chaîne *s* dans un Bourne-shell (*sh*). La valeur de retour est la valeur de retour de la commande, ou bien 127 ou -1 en cas d'erreur. L'appel **system(NULL)** retourne zéro si la commande n'est pas utilisable.

`char *getenv(const char *nom)`
Récupère la déclaration de la variable d'environnement dont le nom est *nom*. La valeur de retour est un pointeur vers une chaîne du type *nom = valeur*.

```
void qsort(void *base, size_t n, size_t taille, \
int (*comp) (const void *, const void *))
```
Il s'agit d'une fonction de tri (par l'algorithme Quicksort) du tableau $base[0]..base[n-1]$ dont les objets sont de taille *taille*. La fonction de comparaison **comp(x,y)** utilisée doit retourner un entier plus grand que zéro si $x > y$, nul si $x == y$ et négatif sinon.

```
void *bsearch(const void *key, const void *base, size_t n,\
size_t taille, int (*comp) (const void *, const void *))
```
Cette fonction recherche parmi $base[0]..base[n-1]$ (objets de taille *taille* triés par ordre croissant pour la fonction de comparaison fournie), un objet s'identifiant à la clé ***key*. La valeur de retour est un pointeur sur l'élément identique à ***key*, ou `NULL` s'il n'en existe aucun. La fonction de comparaison **comp()** utilisée doit obéir aux mêmes règles que pour **qsort()**.

30. Sous **Unix** si le programme a été lancé depuis un *shell* csh ou tcsh, la valeur de *status* est stockée dans la variable du *shell* de même nom.

`int abs(int n)`
Valeur absolue d'un élément de type *int*.

`long labs(long l)`
Valeur absolue d'un élément de type *long*. Pour les flottants, voir **fabs()** dans <math.h>.

8.6 <assert.h> : vérifications à l'exécution

Il s'agit de la déclaration d'une macro **assert** permettant de vérifier des conditions lors de l'exécution. On écrit pour cela dans le corps du programme :

```
assert( <expression> );
```

Lors de l'exécution, si l'évaluation de `<expression>` donne zéro (expression fausse) la macro envoie sur *stderr* un message donnant l'expression, le fichier source et la ligne du problème et arrête le programme en appelant **abort()**.

Si *NDEBUG* est défini avant l'inclusion de <assert.h>, la macro *assert* n'a aucun effet.

8.7 <limits.h> , <float.h> : constantes limites

<limits.h>

CHAR_BIT	nombre de bits par caractère
CHAR_MIN, CHAR_MAX	valeurs min et max d'un *char*
SCHAR_MIN, SCHAR_MAX	valeurs min et max d'un *signed char*
UCHAR_MIN, UCHAR_MAX	valeurs min et max d'un *unsigned char*
INT_MIN, INT_MAX	valeurs min et max d'un *int*
UINT_MIN, UINT_MAX	valeurs min et max d'un *unsigned int*
SHRT_MIN, SHRT_MAX	valeurs min et max d'un *short*
USHRT_MIN, USHRT_MAX	valeurs min et max d'un *unsigned short*
LONG_MIN, LONG_MAX	valeurs min et max d'un *long*
ULONG_MIN, ULONG_MAX	valeurs min et max d'un *unsigned long*

<float.h>

FLT_DIG	nombre de chiffres significatifs pour un float
FLT_EPSILON	plus petit float f tel que $f + 1.0 \neq 1.0$
FLT_MIN	plus petit nombre représentable
FLT_MAX	plus grand nombre représentable

Les mêmes constantes existent pour les *double*, en écrivant *DBL_* au lieu de *FLT_*.

8.8 <stdarg.h> : fonctions à nombre variable d'arguments

Les déclarations de <stdarg.h> permettent de construire des fonctions ayant une liste d'arguments de longueur inconnue et ayant des types inconnus à la compilation.

La déclaration d'une telle fonction se fait comme une fonction normale, mis à part que l'on écrit des points de suspension après le dernier argument. La fonction doit avoir au moins un argument nommé. On déclare ensuite à l'intérieur de la fonction une variable de type *va_list*, et on l'initialise en appelant **void va_start(va_list vl, last)** avec comme arguments la *va_list* et le dernier argument nommé de la fonction.

Pour accéder ensuite à chaque élément de la liste d'arguments, on appelle la macro **va_arg(va_list vl, type)** qui retourne la valeur du paramètre. *type* est le nom du type de l'objet à lire. Enfin il ne faut pas oublier d'appeler **void va_end(va_list vl)** à la fin de la fonction.

Comme petit exemple, voici une fonction qui ajoute tous ses arguments, chaque argument étant précédé de son type *int* ou *double* :

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

/* un type enumere decrivant les types ! */
typedef enum {INT, DOUBLE} montype;
```

```
/* chaque element est ecrit */
/* sous la forme "type" puis "valeur" */
/* le premier argument est le nombre d'elements */
double somme (int nb, ...) {
  va_list vl;
  float f=0.0;
  int i;
  int vali;
  float valf;

  va_start(vl, nb);
  for (i=0;i<nb;i++) {
    switch (va_arg(vl,montype)) {
      case INT:
       vali=va_arg(vl,int);
       printf("Ajout de l'entier %d\n",vali);
       f += (float)vali;
       break;
      case DOUBLE:
       valf=va_arg(vl,double);
       printf("Ajout du flottant %f\n",valf);
       f += valf;
       break;
      default:
       printf("Type inconnu\n");
     }
  }
  va_end(vl);
  return f;
}

main() {
  int a,b,c,d;
  double e,f,g;
  double result;

  a=2; b=3; c=-5; d=10;
  e=0.1; f=-32.1; g=3.14e-2;
  result=somme(7, INT,a,DOUBLE,e,DOUBLE,f,INT,b,INT,c,INT,d,DOUBLE,g);
  printf("Somme: %f\n",result);
  exit(0);
}
```

Qui affiche à l'exécution :

```
Ajout de l'entier 2
Ajout du flottant 0.100000
Ajout du flottant -32.099998
Ajout de l'entier 3
Ajout de l'entier -5
Ajout de l'entier 10
Ajout du flottant 0.031400
Somme: -21.968597
```

Un dernier exemple classique, une fonction similaire à **printf()** :

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

```
/* mini fonction printf */
/* elle utilise printf ... mais */
/* l'idee est tout de meme la */
int monprintf(char *format, ...) {
  va_list vl;
  int nb=0;

  va_start(vl, format);

  do {
       if (*format != '%') {
              nb++;
              putchar(*format);
              continue;
       };
       format++;
       switch (*format) {
              case 'd':
                     nb+=printf("%d",(int)va_arg(vl,int));
                     break;
              case 's':
                     nb+=printf("%s",(char *)va_arg(vl,char *));
                     break;
              case 'f':
                     nb+=printf("%f",(double)va_arg(vl,double));
                     break;
              default:
                     fprintf(stderr,"Probleme de format");
       }
  } while (*(++format) !='\0');

  va_end(vl);
  return nb;
}

main() {
  int a;
  int t;
  char s[30];
  double b,c;

  a=1;
  b=2.0;
  c=a+b;
  sprintf(s,"CQFD");
  monprintf("Voila : %d + %f = %f ; %s\n",a,b,c,s);
  exit(0);
}
```

L'exécution du programme donne alors :

```
Voila : 1 + 2.000000 = 3.000000 ; CQFD
```