Cours de C

Entrées/sorties
Fichiers

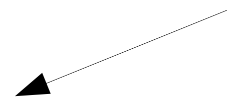
Sébastien Paumier

Utilisation avancée de printf

Retour de printf

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

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



```
$>./test_printf
command=./test_printf
---------------------
```

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

Affichage des entiers

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

```c
int main(int argc,char* argv[]) {
   int i=-45,j=43;
   printf("%d\n",i);
   printf("%u\n",i);
   printf("%o\n",j);
   printf("%x\n",j);
   printf("%X\n",j);
   return 0;
}
```

→

```
$>./a.out
-45
4294967251
53
2b
2B
```

Affichage des réels

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

```
int main(int argc,char* argv[]) {
   double f=345.575;
   printf("%f\n",f);
   printf("%g\n",f);
   printf("%e\n",f);
   printf("%E\n",f);
   return 0;
}
```

```
$>./a.out
345.575000
345.575
3.455750e+002
3.455750E+002
```

Formats spéciaux

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

```c
int main(int argc,char* argv[]) {
   int n;
   double d=458.21;
   printf("%g%n",d,&n);
   printf("=%d chars\n",n);
   printf("&n=%p\n",&n);
   return 0;
}
```

→

```
$>./a.out
458.21=6 chars
&n=0022FF6C
```

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

```c
int main(int argc,char* argv[]) {
   printf("%5d\n",4);
   printf("%5d\n",123456);
   printf("%5f\n",3.1f);
   printf("%5s\n","abc");
   printf("%05d\n",4);
   return 0;
}
```

```
$>./a.out
    4
123456
3.100000
  abc
00004
```

gabarit ≠ précision

Précision

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

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

→

```
$>./a.out
3.142
0003.142
3.1416e+000
```

Gabarit et précision variables

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

```
int main(int argc,char* argv[]) {
int i,tmp;
int max=0;
for (i=0;i<argc;i++) {
   tmp=strlen(argv[i]);
   if (tmp>max) max=tmp;
}
for (i=0;i<argc;i++) {
   printf("arg #%d=%*s\n",i,max,argv[i]);
}
return 0;
}
```

```
$>./t is the command
arg #0=     ./t
arg #1=      is
arg #2=     the
arg #3=command
```

Signes des nombres

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

```
int main(int argc,char* argv[]) {
   float f=12.54f;
   float g=-12.54f;
   printf("%+f\n",f);
   printf("%+f\n",g);
   printf("% f\n",f);
   printf("% f\n",g);
   printf("%+.3f\n",f);
   return 0;
}
```

→

```
$>./a.out
+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

```
int main(int argc,char* argv[]) {
   char* firstname="John";
   char* lastname="Doe";
   char email[256];
   sprintf(email,"%s.%s@univ-mlv.fr",
                 firstname,lastname);
   printf("email=%s\n",email);
   return 0;
}
```

```
$>./a.out
email=John.Doe@univ-mlv.fr
```

scanf

- `%*...` : saisir sans sauver le résultat
- permet de "manger" des choses

```
int scan_positive_integer() {
    int res,n=-1;
    char foo;
    do {
        res=scanf("%d",&n);
        if (res==-1) return -1;
        if (res==0) {
            scanf("%c",&foo);
        }
    } while (n<0);
    return n;
}
```

```
int scan_positive_integer() {
    int res,n=-1;
    do {
        res=scanf("%d",&n);
        if (res==-1) return -1;
        if (res==0) {
            scanf("%*c");
        }
    } while (n<0);
    return n;
}
```

sscanf

- les données sont lues depuis une chaîne

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

```
$>./a.out
Tue 17 Jul 2007
```

getchar

- `int getchar();`
- 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

Les fichiers

Qu'est-ce qu'un fichier ?

- espace de stockage de données
- opérations permises:
  - lire, écrire, tronquer
- 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 pratiques car ne manipulent que des octets
- `printf("%f",r);` ⇔ truc très méchant si demandé à l'examen

E/S civilisées

- **FILE** = structure décrivant un fichier
- varie selon les implémentations

  ⇒ ne jamais utiliser ses champs directement
- fonctions de **stdio.h**:

```
FILE* fopen(const char* path,const char* mode);
int fclose(FILE* stream);
size_t fread(void* ptr,size_t size,size_t n,FILE* stream);
size_t fwrite(const void* ptr,size_t size,size_t n,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, opérations en mode texte
  - différences pour les retours à la ligne suivants 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-r--r--** par défaut)

```
int main(int argc,char* argv[]) {
   FILE* foo=fopen("foo","w");
   if (foo==NULL) exit(1);
   fprintf(foo,"hello\n");
   fclose(foo);
   return 0;
}
```

```
$>../bin/a.out
$>ls -l foo
-rw-r--r--  1 paumier igm 6 jui 17 15:38 foo
```

Fermer un fichier

- `fclose(f);`
- `f` doit être non `NULL`
- `f` doit représenter une adresse 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 n,
             FILE *f);
```

- lit **`n`** éléments de taille **`size`** chacun
- les met dans la zone d'adresse **`ptr`**
- retourne le nombre d'éléments lus

≠ nombre d'octets lus ⚡

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 ↯

```
int read_n_ints(int* t,unsigned int n,FILE* f) {
   int i;
   while (n && (i=fread(t,n,sizeof(int),f))) {
      t=t+i;
      n=n-i;
   }
   return (n==0);
}
```

E/S binaires

```
size_t fwrite(const void* ptr,
              size_t size,
              size_t n,
              FILE* f);
```

- écrit `n` éléments de taille `size` chacun
- les lit dans la zone d'adresse `ptr`
- retourne le nombre d'éléments écrits

≠ nombre d'octets écrits

E/S binaires

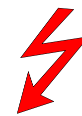
- valeur de retour de `fread` et `fwrite`:
  - peut être <n voire 0 en cas d'erreur
  - ou si la fin de fichier est atteinte en lecture
- la fin de fichier en lecture est traitée comme une erreur

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 allers-retours

fgets

- `char* fgets(char* s,int n,FILE* f);`
- `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 d'erreur, `s` sinon
- évite les problèmes de débordement

fputs

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

Fin de fichier en lecture

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

```
void copy(FILE* src,FILE* dst) {
   char buffer[4096];
   size_t n;
   while ((n=fread(buffer,sizeof(char),4096,src))>0) {
      fwrite(buffer,sizeof(char),n,dst);
   }
}
```

Fin de fichier en lecture

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

```
int main(int argc,char* argv[]) {
   char s[4096];
   FILE* f=fopen("foo","r");
   if (f==NULL) exit(1);
   while (!feof(f)) {
      fscanf(f,"%s",s);
      printf("** %s **\n",s);
   }
   fclose(f);
   return 0;
}
```

→

```
$>cat foo
abc toto
23
$>./a.out
** abc **
** toto **
** 23 **
** 23 **
```

Fichiers textes ou binaires?

- différence d'efficacité
- au programmeur de choisir
- on peut mélanger

```
int main(int argc,char* argv[]) {
   int i=708077092;
   FILE* f=fopen("text","w");
   if (f==NULL) exit(1);
   fprintf(f,"%d\n",i);
   fclose(f);
   f=fopen("bin","wb");
   if (f==NULL) exit(1);
   fwrite(&i,sizeof(int),1,f);
   fclose(f);
   return 0;
}
```

```
$>./a.out
$>cat text
708077092
$>cat bin
$f4*$>
```

Sérialisation

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

```
typedef struct {
   int score;
   int board[N][N];
} Game;

int save(char* n,Game* g) {
   FILE* f=fopen(n,"wb");
   if (f==NULL) return 0;
   fwrite(g,sizeof(Game),1,f);
   fclose(f);
   return 1;
}
```

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 texte, 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* f,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

```
long filesize(FILE* f) {
   long old_pos=ftell(f);
   fseek(f,0,SEEK_END);
   long size=1+ftell(f);
   fseek(f,old_pos,SEEK_SET);
   return size;
}
```

→ +1 car les positions vont de 0 à taille-1

Positionnement

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

```
int fsetpos(FILE* f,const fpos_t* pos);

int fgetpos(FILE* f,fpos_t *pos);
```

Renommer et détruire

- `int rename(const char* oldpath,const char* newpath);`
- `int remove(const char* pathname);`
- ces fonctions (`stdio.h`) fonctionnement aussi sur 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 répertoire
- ces droits seront combinés avec le umask
- fonction non portable!
- nécessite `sys/types.h` `sys/stat.h`

Noms de fichiers

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

  `projet-C-IR1.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 suivant 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 :(
  - utiliser une fonction qui le fait comme

```
char* tempnam(const char* dir,const char* pfx);
```

Problèmes de tempnam

- renvoie un pointeur sur une chaîne globale
  - pas thread-safe
- le fichier est peut-être 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`
- 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* fdopen(int fd,const char* mode);**

```
$>gcc -Wall -ansi tmp.c
tmp.c: Dans la fonction « main »:
tmp.c:8: AVERTISSEMENT: déclaration implicite de la fonction « mkstemp »
tmp.c:9: AVERTISSEMENT: déclaration implicite de la fonction « fdopen »
tmp.c:9: AVERTISSEMENT: initialisation transforme en pointeur un entier
sans transtypage
$>./a.out
$>./a.out
$>ls foo*
foo25DNOK  foojQ0vTo
```

```c
int main(int argc,char* argv[]) {
   char t[32];
   strcpy(t,"fooXXXXXX");
   int fd=mkstemp(t);
   if (fd==-1) exit(1);
   FILE* f=fdopen(fd,"w");
   if (f==NULL) exit(1);
   fprintf(f,"Hello!\n");
   fclose(f);
   return 0;
}
```

pas du C ANSI

Bufferisation des E/S

- 3 modes:
  - bufferisation totale
  - bufferisation par lignes
  - pas de bufferisation

Bufferisation des E/S

- bufferisation totale:

```
void setbuf(FILE* f,char* buf);
void setbuffer(FILE* f,char* buf,int size);
```

- avec **setbuf**, la taille est fixée à **BUFSIZ**

Bufferisation des E/S

- bufferisation par lignes:

```
int setlinebuf(FILE* f);
```

- équivalent à:

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

- valeur de retour: voir `setvbuf`

Bufferisation des E/S

- bufferisation générale:

```
int setvbuf(FILE* f,char* buf,int mode,
                size_t size);
```

- **mode**:
  - **_IONBF** : non bufferisé
  - **_IOLBF** : bufferisé par lignes
  - **_IOFBF** : bufferisé
- **size**: taille du buffer
- renvoie 0 si OK, **EOF** sinon