Programmation en C
Cours 5

Licence Maths-Info
Aix-Marseille Université
2011-2012

Valentin Emiya
valentin.emiya@lif.univ-mrs.fr

13 février 2012

Lundi dernier

* Les pointeurs : notions de base
* Les opérations liées aux pointeurs
* Arithmétique sur les pointeurs
* Pointeurs et passage de paramètres par adresse
* Pointeurs et tableaux
* Pointeurs de structures et d'union
* Les paramètres de la fonction main

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

Pratique du C
Complément sur les pointeurs

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2009-2010

L'opérateur `sizeof` et le type `size_t`

Le type `size_t` : défini (dans `stddef.h`) pour toute variable devant contenir une taille.

L'opérateur `sizeof` :

* 2 syntaxes : `sizeof(`*`descripteur_de_type`*`)`
  `sizeof` *`expression`*
* Renvoie, comme expression de type `size_t`, la taille en mémoire correspondant au *descripteur de type* ou à l'*expression*.
* Exemple :

```
size_t s ;
int n ;
s = sizeof(int) ;
s = sizeof n ;
s = sizeof &n ;
```

malloc et free

Ces fonctions nécessitent l'inclusion de l'entête stdlib.h et manipulent un segment de mémoire associé au processus (appelé le tas).

Fonction d'allocation dynamique de mémoire :

- fonction `malloc` de la librairie standard ;
- réserve un espace mémoire dans le tas du processus ;
  - `void *malloc(size_t size)`
    réserve `size` octets dans le tas et retourne un pointeur sur la zone allouée (`NULL` en cas d'echec).

Fonction de désallocation de mémoire ;

- fonction `free` de la librairie standard ;
  - `void free(void *ptr)`
    libère une zone allouée par un précédent `malloc`.
    `ptr` doit obligatoirement être un pointeur retourné par un précédent `malloc`.

Fonctions malloc et calloc

`void *malloc(size_t taille)`
Allocation d'un espace pouvant contenir un objet de la taille indiquée. Renvoie l'adresse, et `NULL` en cas d'échec. Pas d'initialisation.

`void *calloc(size_t nombre, size_t taille)`
Allocation d'un espace pouvant contenir `nombre` objets de la taille indiquée. Renvoie l'adresse, et `NULL` en cas d'échec. *Initialisation à 0*.
≡ `malloc(nombre*taille)` puis initialisation

Conversion de type

Petit rappel sur le forçage de type : coercition (cast)

- force la conversion de type de la valeur d'un expression :
  (*type*) *expression*
- ne peut être une valeur gauche.

Petit rappel sur la taille d'un objet : opérateur `sizeof`

- `sizeof`(*identificateur_de_type*)
  donne la taille en octets de tout objet de type *identificateur_de_type* ;
- Avec beaucoup de précaution, on peut utiliser
  `sizeof` *expression*
  qui donne la taille en octets de son opérande `expression`. Mais attention :

```
char *ch = "Hello world" ; /*comment est-ce stock\'e~?*/
int main(void){
  char *chlocal = "Hello world" ; /* idem */
  return sizeof(ch) ; /* que retourne cette fonction~?*/
}
```

Précautions à prendre : exemple

```
char * s = "bonjour!!", *r;
char t[] = "bonjour!!";

printf("%d\n",(int) sizeof(s));
printf("%d\n",(int) sizeof(t));
r = s;
printf("%d\n",(int) sizeof(r));
r = t;
printf("%d\n",(int) sizeof(r));
```

```
Affichage :

8
10
8
8
```

Allocation dynamique : exemple simple

```c
#include <stdlib.h>

/* ... */

{
  double *p;
  p = malloc(3*sizeof(double));
  p[0] = 1.1; p[2] = 1.2; p[3] = p[0]+p[1];
  /* ... */
  free(p);
  p = malloc(5*sizeof(double));
  p[0] = 10; p[5] = -100.1;
  /* ... */
  free(p);
}
```

Allocation dynamique : exemple 2

```c
#include <stdlib.h>

double *creeTableauDouble(int k){
    return malloc(k*sizeof(double));
}

int main(){
  double *p;
  p = creeTableauDouble(3);

  p[0] = 1.1; p[1] = 1.2; p[2] = p[0]+p[1];
  /* ... */
  free(p);
  p = creeTableauDouble(5);

  p[0] = 10; p[4] = -100.1;
  /* ... */
  free(p); return 0;
}
```

Allocation dynamique : exemple 3

```c
#include <stdlib.h>

typedef struct {
  int ID;
  int *cartes;
} joueur;

joueur *creeJoueur(int id, int nb){
  int k;
  joueur *j = malloc(sizeof(joueur));
  if (!j)
    exit(EXIT_FAILURE);
  j->ID = id;
  j->cartes = malloc(nb*sizeof(int));
  if (!j->cartes)
    exit(EXIT_FAILURE);
  for (k=0; k<nb; k++)
      j->cartes[k] = 0;
  return j;
}
```

```c
void
supprimeJoueur(joueur *j)
{
    free(j->cartes);
    free(j);
}


int main(void){

    joueur *j1, *j2;
    j1 = creeJoueur(1,5);
    j2 = creeJoueur(2,8);
    /* ... */
    supprimeJoueur(j1);
    supprimeJoueur(j2);
    return 0;
}
```

Bonnes pratiques

- Couplage (1) : utiliser `free` à chaque utilisation de `malloc`
- Couplage (2) : idem pour les fonctions "`créer...`" et "`supprimer...`" de plus haut-niveau
- Tester le succès de `malloc` (`exit(EXIT_FAILURE)` sinon)

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

Déclarations des objets en C

- Variable simple : `type nom ;`
- Tableau : `type nom[taille] ;`
- Fonction : `type1 nom(type2,type3,…);`

3 déclarations similaires : «nom» est de type «…»

Une fonction=objet typé

On veut manipuler les fonctions comme les autres objets typés : déclarer des variables « fonctions », les passer en paramètre à d'autres fonctions, etc.

Une fonction en C est

- un objet de première classe : directement manipulable ;
- avec un déclarateur postfixe `()` : `int sqr(int x) ;`
- son fonctionnement est analogue à celui des tableaux :

Déclaration d'un tableau de 5 entiers
`int ar[5] ;`
`temp = ar[i] ;`
déréférencement du pointeur d'entiers `ar` et accès à son élément `i`.
	Déclaration d'une fonction entière à valeur entière
	`int sqr(int x) ;`
	`temp = sqr(i) ;`
	déréférencement du pointeur de fonction `sqr` et appel avec le paramètre `i`.

L'identificateur d'une fonction en C est associé à un pointeur de fonction constant qui pointe sur elle même.

Plus précisément, le nom d'une fonction est un pointeur de fonction constant sur le début du code correspondant à cette fonction.

```
int                          .text
foo                          .globl foo .type foo,@function
(int bar)              foo:   .......
{                            incl     4(%esp)
   return ++bar ;            movl     4(%esp), %eax
}                            ret
                            .globl main
int                         .type    main,@function
main                   main:   ......
(void)                      pushl      $3
{                           call       foo
foo(3) ;                    addl       $16, %esp
return 0;                   movl       $0, %eax
}                           ret
```

Les pointeurs de fonctions : déclaration

- identique au prototype en rajoutant une `*` ;
- déclarer le type retourné et le type des arguments ;
- attention à la priorité : opérateur droit $<<$ opérateur gauche.

Exemple de déclaration :

- `int (*pf)(int, int)` : pointeur de fonction retournant un entier et prenant deux entiers en paramètre ;
- `int *f(int, int)` : fonction retournant un pointeur sur un entier.

```
int (*pfoo)(int) = foo ;
```

```
 /* pfoo = &foo est aussi */
 /* valide mais peu clair */
```

```
    .globl pfoo
           .data
           .align 4
           .type   pfoo,@object
           .size   pfoo,4
pfoo: .long   foo
```

Déclaration d'un synonyme (typedef)

Comme pour les autres déclarations, il est possible de déclarer un type associé aux fonctions comme suit :

```
typedef int fct_t(int) ;
```

Il est ainsi possible de déclarer des types associés aux pointeurs de fonctions

```
typedef fct_t * fctp1_t ;
typedef int (*fctp2_t)(int) ; /* sans utiliser fct_t */
```

L'utilisation de ces types se fait classiquement :

```
int fct(int par) { return par+1 ; }
fctp1_t ftcpv1 ;
fctp2_t ftcpv2 ;
fct_t * ftcpv3 ;
ftcpv1 = fct ;
ftcpv2 = fct ;
ftcpv3 = fct ;
```

Les pointeurs de fonctions : affectation

Opérations sur les pointeurs de fonctions

- affectation d'un pointeur de fonction à :
  - un nom de fonction (pointeur constant) ;
  - une variable de type pointeur de fonction ;
  - les types retournés doivent être *identiques*.
- Exemple d'assignation :

```
int sqr(int x) {
  return x*x;
}
float fsqr(float x) {
  return x*x;
}
int (*pfint1)(int), (*pfint2)(int);

pfint1 = sqr;
pfint2 = pfint1;
/* pfint2 = fsqr; ILLEGAL */
```

Les pointeurs de fonctions : appel

- appel de la fonction pointée : opérateur ()
  - déréférencer le pointeur de fonction ;
  - appeler la fonction pointée en donnant la liste des arguments entre () ;
  - l'expression est du type retourné par la fonction ;
  - le déréférencement est facultatif en C-ANSI.
- Exemple d'appel

```
/* Avec les d\'eclarations du
   transparent pr\'ec\'edent */
int i;                          /* int tab[2]={666,999} ;
                                /* int *p = tab ; */
i = sqr(12);                    /* i = p[1] ;   */
i = (*pfint1)(12);
i = pfint1(12); /* C-ANSI */
```

Exemple : moyenne d'une fonction sur un segment

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

double moyenne(double (*f)(double), int N, double a,
double b){
    int n; double r=0, x;
    for (n=0;n<N;n++){
        x = rand()*(b-a)/RAND_MAX+a;
        r += f(x)/N;
    }
    return r;
}

int main(void){
    printf("%g\n",moyenne(cos,1000,-1,1));
    printf("%g\n",moyenne(sin,1000,-1,1));
    return 0;
}
```

Menu de fonctions

```
   struct COMMANDE {
      char *nom ;
      void (*fun) (char *) ;
   } MENU [] = {    /* on suppose que ls est une     */
      {"ls", ls},   /* fonction d\'eclar\'ee          */
      {"cd", cd},   /* de prototype void ls(char *); */
      {"more", more} ,  /* idem pour cd, more et cat */
      {"cat",  cat},
      {0,0}
     } ;
void executer (char *commande, char *argument)
{    /* strcmp i.e. string compare */
   struct COMMANDE *p = MENU ;
   while (p->nom && strcmp (p->nom, commande)) p++ ;
   if (p->nom) {
      (*p->fun) (argument) ;
   } else fprintf (stderr, "%s : commande inconnue\n",
                           commande) ;
}
```

Fonction quicksort de la librairie standard

```
extern void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));
   typedef struct {
     char *nom;
     int note;
   } Etudiant;
   int inferieur(const Etudiant *p1, const Etudiant *p2) {
     if (p1->note < p2->note)
       return -1;
     else
       if (p1->note == p2->note)
         return(strcmp(p1->nom, p2->nom));
       else
         return 1;
   }
   Etudiant t[250];
   qsort(t, 250, sizeof(Etudiant), inferieur);
```

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

Dans la déclaration `int *(*(*x)())[5] ;` :

- `(*x)` : x est un pointeur...
- `(*x)()` : de fonction qui retourne...
- `(*(*x)())` : un pointeur sur...
- `(*(*x)())[5]` : un tableau de 5...
- `int *(*(*x)())[5] ;` : pointeurs d'entiers.

Problème des déclarations complexes :

- l'opérateur pointeur `*` est préfixe ;
- les opérateurs tableau `[]` et fonction `()` sont postfixes ;
- l'identificateur d'une déclaration est noyé dans des opérateurs.

Pour s'en sortir, on utilise la méthode suivante :

- partir de l'identificateur d'une variable (ou d'un type) ;
- construire le type de l'intérieur vers l'extérieur ;
- en appliquant les règles suivantes :
  - les opérateurs `[]` et `()` ont une plus grande priorité que l'opérateur `*` ;
  - les opérateurs `[]` et `()` se groupent de gauche à droite, alors que les opérateurs `*` se groupent de droite à gauche.

Exemple : `struct s (*(*(*x)[])())[] ;`

Plus simplement, il convient d'utiliser des synonymes (typedef) pour simplifier les déclarations.

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

En première approximation :

- ce sont des chaînes de caractères stockées par le système dans la zone de données statiques ;
- `argc` : nombre d'arguments (nom de commande compris) ;
- `argv` : tableau de chaînes de caractères, correspondant aux arguments, nom de commande compris ;
- passés comme arguments `main` :

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

- Exemple d'utilisation

```
int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr,"Usage: %s argument\n", argv[0]);
    return 1 ;
  }
  if (!(strcmp(argv[1], "-p")) {...} /* option -p */
  if (!(strcmp(argv[1], "-r")) {...} /* option -r */
  return 0 ;
}
```

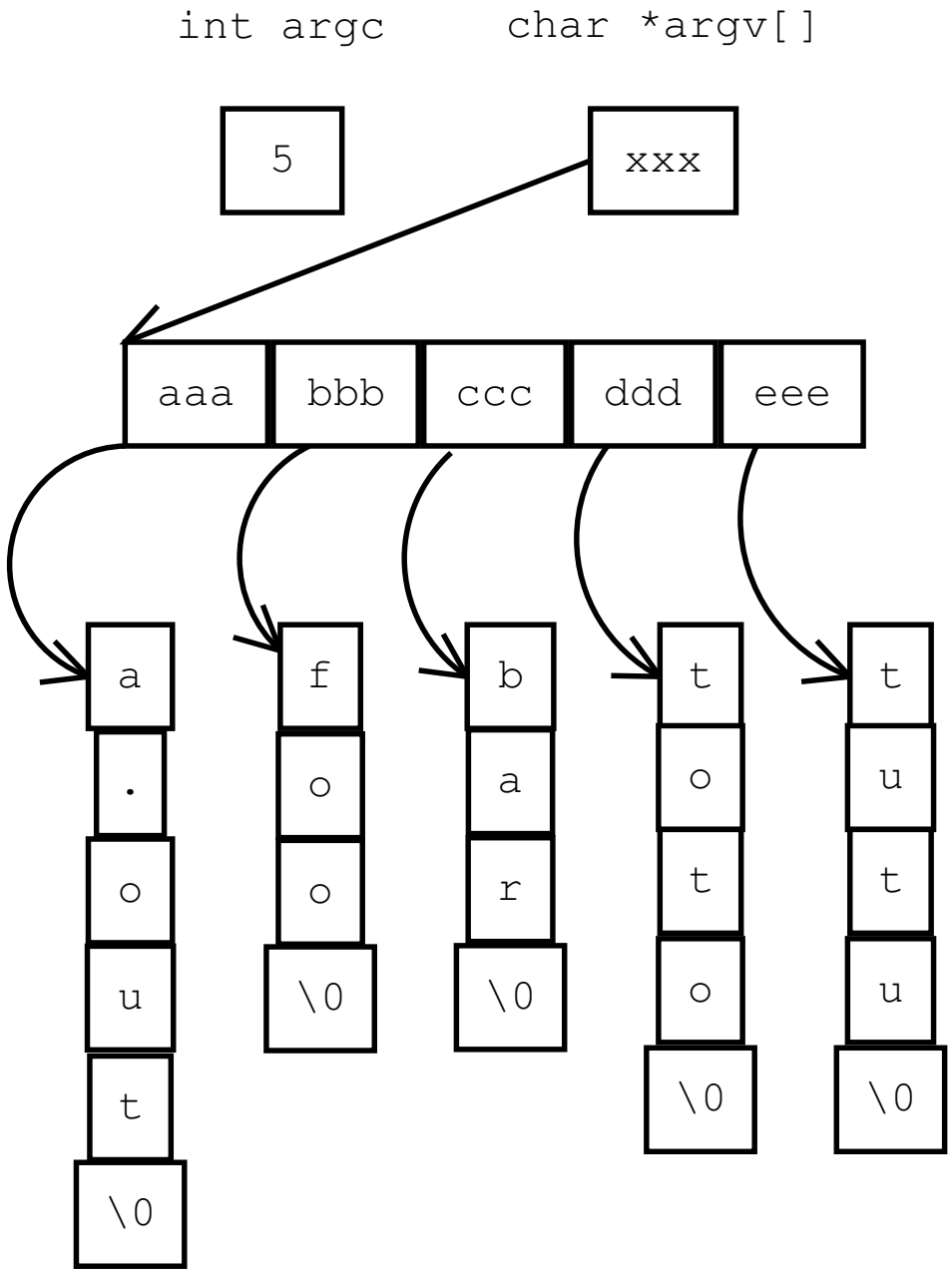
Les paramètres de la fonction `main` : exemple

```
# include <stdio.h>

int main(int argc, char **argv) {
  int i ;
  printf(" %d \n",argc) ;
  for( ; argc > 0 ; argc--){
    printf(" %d ",argc) ;
    i = 0 ;
    while(argv[argc-1][i]!=0)
      putchar(argv[argc-1][i++]) ;
    putchar('\n') ;
  }
  return 0;
}
```

```
%gcc mainPar.c
%a.out foo bar toto tutu
 5
 5 tutu
 4 toto
 3 bar
 2 foo
 1 ./a.out
```

Pourquoi écrire `char **argv` plutôt que `char argv[][]` ?



Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

Les variables d'environnement correspondent aux variables du Shell. Elles sont :

- stockées dans la zone de données statique ;
- qui est constituée d'une suite de chaînes : `<nom>=<value>` ;
- accessibles par la fonction `getenv` :

```
#include <stdlib.h>
char *getenv(const char *name)
```

recherche dans l'environnement une chaîne de la forme `name=value` et retourne un pointeur sur `value` si elle est présente.

Mais on peut aussi y accéder par les paramètres de la fonction main.

Forme générale des paramètres de la foncion main

Cette forme est :

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

Le dernier paramètre `arge` étant une suite — terminées par null — de chaînes de caractères du types : `varname=value`.

Le code suivant affiche l'ensemble des variables d'environnement dont il dispose :

```
# include <stdio.h>
int main(int argc, char **argv, char **arge) {
   while(*arge)
      printf("%s\n",*(arge++)) ;
  return 0;
}
```

On obtient entre autre :

```
PWD=/home/calforme/sedoglav/Enseignement/C/Cours/Sources
TERM=xterm
OSTYPE=linux
HOST=espoir.lifl.fr
```

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio

Lire et écrire des fichiers via `stdio`

- Notion de flot (*stream*)
- Ouvrir/fermer un fichier : `fopen, fclose`
- Lire/écrire en mode texte : `fgetc, fgets, fputc, fputs, fprintf, fscanf`
- Lire/écrire en mode binaire : `fread, fwrite`

Notion de flot (*stream*)

- Flot = suite d'octets pouvant représenter le contenu d'un fichier par exemple
- Exemples :
  - « Bonjour\nNous sommes le 12 février\nSalutations »
  - « 0F123A331EDDDD00F0A25354645AA »
- On peut lire/écrire un flot en mode texte ou binaire.
- Type flot pour fichiers : `FILE * mon_fichier ;`

Ouvrir un fichier

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

Modes :

`"r"` : lecture d'un fichier existant, flot positionné au début
`"r+"` : comme `"r"` avec écriture possible
`"w"` : création et écriture d'un fichier
`"w+"` : comme `"w"` avec lecture possible
`"a"` : écriture à la fin d'un fichier existant
`"a+"` : comme `"a"` avec lecture possible
`"rb", "rb+", "wb"...` : idem en mode binaire.

Exemple :

```
FILE *mon_fichier ;
mon_fichier = fopen("readme.txt","r") ;
if (!mon_fichier) exit(EXIT_FAILURE) ;/*si NULL*/
```

Fermer un fichier

```
int fclose(FILE *flot)
```

Renvoie 0 en cas de succès.

Exemple :

```
FILE *mon_fichier ;
mon_fichier = fopen("readme.txt","r") ;
if (!mon_fichier) exit(EXIT_FAILURE) ;/*si NULL*/
fclose(mon_fichier) ;
```

Lire en mode texte

`int fgetc(FILE *flot)` : lecture d'un caractère (`EOF` si fin)

`char *fgets(char *s, int n, FILE *flot)` :

- lecture d'une ligne (maximum n caractères),
- ajout de `'\0'`,
- stockage dans l'espace pointé par `s`,
- renvoie `s` ou `NULL` si erreur ou fin de fichier.

Exemple :

```
char *s = malloc(100*sizeof(char)) ;
FILE *mon_fichier ;
mon_fichier = fopen("readme.txt","r") ;
if (!mon_fichier) exit(EXIT_FAILURE) ;/*si NULL*/
while (!fgets(s,100,mon_fichier)) printf("%s",s);
fclose(mon_fichier) ;
```

Ecrire en mode texte

`int fputc(int c, FILE *flot)` : écriture d'un caractère (EOF si erreur)

`int fputs(const char *s, FILE *flot)` : écrit la chaîne sur le flot (EOF si erreur)

Ecriture formatée, lecture formatée

De façon similaire à `printf` et `scanf`,

- écriture formatée dans un flot (cf. format p. 92 du poly) :

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

- lecture formatée dans un flot (cf. format p. 94 du poly) :

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

De même, écritre/lecture formatée dans une chaîne de caractères,

```
int sprintf(const char *dest, const char * format,...)
int sscanf(const char *source, const char * format,...)
```

Unités standard d'entrée/sortie

`FILE * stdin` : entrée standard (clavier en général)
`FILE * stdout` : sortie standard (écran en général)
`FILE * stderr` : erreur standard (écran en général)

```
int fprintf(stdout, format,...)
```
équivaut à
```
int printf(format,...)
```

```
int fscanf(stdin, format,...)
```
équivaut à
```
int scanf(format,...)
```

Lecture/écriture en mode binaire

```
size_t fread(void *destination, size_t taille,
             size_t nombre, FILE *flot) :
```

- essaie de lire nombre objets de taille taille
- écrit dans l'espace pointé par destination
- renvoie le nombre d'objets réellement lus.

De même, pour l'écriture :

```
size_t fwrite(void *source, size_t taille,
              size_t nombre, FILE *flot)
```

Pour en savoir plus...

Exemples concrets,
Autres fonctions de lecture/écriture,
Fonction de déplacement de la position courante dans un flot,
Etc.

cf. chapitre 7 du poly

Aujourd'hui

Les pointeurs : suite & fin

* Allocation dynamique : `malloc`, `calloc`, `free`
* Les pointeurs de fonctions
* Les déclarations complexes
* Les paramètres de la fonction main [déjà vu]
* Les variables d'environnement
* Lire et écrire des fichiers via stdio