création d'une bibliothèque statique `ar`

Definition (`ar`)

La commande `ar` permet de créer une archive de codes objets. Ces archives peuvent être utilisées au cours d'une édition de liens afin d'y retrouver le code de fonctions utilisées mais non définies.
Nous invitons le lecteur à consulter le manuel en ligne de la commande `ar`.

Exemple

```
INC = include
SRC = src
SRCLIBK = libkey
SRCLIBF = libfile
BUILD = build
BUILDLIBK = $(BUILD)/libkey
BUILDLIBF = $(BUILD)/libfile
CC = gcc
CFLAGS = --std=c11 -pedantic -Wall -I$(INC)
LDFLAGS =
COMPILE = $(CC) $(CFLAGS)
LINK = $(CC) $(LDFLAGS)
AR = ar -r

.PHONY : key file clean libkey libfile all
.SILENT : key file clean
```

Exemple

```
$(BUILDLIBK)/ecrire.o : $(SRCLIBK)/ecrire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBK) || mkdir $(BUILDLIBK)
        $(COMPILE) -c $< -o $@

$(BUILDLIBK)/lire.o : $(SRCLIBK)/lire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBK) || mkdir $(BUILDLIBK)
        $(COMPILE) -c $< -o $@

$(BUILDLIBF)/ecrire.o : $(SRCLIBF)/ecrire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBF) || mkdir $(BUILDLIBF)
        $(COMPILE) -c $< -o $@

$(BUILDLIBF)/lire.o : $(SRCLIBF)/lire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBF) || mkdir $(BUILDLIBF)
        $(COMPILE) -c $< -o $@
```


Exemple

```
$(BUILD)/libfile.a : $(BUILDLIBF)/ecrire.o $(BUILDLIBF)/lire.o
        @test -d $(BUILD) || mkdir $(BUILD)
        $(AR) $@ $^

libfile : $(BUILD)/libfile.a

$(BUILD)/libkey.a : $(BUILDLIBK)/ecrire.o $(BUILDLIBK)/lire.o
        @test -d $(BUILD) || mkdir $(BUILD)
        $(AR) $@ $^

libkey : $(BUILD)/libkey.a
```

Exemple

```
$(BUILD)/mainmod.o : $(SRC)/mainmod.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        $(COMPILE) -c $< -o $@

$(BUILD)/file : $(BUILD)/mainmod.o $(BUILD)/libfile.a
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) -o $@ $< -L$(BUILD) -lfile

$(BUILD)/key : $(BUILD)/mainmod.o $(BUILD)/libkey.a
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) -o $@ $< -L$(BUILD) -lkey

key : libkey $(BUILD)/key
        echo "Generated: $^"

file : libfile $(BUILD)/file
        echo "Generated: $^"

all : key file

clean :
        echo "Cleaning..."
        rm -rf build
```



```
$ make libkey
gcc --std=c11 -pedantic -Wall -Iinclude -c libkey/ecrire.c -
    o build/libkey/ecrire.o
gcc --std=c11 -pedantic -Wall -Iinclude -c libkey/lire.c -o
    build/libkey/lire.o
ar -r build/libkey.a build/libkey/ecrire.o build/libkey/lire
    .o
ar: creating archive build/libkey.a
$ make key
gcc --std=c11 -pedantic -Wall -Iinclude -c src/mainmod.c -o
    build/mainmod.o
gcc -o build/key build/mainmod.o -Lbuild -lkey
Generated: libkey build/key
$ make file
gcc --std=c11 -pedantic -Wall -Iinclude -c libfile/ecrire.c
    -o build/libfile/ecrire.o
gcc --std=c11 -pedantic -Wall -Iinclude -c libfile/lire.c -o
     build/libfile/lire.o
ar -r build/libfile.a build/libfile/ecrire.o build/libfile/
    lire.o
ar: creating archive build/libfile.a
gcc -o build/file build/mainmod.o -Lbuild -lfile
Generated: libfile build/file
$
```

examen du contenu d'une bibliothèque

La commande `nm` (voir page. **??**) permet d'obtenir la liste des symboles définis ainsi que les modules dans lesquels ils l'ont été :

```
$ nm build/libkey.a

build/libkey.a(ecrire.o):
0000000000000000 T _ecrire
                 U _printf

build/libkey.a(lire.o):
0000000000000000 T _lire
                 U _puts
                 U _scanf
$
```

Remarque

On peut remarquer que ces fonctions reposent elles-même sur d'autres fonctions définies par ailleurs...

examen du contenu d'un exécutable lié statiquement

```
$ nm build/file
                 U ___stderrp
0000000100000000 T __mh_execute_header
0000000100000e20 T _ecrire
                 U _exit
                 U _fclose
0000000100001048 b _file
                 U _fopen
                 U _fprintf
                 U _fscanf
0000000100000e50 T _lire
0000000100000de0 T _main
0000000100000ec0 t _ouvre
                 U _printf
                 U dyld_stub_binder
$
```

Remarque

On peut remarquer qu'il reste des symboles non-définis (voir liaison dynamique, plus loin).

création d'une bibliothèque dynamique
-fPIC, -shared

Definition (-fPIC)

La liaison dynamique nécessite du code objet relogeable; c'est le rôle de l'option `-fPIC` de `gcc` que de produire ce type de code à la compilation.

Definition (-shared)

L'option `-shared` de `gcc` permet d'obtenir en sortie une bibliothèque dynamique à partir de codes objets relogeables.

⚠ Attention

Les exemples qui suivent ont été produits sous OSX DARWIN qui utilise `.dylib` (*dynamic library*) comme extension pour les bibliothèques dynamiques et `DYLD_LIBRARY_PATH` pour le contrôle de la localisation des bibliothèques.
Sous *NIX le format des bibliothèques dynamiques est `.so` (*shared object*) et `LD_LIBRARY_PATH` pour le contrôle de la localisation.

Exemple

```
INC = include
SRC = src
SRCLIBK = libkey
SRCLIBF = libfile
BUILD = build
BUILDLIBK = $(BUILD)/libkey
BUILDLIBF = $(BUILD)/libfile
CC = gcc
CFLAGS = --std=c11 -pedantic -Wall -I$(INC)
PIC = -fPIC #picture independent code (dll)
LDFLAGS =
COMPILE = $(CC) $(CFLAGS)
LINK = $(CC) $(LDFLAGS)
AR = ar -r

.PHONY : key file clean libkey libfile all
.SILENT : key file clean
```


Exemple

```
$(BUILDLIBK)/ecrire.o : $(SRCLIBK)/ecrire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBK) || mkdir $(BUILDLIBK)
        $(COMPILE) $(PIC) -c $< -o $@

$(BUILDLIBK)/lire.o : $(SRCLIBK)/lire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBK) || mkdir $(BUILDLIBK)
        $(COMPILE) $(PIC) -c $< -o $@

$(BUILDLIBF)/ecrire.o : $(SRCLIBF)/ecrire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBF) || mkdir $(BUILDLIBF)
        $(COMPILE) $(PIC) -c $< -o $@

$(BUILDLIBF)/lire.o : $(SRCLIBF)/lire.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        @test -d $(BUILDLIBF) || mkdir $(BUILDLIBF)
        $(COMPILE) $(PIC) -c $< -o $@
```

Exemple

```
$(BUILD)/libfile.dylib : $(BUILDLIBF)/ecrire.o $(BUILDLIBF)/lire.o
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) $(PIC) -shared -o $@ $^

libfile : $(BUILD)/libfile.dylib

$(BUILD)/libkey.dylib : $(BUILDLIBK)/ecrire.o $(BUILDLIBK)/lire.o
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) $(PIC) -shared -o $@ $^

libkey : $(BUILD)/libkey.dylib
```


Exemple

```
$(BUILD)/mainmod.o : $(SRC)/mainmod.c $(INC)/es.h
        @test -d $(BUILD) || mkdir $(BUILD)
        $(COMPILE) -c $< -o $@

$(BUILD)/file : $(BUILD)/mainmod.o $(BUILD)/libfile.dylib
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) -o $@ $< -L$(BUILD) -lfile

$(BUILD)/key : $(BUILD)/mainmod.o $(BUILD)/libkey.dylib
        @test -d $(BUILD) || mkdir $(BUILD)
        $(CC) -o $@ $< -L$(BUILD) -lkey

key : libkey $(BUILD)/key
        echo "Generated: $^"

file : libfile $(BUILD)/file
        echo "Generated: $^"

all : key file

clean :
        echo "Cleaning..."
        rm -rf build
```

```
$ make all
gcc --std=c11 -pedantic -Wall -Iinclude -fPIC  -c libkey/
    ecrire.c -o build/libkey/ecrire.o
gcc --std=c11 -pedantic -Wall -Iinclude -fPIC  -c libkey/
    lire.c -o build/libkey/lire.o
gcc -fPIC  -shared -o build/libkey.dylib build/libkey/ecrire
    .o build/libkey/lire.o
gcc --std=c11 -pedantic -Wall -Iinclude -c src/mainmod.c -o
    build/mainmod.o
gcc -o build/key build/mainmod.o -Lbuild -lkey
Generated: libkey build/key
gcc --std=c11 -pedantic -Wall -Iinclude -fPIC  -c libfile/
    ecrire.c -o build/libfile/ecrire.o
gcc --std=c11 -pedantic -Wall -Iinclude -fPIC  -c libfile/
    lire.c -o build/libfile/lire.o
gcc -fPIC  -shared -o build/libfile.dylib build/libfile/
    ecrire.o build/libfile/lire.o
gcc -o build/file build/mainmod.o -Lbuild -lfile
Generated: libfile build/file
$
```


examen du contenu d'une bibliothèque dynamique

```
$ nm build/libkey.dylib
0000000000000ee0 T _ecrire
0000000000000f10 T _lire
                 U _printf
                 U _puts
                 U _scanf
                 U dyld_stub_binder
$
```

Remarque

On peut remarquer qu'il reste des symboles non-définis (voir liaison dynamique, plus loin).

examen du contenu d'un exécutable lié dynamiquement

```
$ nm build/key
0000000100000000 T __mh_execute_header
                 U _ecrire
                 U _lire
0000000100000f50 T _main
                 U dyld_stub_binder

$
```


Une **exécution dans le répertoire principal du projet** donne :

```
$ build/key
Give a number
12
Read: 12
Give a number
0
Read: 0
Give a number
^D
$
```

Une **exécution dans un autre répertoire que le principal** donne :

```
$ mainmoddyn/build/key
dyld: Library not loaded: build/libkey.dylib
  Referenced from: /Users/yunes/ownCloud/TeX
     /Cours/Systemes/src/mainmoddyn/build/
     key
  Reason: image not found
[1]    36059 trace trap  mainmoddyn/build/
   key
$
```

En effet, cette fois, **à l'exécution**, le chargeur dynamique cherche une bibliothèque de référence `build/libkey.dylib` mais ne la trouve pas...

DYLD_LIBRARY_PATH, LD_LIBRARY_PATH

Definition (DYLD_LIBRARY_PATH, LD_LIBRARY_PATH)

Cette variable (la première sous OSX, la seconde pour *NIX) permet d'indiquer un chemin de recherche pour les bibliothèques liées dynamiquement (c'est-à-dire à l'exécution).

Il faut donc positionner la variable de sorte qu'elle désigne le répertoire contenant la bibliothèque dynamique :

```
$ export DYLD_LIBRARY_PATH=~/TeX/Cours/
   Systemes/src/mainmoddyn/build
$ mainmoddyn/build/key
Give a number
12
Read: 12
Give a number
^D
$
```


Les entrées/sorties du C

Definition (Entrée/sortie)

Une **entrée/sortie** ou E/S (*Input/Output* I/O) désigne l'opération d'un système visant à communiquer avec son extérieur. La définition est très large.
Celle qui nous intéresse ici est la notion d'entrée/sortie pour l'architecture, c'est-à-dire celle de communication entre le processeur et sa mémoire vers le reste des périphériques, en particulier les fichiers...

opérations sur les fichiers

Bien que le language C ne spécifie peu ou pas grand chose à propos des fichiers, son standard définit l'existence de quelques fonctions permettant de :

- supprimer un fichier;
- renommer un fichier;
- obtenir des fichiers temporaires.

⚠ Attention

Le langage C ne permet de manipuler que des fichiers, il ignore le concept de répertoire.

supprimer un fichier `remove`

Definition (`remove`)

Cette opération, dont la syntaxe est :

```
#include <stdio.h>
int remove(const char *filename};
```

permet d'obtenir la suppression dans le système de fichiers hôte, du nom de fichier passé en paramètre. 0 en cas de succès et $\neq 0$ sinon.

⚠ Attention (*NIX)

Comme on le verra plus tard, la suppression d'un nom de fichier ne provoque pas nécessairement la suppression du fichier...

renommer un fichier `rename`

Definition (`rename`)

Cette opération, dont la syntaxe est :

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

permet d'obtenir la modification du nom du fichier spécifié *via old* en celui spécifié *via new*.
Si *new* existait déjà avant cet appel, le comportement n'est pas défini par la norme du C. Le résultat est 0 en cas de succès et $\neq 0$ sinon (dans ce cas l'ancien nom est conservé).

obtenir un nom de fichier temporaire

`tmpnam`

Definition (`tmpnam`)

Cette fonction, de prototype :

```
#include <stdio.h>
char *tmpnam(char *name);
```

permet d'obtenir un nom de fichier dont il est garanti qu'à l'appel aucun fichier de ce nom n'existe déjà. En cas d'échec le retour est le pointeur nul. En cas de succès un pointeur sur une chaîne est renvoyé, mais cette chaîne doit être recopiée en vue de son utilisation. Si l'argument n'est pas nul, il doit correspondre à une zone d'au moins `L_tmpnam` caractères, laquelle est remplie par la fonction et son adresse renvoyée.

Exemple

```
#include <stdio.h>

int main(void) {
  char *n1 = tmpnam(NULL);
  printf("%s\n",n1);
  char *n2 = tmpnam(NULL);
  printf("%s␣%s\n",n1,n2); // Argh!!!
  char nn1[L_tmpnam];
  char nn2[L_tmpnam];
  tmpnam(nn1);
  tmpnam(nn2);
  printf("%s␣%s\n",nn1,nn2);
}
```

```
$ ./tmpnamtest
/var/tmp/tmp.0.CgMGnq
/var/tmp/tmp.1.VWfVyF /var/tmp/tmp.1.VWfVyF
/var/tmp/tmp.2.n4tahD /var/tmp/tmp.3.FszaM1
$
```


Toute manipulation d'objet nécessite l'emploi d'une procédure qui a la forme générale :

1. allocation de la ressource,
2. utilisation de la ressource,
3. libération de la ressource.

Ce qui est important de noter c'est que l'allocation de la ressource, une fois obtenue, garantit la liberté totale quant à l'utilisation et ce jusqu'à la libération de la ressource. L'allocation est une permission d'utiliser...
On notera que même l'utilisation d'une variable en programmation entre dans un tel schéma.

C'est aussi le cas pour les entrées/sorties sur fichier, pour lesquelles il est nécessaire :

1. de requérir l'ouverture du fichier,
2. manipuler le contenu du fichier,
3. rendre la ressource en fermant le fichier.

Au même titre qu'il y a une distinction entre un programme et son exécution (processus), il y a une distinction entre un fichier (l'objet statique) et sa manipulation.

le type `FILE`

Definition (`FILE`)

Le type fondamental de la manipulation des fichiers (disponible via l'inclusion de `stdio.h`) est `FILE` qui représente un **flot** (*stream*) d'entrées/sorties.
Ce type permet d'encapsuler les différentes caractéristiques de la manipulation d'un fichier, c'est-à-dire :

- la position courante de lecture/écriture;
- un éventuel tampon d'entrées/sorties;
- un indicateur d'erreur de lecture/écriture;
- un indicateur de présence de fin de fichier.

Bien qu'il s'agisse d'un type complet, on ne l'utilise jamais qu'à travers un pointeur sur ce type.
L'autre point est qu'il s'agit d'un type opaque, c'est-à-dire que sa structure n'est a-priori pas connue; il n'y a surtout aucune nécessité à la déterminer (sauf à vouloir comprendre comment une implémentation particulière fonctionne). Sa manipulation repose sur l'utilisation de fonctions.

Note

On peut considérer le type `FILE *` comme une référence sur un objet et les fonctions afférentes comme des méthodes de cet objet.

Attention

On ne doit pas copier un type `FILE`, une telle opération a un comportement non défini.

types de flots

La bibliothèque C distingue deux types de flots :

flots de texte (*text streams*) : une suite de lignes. Une ligne de texte étant définie (grossièrement) comme une suite de caractères imprimables terminée par un retour à la ligne (sauf éventuellement pour la dernière ligne).

flots binaires (*binary streams*) : une suite d'octets.

Note

Les systèmes de la famille *NIX ne distinguent pas ces types de fichiers, par conséquent, il n'y a habituellement aucune nécessité à les différencier. Sous WINDOWS c'est une autre paire de manches...

Normalement, un processus hérite de trois flots standard (sauf cas exceptionnel) :

`stdin` ouvert en lecture

`stdout` ouvert en écriture

`stderr` ouvert en écriture

effet des entrées/sorties

Toute entrée/sortie réussie a un effet, en plus d'extraire les informations ou d'écrire les informations voulues à la position courante, la position associée au flot est modifiée; la position est incrémentée de la longueur de la lecture ou de l'écriture (effet secondaire).
On verra qu'il existe la possibilité de modifier cette position, indépendamment d'une lecture ou d'une écriture.

entrées/sorties formatées

Ces entrées/sorties permettent d'obtenir la lecture ou l'écriture de chaînes de caractères et d'en contrôler le formatage :

Definition (Larousse)

Formater : Mettre en forme un texte en vue de son impression selon des caractéristiques données de mise en pages.

Les fonctions de la bibliothèque C qui s'en chargent sont celles de la famille `printf` et `scanf`. Elles s'utilisent normalement avec des flots de texte.

`printf, scanf`

Definition (`printf, scanf`)

Ces fonctions permettent de réaliser (respectivement) des sorties ou des entrées formatées sur les flots `stdout` et `stdin` :

```
#include <stdio.h>
int printf(const char *format,...);
int scanf(const char *format,...);
```

Un appel à `printf(...);` est équivalent à un appel à `fprintf(stdout,...);`.
Un appel à `scanf(...);` est équivalent à un appel à `fscanf(stdin,...);`.

les formats de `printf`

Le *format* spécifié à l'appel peut contenir des caractères ordinaires et des spécifications de conversion. À chaque conversion doit correspondre, dans l'ordre, un argument du type correspondant.
Une conversion est :

- introduite par le caractère `%`, pour afficher un `%` il est nécessaire de le doubler...
- suivi de la spécification de variantes (optionnelle),
- d'une taille de champ (optionnelle),
- d'une précision (optionnelle),
- du modificateur de longueur (optionnel) et d'une conversion.

Note

Nous renvoyons le lecteur à la documentation en ligne pour obtenir les détails les plus complets et nous nous contenterons des fonctionnalités les plus utilisées.

variantes pour `printf`

- permet d'obtenir la justification à gauche;
+ permet d'obtenir la présence systématique du signe;
0 permet d'obtenir le bourrage du champ par des 0.

taille de champ pour `printf`

un entier pour spécifier une longueur de champ d'affichage;

* qui permet que la longueur soit spécifiée par un argument.

précision pour `printf`

un entier pour spécifier une longueur minimale pour l'affichage des entiers, et après la virgule pour les flottants;

`*` qui permet que la précision soit spécifiée par un argument.

longueurs pour format de `printf`

`l` pour spécifier que l'argument est un **`long`** (signé ou non);

`ll` pour spécifier que l'argument est un **`long long`** (signé ou non);

`z` pour spécifier que l'argument est de type `size_t`;

`L` pour spécifier que l'argument est de type **`long double`**.

conversions de `printf`

- `d, i` pour afficher un **`int`**.
- `o, u, x, X` pour afficher un **`unsigned int`** en octal, décimal ou hexadécimal.
- `f, F` pour afficher un **`double`** en notation flottante.
- `e, E` pour afficher un **`double`** en notation scientifique.
- `c` pour afficher un **`int`** sous la forme d'un caractère du code ASCII correspondant.
- `s` pour afficher une chaîne de caractères pointée par un `char *`.
- `p` pour afficher un **`void *`**
- `n` pour écrire dans la variable de type **`int`** pointée par l'argument le nombre de caractères déjà écrits par l'appel courant.

Un exemple plutôt qu'un long discours :

Exemple

```
#include <stdio.h>

int main(void) {
  int i = 123;
  printf("|%d|\n",i);
  printf("|%8d|\n",i);
  printf("|%-8d|\n",i);
  printf("|%-+8d|\n",i);
  printf("|%+08d|\n",i);
  for (int l=0; l<8; l++) {
    printf("|%*d|\n",l,i);
  }
  printf("|%+8.5d|\n",i);
  for (int p=0; p<8; p++) {
    printf("|%8.*d|\n",p,i);
  }
}
```

```
$ ./printfex
|123|
|      123|
|123      |
|+123     |
|+0000123|
|123|
|123|
|123|
|123|
|  123|
|   123|
|    123|
|     123|
|   +00123|
|      123|
|      123|
|      123|
|      123|
|     0123|
|    00123|
|   000123|
|  0000123|
$
```


la lecture par `scanf`

La fonction `scanf` est assez compliquée à utiliser si l'on souhaite contrôler avec exactitude la lecture, et son fonctionnement compliqué ne l'est pas assez pour obtenir de bonnes analyses lexicales; par conséquent nous ferons l'impasse sur son fonctionnement détaillé.

E/S formatées avec la mémoire

Les fonctions «habituelles» de lectures et écritures formatées permettent de lire ou d'écrire depuis ou vers des flots. La bibliothèque offre aussi la possibilité de réaliser ces opérations depuis ou vers la mémoire. L'utilisation courante est de formater des chaînes de caractères et de «parser» celles-ci. Les fonctions sont :

```
#include <stdio.h>
int snprintf(char *s, size_t n, const char *
    format, ...);
int sprintf(char *s, const char *format,
    ...);
int sscanf(const char *s, const char *format
    , ...);
```


E/S formatées avec liste d'arguments

Pour les fonctions à nombre variable d'arguments, se reporter à la page **??**.

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg)
    ;
int vscanf(const char *format, va_list arg);
int vsnprintf(char *s, size_t n, const char
    *format, va_list arg);
int vsprintf(char *s, const char *format,
    va_list arg);
int vsscanf(const char *s, const char *
    format, va_list arg);
```

Les entrées/sorties de caractères

Il s'agit de lectures/écritures de suites ordonnées de **`char`** sans opération de formatage.
Ces opérations fonctionnent habituellement plus naturellement sur des flots de type texte.

lecture de caractères `getchar`

Definition (`getchar`)

```
#include <stdio.h>
int getchar(void);
```

permet d'obtenir la lecture d'un caractère sur l'entrée standard. Un appel à cette fonction est équivalent à un appel à `getc(stdin);`.
En cas de succès, l'entier envoyé est réductible à un **`char`**, en cas d'échec la valeur est `EOF` et si la fin de fichier a été détectée l'indicateur de fin de fichier du flot est positionné, alors que si c'est une erreur c'est l'indicateur d'erreur.

⚠ Attention

La fonction `gets` a été supprimé du standard depuis c11. Elle est considérée comme trop dangereuse car permet trop facilement les attaques par débordement (**buffer overflow**).

l'écriture de caractères `putchar`

Definition (`putchar`)

```
#include <stdio.h>
int putchar(int c);
```

permet d'obtenir l'écriture du caractère `c` sur la sortie standard. Un appel à cette fonction est équivalent à un appel à `putc(c,stdin);`.
En cas de succès, l'entier renvoyé est le **`char`** passé en paramètre, en cas d'échec la valeur est `EOF` et l'indicateur d'erreur est positionné sur le flot.

l'écriture de chaîne puts

Definition (puts)

```
#include <stdio.h>
int puts(const char *s);
```

permet d'obtenir l'écriture de la chaîne pointée par s sur la sortie standard suivie d'un passage à la ligne. Un appel à cette fonction est équivalent à un appel à `fputs(s,stdout); fputc('\n',stdout);`.
En cas de succès, l'entier renvoyé est non nul, en cas d'échec la valeur est EOF et l'indicateur d'erreur est positionné sur le flot.

Exemple

```
#include <stdio.h>

int main(void) {
  int c;
  while ( (c=getchar()) != EOF)
    putchar(c);
}
```

```
$ mycat
toto  #tapé
toto
fsdhfjkdslfhsl  #tapé
fsdhfjkdslfhsl
^D  #fin de flot
$
```

accès aux fichiers

L'accès aux fichiers est obtenu *via* les flots que l'on peut manipuler *via* les fonctions :

- `fopen` afin d'obtenir la création d'un flot associé à un fichier;
- `fclose` afin de libérer un flot;
- `freopen` afin de réassocier un flot existant à un nouveau fichier.

`fopen`

L'obtention d'un flot `FILE *` s'effectue via un appel à :

Definition (`fopen`)

```
#include <stdio.h>
FILE *fopen(const char *filename, const char
    *mode);
```

permet d'obtenir un flot associé au fichier de nom *filename* dans le *mode* considéré. Les modes valides sont : r, w, wx, a, rb, wb, wbx, ab, r+, w+, w+x, a+, r+b/rb+, w+b/wb+, w+bx/wb+x, a+b, ab+.
Les modes de base sont r, w, a (read, create, append) que l'on peut combiner différemment avec b, x, + (binary, exclusive, update).
Si l'ouverture réclamée n'est pas possible, la fonction renvoie le pointeur nul.

fclose

Definition (fclose)

```
#include <stdio.h>
int fclose(FILE *flot);
```

permet d'obtenir la fermeture du flot précédemment ouvert. En cas de succès la valeur 0 est renvoyée, sinon EOF.
La fermeture provoque l'écriture physique de toutes les écritures précédentes et qui n'auraient pas été encore effectivement réalisées (voir tampon page **??**).

freopen

Definition (freopen)

```
#include <stdio.h>
FILE *freopen(const char *nom, const char *
   mode, FILE *flot);
```

permet d'obtenir l'ouverture du fichier de nom donné dans le mode spécifié sur le flot indiqué. Si le *nom* passé est le pointeur nul, une tentative de réouverture du fichier précédemment associé au *flot* dans le *mode* indiqué est réalisée.
En cas d'échec le pointeur nul est renvoyé, et le flot sinon.

Note

Cette fonction est un équivalent «interne» de la redirection des shells.

Une fois un flot obtenu, on peut alors utiliser différents fonctions de lecture ou écriture, voire modifier certaines de ces caractéristiques.

E/S formatées sur flots explicites : `fprintf`, `fscanf`

Definition (`fprintf`, `fscanf`)

Leur prototype est :

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format
    , ...);
int fscanf(FILE *stream, const char *format,
    ...);
```

Elles permettent de réaliser des entrées/sorties à l'identique des fonctions `printf` et `scanf` avec spécification du flot concerné. Les opérations s'effectuant alors depuis ou vers le fichier associé au flot.

E/S formatées sur flots explicites : `vfprintf`, `vfscanf`

Definition (`fprintf`, `fscanf`)

Leur prototype est :

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *
    format, va_list ap);
int vfscanf(FILE *stream, const char *format
    , va_list arg);
```

Elles permettent de réaliser des entrées/sorties à l'identique des fonctions `vprintf` et `vscanf` avec spécification du flot concerné. Les opérations s'effectuant alors depuis ou vers le fichier associé au flot.

lecture de caractères sur flots explicites : `fgetc`, `getc`

Definition (lecture sur flot explicite)

```
#include <stdio.h>
int fgetc(FILE *flot);
int getc(FILE *flot);
```

Les fonctions `fgetc` et `getc` permettent de lire un caractère sur le *flot*. En cas de succès le caractère est renvoyé, sinon `EOF` et les indicateurs adéquats positionnés en conséquence (voir `getchar`, page 602 et indicateurs page **??**). `getc` peut être disponible comme macro-définition du pré-processeur.

lecture de caractères sur flots explicites : ungetc

Definition (*pushback* ungetc)

```
#include <stdio.h>
int ungetc(int c, FILE *flot);
```

La fonction ungetc permet de replacer un caractère dans le flot, qui sera donc lu à la prochaine lecture. Un seul appel entre lectures est garanti. Attention, cela ne provoque pas l'écriture dans le fichier associé... En cas de succès le caractère est renvoyé, et EOF sinon.

Exemple

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

void cpy(FILE *in) {
  int c;
  while ( (c=getc(in)) != EOF)
    putchar(c);
}

int main(int argc,char *argv[]) {
  if (argc==2) {
    FILE *f = fopen(argv[1],"r");
    if (f==NULL) {
      fprintf(stderr,"Can't open %s\n",argv[1]);
      exit(1);
    }
    cpy(f);
    fclose(f);
  } else
    cpy(stdin);
}
```

```
$ ./mybettercat
fsdhfkj
fsdhfkj
eamzlieu
eamzlieu
$ ./mybettercat data.txt
123
666
999
42
$
```


écriture de caractères sur flots explicites : `fputc, putc`

Definition (lecture sur flot explicite)

```
#include <stdio.h>
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
```

Les fonctions `fputc` et `putc` permettent d'écrire un caractère sur le *flot*. En cas de succès le caractère est renvoyé, sinon `EOF` et les indicateurs adéquats positionnés en conséquence (voir `putchar`, page 604 et indicateurs page **??**). `putc` peut être disponible en tant que macro-définition du pré-processeur.

Exemple

```c
// mybettercat2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void cpy(FILE *in,FILE *out) {
  int c;
  while ( (c=getc(in)) != EOF)
    putc(c,out);
}

int main(int argc,char *argv[]) {
  FILE *in=stdin, *out=stdout;
  int flag = 0;
  if (argc==1) {
    cpy(in,out);
    exit(0);
  }
  if (!strcmp(argv[1],"-o")) {
    if (argc>2) {
      out = fopen(argv[2],"w");
      if (out==NULL) {
        fprintf(stderr,"Can't open %s for writing\n",argv[2]);
        exit(1);
      }
      argv += 2;
      argc -= 2;
    } else {
      fprintf(stderr,"No file for output\n");
      exit(1);
    }
  }
  if (argc==1) {
    cpy(in,out);
    exit(0);
  }
  while (--argc) {
    in = fopen(*++argv,"r");
    if (in==NULL) {
      fprintf(stderr,"Can't open %s for reading\n",argv[1]);
      flag = 1;
    }
    cpy(in,out);
    fclose(in);
  }
  exit(flag);
}
```



```
$ ./mybettercat2 -o /tmp/toto data.txt data.
   txt
$ cat /tmp/toto
123
666
999
42
123
666
999
42
$
```

lecture de chaine sur flot explicite : `fgets`

Definition (`fgets`)

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *flot);
```

permet de lire une chaîne de caractères depuis le flot. `fgets` lit au plus $n-1$ caractères depuis le *flot* et s'arrête sinon à une fin de ligne ou sur la fin de fichier et place le résultat dans *s*. Un caractère nul ASCII 0 est écrit dans la chaîne. Si aucun caractère n'est lu car le flot est en fin de fichier, le pointeur nul est renvoyé, comme lorsqu'une erreur est rencontrée. En cas de succès la chaîne lue est renvoyée.

écriture de chaine sur flot explicite : `fputs`

Definition (`fputs`)

```
#include <stdio.h>
int fputs(const char *s, FILE *flot);
```

`fputs` écrit la chaîne pointée par *s* sur le *flot*. En cas de succès une valeur non nulle est renvoyée, en cas d'échec c'est `EOF`.

Exemple

```c
// exgets.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc,char *argv[]) {
  if (argc>2) {
    fprintf(stderr,"Too much args\n");
    exit(1);
  }
  FILE *in = stdin;
  if (argc==2) {
    in = fopen(argv[1],"r");
    if (in==NULL) {
      fprintf(stderr,"Can't open %s for reading\n",argv[1]);
      exit(1);
    }
  }
  char name[100];
  char ages[100];
  int age=0;
  if (argc==1) fputs("Your name ? ",stdout);
  if (fgets(name,sizeof(name),in)==NULL) {
    fputs("Missing name\n",stderr);
    exit(1);
  }
  name[strlen(name)-1] = '\0';
  if (argc==1) fputs("How old are you ? ",stdout);
  if (fgets(ages,sizeof(ages),in)==NULL) {
    fputs("Missing age\n",stderr);
    exit(1);
  }
  sscanf(ages,"%d",&age);
  printf("Hello %s, you are %s\n",name,age<=18?"young":"old");
  exit(0);
}
```



```
$ ./exgets
Your name ? jb
How old are you ? 99
Hello jb, you are old
$ cat data.txt
123
666
999
42
$ ./exgets data.txt
Hello 123, you are old
$
```

fichier temporaire

Definition (`tmpfile`)

```
#include <stdio.h>
FILE *tmpfile(void);
```

permet d'obtenir l'ouverture d'un fichier temporaire unique en mode `"wb+"` et qui sera automatiquement supprimé à sa fermeture ou à la terminaison du programme. En cas d'échec la fonction renvoie le pointeur nul.

entrées/sorties brutes

Ces entrées/sorties permettent de lire ou écrire n'importe quel contenu depuis ou vers la mémoire, depuis ou vers un fichier. Il n'y a aucune hypothèse sur les contenus, se sont de simples suites d'octets bruts. Il s'agit généralement de manipuler des flots binaires.

lecture brute : `fread`

Definition (`fread`)

Les fonctions :

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t
   nmemb, FILE *flot);
```

permet de lire *nmemb* éléments, chacun de taille *size*, depuis le *flot* et de placer le résultat à l'adresse *ptr*. Le retour de la fonction est le nombre d'éléments effectivement lus. Si la valeur est inférieure à la demande, il s'agit soit d'une erreur, soit d'une fin de fichier prématurée.
Cette fonction modifie la position courante du flot en conséquence.

écriture brute : `fwrite`

Definition (`fwrite`)

Les fonctions :

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
   size_t nmemb, FILE *flot);
```

permet d'écrire *nmemb* éléments, chacun de taille *size*, vers le *flot* en récupérant les données à l'adresse *ptr*. Le retour de la fonction est le nombre d'éléments effectivement écrit. Pour `fwrite`, une valeur inférieure à la demande indique une erreur d'écriture.
Cette fonction modifie la position courante du flot en conséquence.

Exemple

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

int main(int argc,char *argv[]) {
  if (argc!=3) exit(1);
  FILE *r = fopen(argv[1],"r");
  if (r==NULL) exit(2);
  FILE *w = fopen(argv[2],"w");
  if (w==NULL) fclose(r), exit(3);

#define L 1000
  char data[L];
  size_t nr;
  while ((nr=fread(data,1,L,r))>0)
    if (fwrite(data,1,nr,w)!=nr) fclose(r), fclose(w), exit(3);
  fclose(r);
  fclose(w);
  exit(0);
}
```



```
$ ./mycp mycp.c toto.c
$ diff mycp.c toto.c
$ ./mycp mycp.c toto.c
$ diff mycp.c toto.c
$ ls -ail mycp.c toto.c
53321646 -rw-r--r--+ 1 yunes  staff  404 19
   oct 20:42 mycp.c
53321692 -rw-r--r--+ 1 yunes  staff  404 19
   oct 20:43 toto.c
$
```

Exemple

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

int main(void) {
  int array[10];
  srandom(0);
  for (int i=0; i<10; i++) array[i] = random()%10;
  for (int i=0; i<10; i++) printf("%8d␣",array[i]);
  putchar('\n');

  char *filename = tmpnam(NULL);
  if (filename==NULL) exit(1);
  FILE *file = fopen(filename,"w");
  if (file==NULL) exit(2);
  size_t w = fwrite(array,sizeof *array,sizeof array/sizeof *array,
       file);
  if (w!=sizeof array/sizeof *array) {
    fprintf(stderr,"Bad␣write\n");
    fclose(file);
    exit(3);
  }
  fclose(file);

  file = fopen(filename,"r");
  if (file==NULL) exit(4);
  int i;
  while (fread(&i,sizeof i,1,file)==1)
    printf("%8d␣",i);
  putchar('\n');
  fclose(file);
  remove(filename);
}
```



```
$ ./le
       1       7       3       5       9
               5       2       2
               6       5
       1       7       3       5       9
               5       2       2
               6       5
$
```

le positionnement dans le flot

À beaucoup de flots sont associés une position courante qui représente la position de la tête de lecture ou écriture dans la séquence des octets qui constituent le contenu du flot.
Il existe donc des fonctions permettant de consulter ou modifier cette propriété au gré des besoins.

positionnement par : `fgetpos`, `fsetpos`

Definition (`fgetpos`, `fsetpos`)

```
#include <stdio.h>
int fgetpos(FILE *flot, fpos_t *pos);
int fsetpos(FILE *flot, const fpos_t *pos);
```

Ces fonctions permettent d'obtenir/positionner la position courante du *flot* spécifiée *via pos*.
Le type `fpos_t` est un type opaque, on peut l'interpréter comme un marqueur de position (sans unité). Pour un appel à `fsetpos`, on ne doit utiliser que des valeurs obtenues par appel à `fgetpos`.

Exemple

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

int main(void) {
  srandom(0);

  char *filename = tmpnam(NULL);
  if (filename==NULL) exit(1);
  FILE *file = fopen(filename,"w+");
  if (file==NULL) exit(2);
  int p = random()%10;
  fpos_t pos;
  for (int j=0; j<10; j++) {
    int i = random()%100;
    if (j==p) fgetpos(file,&pos);
    if (fwrite(&i,sizeof i,1,file)!=1) {
      fprintf(stderr,"Bad write\n");
      fclose(file);
      remove(filename);
      exit(3);
    }
    printf("%8d ",i);
  }
  putchar('\n');

  fsetpos(file,&pos);
  int i;
  if (fread(&i,sizeof i,1,file)!=1) {
    fprintf(stderr,"Bad read\n");
    fclose(file);
    remove(filename);
    exit(4);
  }
  printf("At %d %d\n",p,i);
  fclose(file);
  remove(filename);
}
```

38


```
$ ./lepos
      17       63       95       79       95
                62       32       56
          45       37
At 1 63
$
```

lecture de la position avec : `ftell`

Definition (`ftell`)

```
#include <stdio.h>
long int ftell(FILE *stream);
```

permet d'obtenir la position courante du *flot*. Pour les flots binaires la position renvoyée est le nombre de caractères qui séparent le début du flot de la position courante, pour les flots de type texte c'est une mesure d'unité non spécifiée. En cas d'échec la valeur retournée est -1L.

positionnement par : `fseek, rewind`

Definition (`fseek`)

```
#include <stdio.h>
int fseek(FILE *stream, long int pos, int whence);
void rewind(FILE *stream);
```

La fonction `fseek` permet de modifier la position courante du *flot* et de positionner celle-ci *via pos* et relativement à *whence* :

- `SEEK_SET` pour une position absolue;
- `SEEK_CUR` pour une position relative à la position courante;
- `SEEK_END` pour une position relative à la fin du flot.

La valeur renvoyée est 0 en cas de succès et $\neq 0$ en cas d'échec.
Un appel à `rewind(stream);` est équivalent à un appel à `(void)fseek(stream,0L,SEEK_SET);`.

Exemple

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

int main(void) {
  srandom(0);

  char *filename = tmpnam(NULL);
  if (filename==NULL) exit(1);
  FILE *file = fopen(filename,"wb+");
  if (file==NULL) exit(2);
  for (int j=0; j<10; j++) {
    int i = random()%100;
    if (fwrite(&i,sizeof i,1,file)!=1) {
      fprintf(stderr,"Bad write\n");
      fclose(file);
      remove(filename);
      exit(3);
    }
    printf("%8d ",i);
  }
  putchar('\n');

  long int pos = -1;
  int i;
  while (fseek(file,pos*sizeof i,SEEK_END)==0) {
    long int realpos = ftell(file);
    if (fread(&i,sizeof i,1,file)!=1) {
      fprintf(stderr,"Bad read\n");
      fclose(file);
      remove(filename);
      exit(4);
    }
    printf("at %ld %ld %d; ",realpos,pos,i);
    pos--;
  }
  putchar('\n');
  fclose(file);
  remove(filename);
}
```



```
$ ./leback
       1       17       63       95       79
                95       62       32
           56       45
at 36 -1 45; at 32 -2 56; at 28 -3 32; at 24
    -4 62; at 20 -5 95; at 16 -6 79; at 12
   -7 95; at 8 -8 63; at 4 -9 17; at 0 -10
   1;
$
```

fichiers à trous

Note

Cette terminologie peut prêter à confusion, aucun fichier ne comporte jamais de trou d'un point de vue logique; un fichier est définitivement une séquence ordonnée d'octets d'une certaine longueur. Il n'y a donc pas de «trou».
C'est la façon de créer la séquence qui entretient la confusion car il est possible de déplacer la position courante d'un flot au-delà de la «fin de fichier» et de tenter d'écrire. En ce cas, si l'écriture est possible, entre la fin de fichier et la position à laquelle l'écriture a été réalisée des octets de valeur 0 sont «ajoutés».

Exemple

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

int main(int argc,char *argv[]) {
  if (argc!=2) exit(1);
  FILE *f = fopen(argv[1],"w");
  if (f==NULL) { perror(argv[0]); exit(1); }
  if (fputc('a',f)==EOF) { fclose(f); exit
     (1); }
  if (fseek(f,10L,SEEK_SET)!=0) { perror(
     argv[0]); exit(1); }
  if (fputc('a',f)==EOF) { fclose(f); exit
     (1); }
  fclose(f);
  exit(0);
}
```

```
$ ls -l /tmp/toto.data
ls: /tmp/toto.data: No such file or
   directory
$ ./hole /tmp/toto.data
$ ls -l /tmp/toto.data
-rw-r--r--  1 yunes  wheel  11 11 oct 12:16
   /tmp/toto.data
$ od -c /tmp/toto.data
0000000    a  \0  \0  \0  \0  \0  \0  \0  \0
     \0   a
0000013
$
```