Programmation C

Entrées/sorties et fichiers

ESIFE Info 1^{ère} année

Université de Marne-la-Vallée

Retour de printf

- ▶ valeur de retour = nombre de caractères écrits sur la sortie standard

```
1 int main(int argc, char* argv[]){
2     int n=printf("command=%s\n", argv[0]);
3     int i;
4     for (i=1 ; i<n ; i++)
5         printf("-");
6     printf("\n");
7     return 0;
8 }
```

donne

```
nborie@perceval:~> ./test
command=./test
-----
```

Retour de printf

- ▶ retour négatif en cas d'erreur
- ▶ très rare : sortie redirigée vers un fichier et plus de place sur le disque
- ▶ le système gère les cas où :
 - le support amovible est absent
 - le programme n'a pas les droits pour écrire sur le fichier de redirection

```
nborie@perceval:> echo "FOO\nBAR\nBLA" > /dev/stdout  
FOO  
BAR  
BLA  
nborie@perceval:>
```

Affichage des entiers

- ▶ `%d` : entier signé
- ▶ `%u` : entier non signé
- ▶ `%o` : entier non signé en octal
- ▶ `%x %X` : entier non signé en hexadécimal

```
1 int main(int argc, char* argv[]){
2     int i=-45, j=43;
3     printf("%d\n", i);
4     printf("%u\n", i);
5     printf("%o\n", j);
6     printf("%x\n", j);
7     printf("%X\n", j);
8     return 0;
9 }
```

donne

```
nborie@perceval:~> ./test
```

```
-45
```

```
4294967251
```

```
53
```

```
2b
```

```
2B
```

Affichage des réels

- ▶ `%f` : standard, 6 décimales
- ▶ `%g` : standard, sans les zéros finaux
- ▶ `%e %E` : notation exponentielle

```
1 int main(int argc, char* argv[]){
2     double f=345.575;
3     printf("%f\n", f);
4     printf("%g\n", f);
5     printf("%e\n", f);
6     printf("%E\n", f);
7     return 0;
8 }
```

donne

```
nborie@perceval:~> ./test
345.575000
345.575
3.455750e+02
3.455750E+02
```

Formats spéciaux

- ▶ `%p` : affichage en hexadécimal des adresses mémoires
- ▶ `%n` : stocke le nombre de caractères déjà écrits

```
1 int main(int argc, char* argv[]){
2     int n;
3     double d=458.21;
4     printf("%g%n", d, &n); /* Une adresse a printf ??? */
5     printf("->%d_chars\n", n);
6     printf("&n=%p\n", &n);
7     return 0;
8 }
```

donne

```
nborie@perceval:~> ./test
458.21->6 chars
&n=0x7fffbcb88aa64
```

Le gabarit

- ▶ entier après le % = nombre minimum de caractères à utiliser
- ▶ `printf` complète avec des espaces si nécessaire, ou des zéros si `%0` avant un nombre

```
1 int main(int argc, char* argv[]){
2     printf ("%5d\n", 4);
3     printf ("%5d\n", 123456);
4     printf ("%5f\n", 3.1 f);
5     printf ("%5g\n", 3.1 f);
6     printf ("%5s\n", "abc");
7     printf ("%05d\n", 4);
8     return 0;
9 }
```

donne

```
nborie@perceval:~> ./test
```

```
  4
123456
3.100000
 3.1
 abc
00004
```

Précision

- ▶ `%.3f` : 3 décimales
- ▶ règles d'arrondi
- ▶ compatible avec un gabarit

```
1 int main(int argc, char* argv[]){  
2     double f=3.14159265;  
3     printf("%.3f\n", f);  
4     printf("%08.3f\n", f);  
5     printf("%.4e\n", f);  
6     return 0;  
7 }
```

donne

```
nborie@perceval:~> ./test  
3.142  
0003.142  
3.1416e+00
```

Gabarit et précision variables

- ▶ si on ne les connaît pas à la compilation :
 - * au lieu d'un entier + variable en argument de `printf`

```
1 int main(int argc, char* argv[]){
2     int i, tmp;
3     int max=0;
4     for (i=0 ; i<argc ; i++){
5         tmp=strlen(argv[i]);
6         if (tmp>max) max=tmp;
7     }
8     for (i=0 ; i<argc ; i++)
9         printf("arg #%d=%s\n", i, max,argv[i]);
10    return 0;
11 }
```

donne

```
nborie@perceval:~> ./test this lightsaber padawan!
arg #0=    ./test
arg #1=    this
arg #2=lightsaber
arg #3=    padawan!
```

signes des nombres

- ▶ `%+d` : force l'affichage du signe
- ▶ `% d` : met un espace si le signe est +

```
1 int main(int argc, char* argv[]){
2     float f=12.54f;
3     float g=-12.54f;
4     printf("%+f\n", f);
5     printf("%+f\n", g);
6     printf("%_f\n", f);
7     printf("%_f\n", g);
8     printf("%+.3f\n", f);
9     return 0;
10 }
```

donne

```
nborie@perceval:~> ./test
+12.540000
-12.540000
 12.540000
-12.540000
+12.540
```

sprintf

- ▶ le résultat est mis dans une chaîne de caractères supposée assez longue
- ▶ attention aux débordements

```
1 int main(int argc, char* argv[]){
2     char* firstname="Hannibal";
3     char* lastname="Smith";
4     char email[256];
5     sprintf(email, "%s.%s@a-team.org", firstname, lastname);
6     printf("mail=%s\n", email);
7     return 0;
8 }
```

donne

```
nborie@perceval:~> ./test
mail=Hannibal.Smith@a-team.org
```

scanf

- ▶ `%*...` : saisir sans sauver le résultat
- ▶ permet de "manger" des choses sur l'entrée standard

```
1  /* Examine l'entree standard jusqu'a trouver un entier  
2  positif et le retourne. Retourne -1 s'il n'y a plus  
3  rien a lire. */  
4  int scan_positive_integer(void)  
5  {  
6      int res, n=-1;  
7      do{  
8          res=scanf("%d",&n);  
9          if (res==-1) return -1;  
10         if (res==0) scanf("%*c");  
11     } while (n<0);  
12     return n;  
13 }
```

sscanf

- ▶ les données sont lues depuis une chaîne

```
1 int main(int argc, char* argv[]){
2     char* date="Tue Jul 17 14:50:00 CEST 2007";
3     char day[4], month[4];
4     int nday, year;
5     sscanf(date, "%s %s %d %*s %*s %d",
6             day, month, &nday, &year);
7     printf("%s %d %s %d\n", day, nday, month, year);
8     return 0;
9 }
```

donne

```
nborie@perceval:~> ./test
```

```
Tue 17 Jul 2007
```

getchar

- ▶ `int getchar(void);`
- ▶ équivalent à `scanf("%c", &c);`
- ▶ renvoie le caractère lu ou `EOF` en cas d'erreur
- ▶ plus lisible
- ▶ plus rapide, car pas de format à analyser

Qu'est-ce qu'un fichier ?

- ▶ espace de stockage des données
- ▶ à peu près tout est fichier sous Unix
- ▶ opérations permises:
 - ouvrir, lire, écrire, fermer
- ▶ il n'est pas prévu de mécanisme pour enlever un morceau au milieu d'un fichier
- ▶ tous les fichiers ne sont pas en *random access*
(ex: `/dev/mouse`)

Les primitives

- ▶ primitives systèmes :

```
int open(const char *pathname, int flags, mode_t mode);
```

```
int close(int fd);
```

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

- ▶ peu pratique car ne manipule que des octets
- ▶ exercice horrible : écrire une fonction `void affiche_reel(const void* r);` qui a le comportement de `printf("%f", r);` mais qui n'utilise que des primitives systèmes.

Un petit exemple de primitive d'entrée/sortie

Mon ordinateur personnel me donne :

```
nborie@perceval:~/ > ls -al /dev/stdout
lrwxrwxrwx 1 root root 15 oct. 29 10:34 /dev/stdout -> /proc/self/fd/1
```

La sortie standard appartient à root mais semble autoriser l'écriture à tous ! Son numéro de file descriptor (fd) est le 1

```
1 #include <unistd.h>
2
3 /* Ouais on ecrit a l'ecran sans <stdio.h>,
4    meme pas peur ! */
5 int main(int argc, char* argv[]){
6     char* s="Salut!\n";
7     write(1, (const void*)s, 7);
8     return 0;
9 }
```

donne bien

```
nborie@perceval:~> ./test
Salut!
```

E/S civilisées

- ▶ `FILE` = structure décrivant un fichier (ouf !)
- ▶ varie selon les implémentations
⇒ ne jamais utiliser les champs directement
- ▶ On exploite les variables de cette structure avec les fonctions de `stdio.h`

```
FILE *fopen(const char *path, const char *mode);  
int fclose(FILE *fp);  
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);  
size_t fwrite(const void *ptr, size_t size, size_t nmemb,  
FILE *stream);  
int fscanf(FILE *stream, const char *format, ...);  
int fprintf(FILE *stream, const char *format, ...);
```

Ouvrir un fichier

- ▶ options de `fopen` :
 - "`r`" : lecture seule
 - "`w`" : écriture seule (fichier créé si nécessaire, écrasé si existant)
 - "`a`" : ajout en fin (fichier créé si nécessaire)
- ▶ par défaut, opération en mode texte (et pas binaire)
 - différences pour les retours à la ligne suivant les systèmes
- ▶ **toujours** tester la valeur de retour !

Ouvrir un fichier

- ▶ opérations en mode binaire :

"rb" "wb" "ab"

- ▶ opérations en mode lecture et écriture :

"r+" "w+" "a+"

- ▶ combinaisons possibles :

"rb+" "r+b" "wb+" "w+b" "ab+" "a+b"

Ouvrir un fichier

- ▶ si le nom du fichier est relatif, il l'est par rapport au répertoire courant
- ▶ en cas de création, les droits dépendent du `umask` (`rw-rw-r--` par défaut sous Ubuntu)

```
1 int main(int argc, char* argv[]){
2     FILE* foo=fopen("foo", "w");
3     if (foo==NULL) exit(1);
4     fprintf(foo, "Hello _you\n");
5     fclose(foo);
6     return 0;
7 }
```

donne

```
nborie@perceval:~> ./test
nborie@perceval:~> ls -al foo
-rw-rw-r-- 1 nborie nborie 10 oct. 29 14:04 foo
```

Fermer un fichier

- ▶ `fclose(f);`
- ▶ `f` doit être non `NULL`
- ▶ `f` doit être l'adresse d'un `FILE` valide
- ▶ `f` ne doit pas avoir déjà été fermé
- ▶ tout fichier ouvert doit être fermé

E/S binaires

```
size_t fread(void *ptr,  
             size_t size,  
             size_t nmemb,  
             FILE *stream);
```

- ▶ lit `nmemb` éléments de taille `size` chacun
- ▶ les met dans la zone d'adresse `ptr`
- ▶ retourne le nombre d'éléments de taille `size` lus ≠ nombre d'octets lus
- ▶ `sizeof(size_t)` est machine dépendant (c'est souvent juste un alias de `unsigned int` sous système 32 bits et de `long unsigned int` sous 64 bits, quoi qu'il en soit, on adapte les `printf` en conséquence)

E/S binaires

- ▶ `fread` retourne 0 en cas d'erreur ou de fin de fichier
- ▶ peut renvoyer moins que le `n` demandé
- ▶ si on veut un nombre précis, on doit faire une boucle

```
1 int read_n_ints(int* t, unsigned int n, FILE* f){
2     int i;
3     while (n && (i=fread(t,n,sizeof(int),f))){
4         t=t+i;
5         n=n-i;
6     }
7     return (n==0)
8 }
```

E/S binaires

```
size_t fwrite(const void *ptr,  
              size_t size,  
              size_t nmemb,  
              FILE *stream);
```

- ▶ lit depuis la zone d'adresse `ptr`
- ▶ écrit `nmemb` éléments de taille `size` chacun
- ▶ retourne le nombre d'éléments de taille `size` écrits ≠ nombre d'octets écrits

E/S binaires

- ▶ `fread` et `fwrite` retournent `nmemb` en cas de réussite
- ▶ valeur de retour de `fread` et `fwrite`:
 - peut être `<nmemb` voir 0 en cas d'erreur
 - ou si la fin de fichier est atteinte en lecture
- ▶ la fin de fichier en lecture est traitée comme une erreur

E/S formatées

- ▶ `fprintf` et `fscanf` fonctionnent comme `printf` et `scanf`
- ▶ fonctions bufferisées
- ▶ en fait, `printf(...)`; et `scanf(...)`; sont équivalents à `fprintf(stdout,...)`; et `fscanf(stdin,...)`;
- ▶ `stdin`, `stdout` et `stderr` sont des `FILE*` spéciaux

fgetc/fputc

- ▶ `int fgetc(FILE* stream);`
- ▶ `int fputc(int c, FILE* stream);`
- ▶ servent à lire ou écrire un seul caractère
- ▶ retourne `EOF` en cas d'erreur :
 - pas de caractère de fin de fichier
 - `EOF` n'est pas un caractère

ungetc

- ▶ `int ungetc(int c, FILE *stream);`
- ▶ permet d'annuler la lecture du dernier caractère lu
- ▶ `c` remplace le dernier caractère lu, mais seulement en mémoire, pas dans le fichier
- ▶ à éviter, donc concevoir des formats de fichiers qui évitent les aller-retours

fgets

- ▶ `char *fgets(char *s, int size, FILE *stream);`
- ▶ `fgets` lit des caractères jusqu'à :
 - un `\n` (qui est copié dans le résultat `s`)
 - la fin de fichier
 - avoir lu `n-1` caractères
- ▶ retourne `NULL` en cas erreur, `s` sinon
- ▶ évite les problèmes de débordement

fputs

- ▶ `int fputs(const char *s, FILE *stream);`
- ▶ `fputs` écrit la chaîne `s` dans le fichier `f`
- ▶ retourne `EOF` en cas d'erreur, `0` sinon

Fin de fichier en lecture

Attention à toujours vérifier les descriptions de valeur de retour dans le manuel [man](#) !

- ▶ `fscanf` : retourne `EOF`
- ▶ `fgetc` : retourne `EOF`
- ▶ `fgets` : retourne `NULL`
- ▶ `fread` : retourne `0`

```
1 void copy(FILE* src, FILE* dest){
2     char buffer[4096];
3     size_t n;
4     while ((n=fread(buffer, sizeof(char), 4096, src)) > 0){
5         fwrite(buffer, sizeof(char), n, dest);
6     }
7 }
```

Fin de fichier en lecture

- ▶ `int feof(FILE *stream);`
- ▶ renvoie `0` si la fin de fichier est atteinte
- ▶ pas pratique et peu judicieux car on doit forcément avoir fait échouer une lecture pour que ça marche

```
1 int main(int argc, char* argv[]){
2     char s[4096];
3     FILE* f=fopen("foo", "r");
4     if (f==NULL) exit (1);
5     while (!feof(f)){
6         fscanf(f, "%s", s);
7         printf("** %s\n", s);
8     }
9     fclose(f);
10    return 0;
11 }
```

donne

```
nborie@perceval:~> cat foo
abc toto
23
nborie@perceval:~> ./test
** abc **
** toto **
** 23 **
** 23 **
```

fichiers textes ou binaires ?

- ▶ différences d'efficacité
- ▶ au programmeur de choisir
- ▶ on peut mélanger

```
1  int main(int argc, char* argv[]){
2      int i=708077092;
3      FILE* f=fopen("text", "w");
4      if (f==NULL) exit(1);
5      fprintf(f, "%d\n", i);
6      fclose(f);
7      f=fopen("bin", "wb");
8      if (f==NULL) exit(1);
9      fwrite(&i, sizeof(int), 1, f);
10     fclose(f);
11     return 0;
12 }
```

donne

```
nborie@perceval:~> ./test
nborie@perceval:~> cat text
708077092
nborie@perceval:~> cat bin
$f4*~>
```

Sérialisation

- ▶ si un "objet" ne contient pas de pointeur, `fwrite` permet de le sauver en une seule fois
- ▶ idem en lecture

```
1 typedef struct{
2     int score;
3     int board[N][N];
4 } Game;
5
6 int save_game(char* n, Game g){
7     FILE* f=open(n, "wb");
8     if (f==NULL) return 0;
9     fwrite(g, sizeof(Game), 1, f);
10    fclose(f);
11    return 1;
12 }
```

Endianness

- ▶ les résultats de `fread` et `fwrite` dépendent de l'endianness
- ▶ fichiers potentiellement non portables !
- ▶ sérialisation à éviter si besoin de portabilité

Encodage de texte

- ▶ pour des fichiers textes, on doit gérer les E/S à la main si on veut des codages sur plus d'un octet :
 - UTF8
 - UTF16 (Little ou Big Endian)
 - ...

Positionnement

- ▶ `int fseek(FILE *stream, long offset, int whence);`
- ▶ modifie la position courante dans le fichier
- ▶ `position = offset + whence`
- ▶ `whence` peut valoir :
 - `SEEK_SET` : début du fichier
 - `SEEK_CUR` : position courante
 - `SEEK_END` : fin du fichier

Positionnement

- ▶ `void rewind(FILE* stream);`
- ▶ `long ftell(FILE* stream);`
- ▶ `rewind` revient au début du fichier
- ▶ `ftell` donne la position courante

```
1 long filesize(FILE* f){
2     long old_pos=ftell(f);
3     fseek(f, 0, SEEK_END);
4     long size=1+ftell(f);
5     fseek(f, old_pos, SEEK_SET);
6     return size;
7 }
```

Positionnement

- ▶ sur une machine xxxbits, on ne peut pas adresser un fichier plus grand que 2^{xxx} avec `fseek` et `ftell`
- ▶ équivalents plus portables :

```
int fgetpos(FILE *stream, fpos_t *pos);  
int fsetpos(FILE *stream, fpos_t *pos);
```

Renommer et détruire

- ▶ `int rename(const char *oldpath, const char *newpath);`
- ▶ `int remove(const char *pathname);`
- ▶ ces fonctions (`stdio.h`) fonctionnent aussi pour les répertoires
- ▶ elles renvoient 0 si OK, -1 si erreur

Créer un répertoire

- ▶ `int mkdir(const char *pathname, mode_t mode);`
- ▶ `mode` spécifie les droits du repertoire
- ▶ ces droits seront combinés avec le `umask`
- ▶ fonction non portable!
- ▶ nécessite les bibliothèques `sys/types.h` et `sys/stat.h`

Noms de fichiers

- ▶ un nom de fichier peut être relatif ou absolu
- ▶ extension facultative
- ▶ possibilité d'avoir plusieurs .
`projet_c_L3.tar.gz`

Noms de fichiers

- ▶ attention aux pièges !
 - `/home/foo/t.info/zz` n'a pas `info/zz` comme extension
- ▶ attention aux fichiers cachés
 - `.bashrc` n'a pas d'extension
- ▶ le séparateur varie selon les systèmes
 - `/` sous Linux
 - `\` sous Windows
 - `:` sous les vieux Mac

Fichiers temporaires

- ▶ comment être sûr qu'on ouvre un fichier qui n'existe pas déjà ?
- ▶ pour obtenir un nom de fichier unique :
 - le faire à la main (pénible, fastidieux...)
 - utiliser une fonction qui sait le faire comme
`char *tempnam(const char *dir, const char *pfx);`

Problèmes de tempnam

- ▶ renvoie un pointeur sur une chaîne globale
 - pas thread-safe
- ▶ le fichier peut avoir été créé entre l'obtention du nom et l'ouverture
- ▶ ne pas être parano :
 - problème SSI énormément de fichiers temporaires à gérer (application web, ...)

Version sécurisée

- ▶ `int mkstemp(char* template);`
- ▶ `template` doit être une chaîne modifiable terminée par `XXXXXX` (6 grand X)
- ▶ génère un nom de fichier unique
- ▶ l'ouvre aussitôt avec les droits `rw` avec accès exclusif pour l'utilisateur courant
- ▶ renvoie -1 en cas d'erreur, le descripteur de fichier sinon

Version sécurisée

- ▶ on passe d'un descripteur à un `FILE*` avec
`FILE *fopen(const char *path, const char *mode);`
- ▶ pas du C ANSI malheureusement...

```
1  int main(int argc, char* argv[]){
2      char t[32];
3      strcpy(t, "fooXXXXXX");
4      int fd=mkstemp(t);
5      if (fd== -1) exit(1);
6      FILE* f=fdopen(fd, "w");
7      if (f==NULL) exit(1);
8      fprintf(f, "Hello _you!\n");
9      fclose(f);
10     return 0;
11 }
```

```
nborie@perceval:~> gcc -o test test.c -Wall -ansi
```

```
test.c: In function 'main':
```

```
4: att : impl. decl. of function 'mkstemp' [-Wimplicit-function-declaration]
```

```
6: att : impl. decl. of function 'fdopen' [-Wimplicit-function-declaration]
```

```
6: att : pointer from integer without a cast [enabled by default]
```

Bufferisation des E/S

- ▶ 3 modes :
 - bufferisation totale
 - bufferisation par ligne
 - pas de bufferisation

Un léger parallèle :

- ▶ Vous acheminez régulièrement du poisson du Havre à Paris (On affiche souvent des choses à l'écran)...
- ▶ Le poisson a besoin d'être frais (On veut lire les choses rapidement à l'écran)...
- ▶ Il faut payer les chauffeurs (Solliciter l'affichage demande de la CPU (et d'autres choses coûteuses))...
- ▶ On peut stocker au Havre avant de faire partir un gros camion (On peut remplir un large buffer avant de solliciter l'affichage)...

L'enjeu :

Comment économiser de l'argent et livrer du poisson frais ?

Comment minimiser le coût en CPU sans que les messages attendent trop ?

Bufferisation des E/S

- ▶ bufferisation totale :

```
void setbuf(FILE *stream, char *buf);
```

```
void setbuffer(FILE *stream, char *buf, size_t size);
```

- ▶ avec `setbuf`, la taille du buffer est fixée à `BUFSIZ` (8192 octets par défaut pour une Ubuntu 64)

Bufferisation des E/S

- ▶ bufferisation par lignes :

```
void setlinebuf(FILE *stream);
```

- ▶ équivalent à :

```
setvbuf(stream, (char *)NULL, _IOLBF, 0);
```

- ▶ pour les valeurs de retour, voir `setvbuf`

Bufferisation des E/S

- ▶ bufferisation générale :
`int setvbuf(FILE *stream, char *buf, int mode, size_t size);`
- ▶ `mode` :
 - `_IONBF` : non bufferisé
 - `_IOLBF` : bufferisé par lignes
 - `_IOFBF` : bufferisé complètement (F=full)
- ▶ `size` : taille du buffer
- ▶ renvoie 0 si OK, `EOF` sinon