Programmation « orientée système »
LANGAGE C – FICHIERS

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

Présenter les bases des entrées/sorties en C :

- les entrées/sorties clavier/écran et formatage des entrées/sorties
- les fichiers

Jean-Cédric Chappelier

Interagir avec le monde : les entrées/sorties

Les interactions d'un programme avec « l'extérieur » sont gérées par des **instructions d'entrée/sortie** et ce qu'on appelle des « **flots** ».

Un flot correspond à un canal d'échange de données entre le programme et l'extérieur.

En C, les « **flots** » d'entrée et sortie standard sont représentés respectivement par `stdin` et `stdout` (définis dans `stdio.h`).

Jean-Cédric Chappelier

Affichage à l'écran

```
int printf("FORMAT", expr1, expr2, ...);
```

affiche à l'écran les valeurs des expressions *expr1*, *expr2*, ..., **insérées** dans le texte défini par *FORMAT* (un « langage dans le langage »).

`printf` **retourne** le nombre de caractères écrits ou une valeur négative en cas d'échec.

Exemple d'affichage :

```
int a = 1;
printf("intervalle : [%d, %d]\n", -a, a);
```

➪ `intervalle : [-1, 1]`

Jean-Cédric Chappelier

Formats d'affichage à l'écran

(Avant tout : `man 3 printf`)

- tous les caractères ordinaires (sauf '`%`') sont recopiés tels quels
- '`%`' introduit une conversion de valeur spécifiée par **1 caractère**
- entre le '`%`' et le caractère de spécification de conversion peuvent apparaître
  - '`-`' : ajustement à gauche dans le champ
  - '`+`' : toujours afficher le signe (nombres)
  - ' ' : met un espace si le premier caractère n'est pas un signe (en clair : pour les nombre positifs, et sans l'option `+`)
  - '`#`' : indicateur explicite de format : ajoute un `0` devant les nombres en octal, un `0x` ou `0X` devant les nombres en hexadécimal, un point systématique pour les `double`.
  - '`0`' : compléter le champ par des 0 non significatifs
  - des nombres : taille minimale du champ
  - '`*`' : taille du champ donné par une variable supplémentaire
  - '`.`' suivit d'un nombre ou de '`*`' : la « précision »
  - un indicateur de taille de l'objet : `h` pour `short`, `l` pour `long` et `L` pour `long double`.

Jean-Cédric Chappelier

Formats d'affichage à l'écran (2)

Principaux caractères de conversion :

	type requis	conversion
`d` ou `i`	`int`	affichage en décimal (signé)
`u`	`int`	affichage en décimal non signé
`x` ou `X`	`int`	affichage en hexadécimal (non signé)
`o`	`int`	affichage en octal (non signé)
`c`	`(char) int`	le caractère correspondant
`s`	`char*`	une chaîne de caractères.
`f`	`double`	affichage en décimal signé (123.456)
`e` ou `E`	`double`	affichage en décimal signé avec puissance de 10 (p.ex. `1.234e2`)
`g` ou `G`	`double`	choisit entre `%f` et `%e` en fonction de la précision
`p`	`void*`	adresse
`%`	–	affiche simplement un `%`

Exemples : `%d`, `%3.2f%%`, `%-+#5.4f`, `%-5.4s`

`%5.4s` n'affiche que les 4 premiers caractères de l'argument, placés à droite dans un champ de 5 caractères

```
double x = 10.4276;  double y = 123.456789;  double z = 4.0;
char nom[] = "ABCDEFGH";

printf(">%5.2f%%<\n",    x);        >10.43%<
printf(">%7.2f%%<\n",    x);        >  10.43%<
printf(">%3.2f%%<\n\n", x);         >10.43%<   (le « .2 » est prioritaire sur le « 3 »)

printf("XX%5.4sXX\n",     nom);     XX ABCDXX  (4 caractères au maximum affichés sur 5 « places »)
printf("XX%-5.4sXX\n\n", nom);      XXABCD XX

printf("XX%.4fXX\n",       y);      XX123.4568XX
printf("XX%12.4fXX\n",     y);      XX    123.4568XX
printf("XX%+12.4fXX\n",    y);      XX   +123.4568XX
printf("XX%+012.4fXX\n",   y);      XX+000123.4568XX
printf("XX%012.4fXX\n",    y);      XX0000123.4568XX
printf("XX%-+12.4fXX\n",   y);      XX+123.4568   XX
printf("XX%-12.4fXX\n\n", y);       XX123.4568    XX

printf("%.2f\n",  z);               4.00
printf("%.2g\n",  z);               4
printf("%#.2g\n", z);               4.0
```


Jean-Cédric Chappelier

Buffering

`printf` n'affiche pas toujours quelque chose !

```c
double x = 0.0;
printf("Un message peut ne pas être affiché");
double y = 3.14 / x; /* division par 0 */
```

En fait `printf` envoie ses messages dans un **tampon** (buffer en anglais)...

...que le système affiche « quand il a le temps ».

☞ **NE PAS** se fier aux messages affichés pour chercher où son programme plante ! (debuging)

OU ALORS forcer l'affichage (pour les messages « importants »)

Jean-Cédric Chappelier

Buffering (2)

« forcer l'affichage », ou plus exactement « vider le tampon »
se fait avec l'instruction `fflush`.

Exemple : `fflush(stdout);`

Pour en savoir plus : `man 3 fflush`

NOTE : `fflush(stdin);` n'a **aucun sens**
(ça compile, mais n'a aucun effet sur `stdin`)

Jean-Cédric Chappelier

Lecture au clavier

`int scanf("FORMAT", pointeur1, pointeur2, ...);`

permet à l'utilisateur de saisir au clavier une liste de valeurs *val1*, *val2*, ..., qui seront stockées dans la mémoire pointée par *pointeur1*, *pointeur2*, ...

Bien souvent ces pointeurs sont des **adresses** de variables : `&var1`, `&var2`, ...

Remarque : Lorsque plusieurs valeurs sont lues à la suite, le **caractère séparateur** de ces valeurs est le **un blanc** (au sens large : espace, tabulation, retour à la ligne, ... ; `man 3 isspace`)

☞ pour lire en une fois une ligne entière contenant des blancs (par exemple saisie d'un nom composé), il vaut mieux utiliser `fgets` (qui permet en plus de contrôler la taille)

```
char phrase_a_lire[TAILLE_ALLOUEE];
...
fgets(phrase_a_lire, TAILLE_ALLOUEE, stdin);
```

(`fgets` ne lit que `TAILLE_ALLOUEE`-1 caractères et garantit le `'\0'` en fin de chaîne.

Si un `'\n'` a été lu, il est *inclu* dans la chaîne. Un exemple de comment le supprimer est fourni slide 23.)

Jean-Cédric Chappelier

Formats de lecture `scanf` (1/2)

très similaires aux formats de `printf` (Avant tout : `man 3 scanf`)

Différences notoires cependant :

- format supplémentaire pour les chaînes de caractères : `%[...]` permet de spécifier les caractères à lire

  Exemple : ne lire une séquence composée uniquement de lettres majuscules :
  ```
  scanf("%[A-Z]", chaine);
  ```

  Note : Si le premier caractère est un `^` : la chaîne lue **ne** doit **pas** contenir les caractères en question.
  Exemple : lire tout *sauf* les retours à la ligne :
  ```
  scanf("%[^\n]", chaine);
  ```

- n'importe quel blanc (au sens de `isspace()`) absorbe tous les blancs.

  Exemple : lire tout *sauf* les blancs initiaux les retours à la ligne :
  ```
  scanf(" %[^\n]", chaine);
  ```

  Si l'on entre `"  baba au rhum\n"`, avec cet exemple on lira `"baba au rhum"`, alors qu'avec l'exemple précédent on aurait lu `"  baba au rhum"`.

Jean-Cédric Chappelier

Formats de lecture `scanf` (2/2)

Autres différences notoires avec les formats de `printf` :

- Utiliser `%lf` pour lire des `double`

  

  (alors que la norme ANSI C89 ne supporte pas `%lf` pour `printf` :-([mais C99 oui... ...heureusement !])

- « `%*...` » indique un champ non affecté (au lieu d'une taille donnée par une variable).

  Exemple : `scanf("%d%*d%lf", &i, &x);` lira un entier (et l'affectera à `i`), lira un autre entier sans l'affecter nulle part, puis lira un `double` et l'affectera à `x`.

- `%n` ne lit rien sur l'entrée, mais affecte le nombre de caractères lus à ce stade.

Par ailleurs, `%`*`n`*`s`, où *`n`* est un nombre entier, est très pratique pour contrôler le nombre maximum de caractères lus.
Exemple : `scanf("%19s", chaine);`

Jean-Cédric Chappelier

Contrôle strict de l'entrée

Vous avez sûrement déjà remarqué que la lecture avec `scanf` est parfois « *capricieuse* » (ou difficile à régler dans les cas un peu complexes).

Exemple tiré des séries d'exercices :

```c
do {
  printf("Entrez un nombre entre 1 et 10 : ");
  fflush(stdout);
  scanf("%d", &i);
} while ((i < 1) || (i > 10));
```

Si vous tapez `'a'` (ou n'importe quoi qui ne soit pas un nombre)
☞ boucle infinie ! !

Comment éviter cela ?

☞ en contrôlant l'état du tampon `stdin` (et en « jetant à la poubelle » ce qui ne convient pas)

Jean-Cédric Chappelier

Contrôle strict de l'entrée

D'une façon générale (à un niveau avancé), il ne faut **jamais** utiliser `scanf` pour lire des entrées.

Préférez `getc` (lecture caractère par caractère) et gérez vous-même complètement l'entrée.
(mieux : il existe des aides pour cela, comme le langage (f)lex)

Ceci dit, voici une solution permettant de contrôler un peu mieux ce qui se passe avec des `scanf` :

```
do {
  printf("Entrez un nombre entre 1 et 10 : "); fflush(stdout);
  j = scanf("%d", &i);
  if (j != 1) {
    printf("Je vous ai demandé un nombre, pas du charabia !\n");
    /* vide le tampon d'entrée */
    while (!feof(stdin) && !ferror(stdin) && getc(stdin) != '\n');
  }
} while (!feof(stdin) && !ferror(stdin) && ((j!=1) || (i<1) || (i>10)));
```

La valeur retournée par `scanf` est le nombre de champs correctement saisis ou `EOF` si une erreur est survenue avant toute conversion.

Sortie erreur standard

En plus de `stdin` et `stdout`, il existe une **sortie d'erreur standard**, `stderr`.

Par défaut, `stderr` est envoyée sur le terminal, comme `stdout`.
Mais il s'agit bien d'un flot séparé !

De plus, `stderr` n'a pas de mémoire tampon. L'écriture sur `stderr` se fait donc directement (on n'a pas besoin de `fflush`).

☞ Conseil : Pour afficher des messages d'erreur depuis votre programme, préférez `stderr` plutôt que `stdout`.

☞ La question c'est : comment fait-on ?...

Utilisation des fichiers

Le type permettant de représenter (le flot associé à) un fichier dans un programme C est `FILE` défini dans `stdio.h`.

L'association d'un flot d'entrée-sortie avec le fichier se fait par le biais de la fonction spécifique `fopen`.

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

Exemple :

```
FILE* entree = NULL;
char nom_entree[FILENAME_MAX+1];
...
entree = fopen(nom_entree, "r");
```

associe le flot `entree` avec le fichier physique dont le nom est contenu dans `nom_entree`.

Lien avec un fichier

Dans le cas des fichiers textes (fichiers lisibles par les humains), les « modes » d'ouverture possibles sont :

`r`	en lecture
`w`	en écriture (écrasement)
`a`	en écriture (à la fin)

On peut de plus ajouter un `+` après l'un quelconque des modes ci-dessus pour ouvrir le fichier en lecture et écriture (mais attention à la position de la tête de lecture et à l'état du tampon (`fflush`) !).

Si l'on souhaite manipuler des fichiers binaires, il faut ajouter `b` à la fin.

Exemples :

```
fichier1 = fopen(nom1, "rb");
fichier2 = fopen(nom2, "w+");
fichier3 = fopen(nom3, "a+b");
```

Lien avec un fichier : erreurs

En cas d'erreur d'ouverture, la fonction `fopen` retourne la valeur `NULL`.

On serait donc bien avisé d'écrire ses ouvertures de fichiers de la façon suivante :

```
entree = fopen(...);
if (entree == NULL) {
   /* gestion de l'erreur */
} else {
   /* suite (avec un fichier entree valide) */
}
```

Pour en savoir un peu plus sur l'erreur qui s'est produite, on peut utiliser `strerror(errno)` ou `perror`.

☞ `man 3 errno`, `man 3 perror`, `man 3 strerror`

Utilisation des flots

L'utilisation de variables de type `FILE*` dans les programmes pour réaliser les entrées-sorties se fait ensuite de façon similaire aux flots particuliers `stdin` et `stdout`, en utilisant `fprintf` et `fscanf` au lieu de `printf` et `scanf`.

Exemple :

```
FILE* entree = NULL;
FILE* sortie = NULL;
...
entree = fopen(nom1, "r");
sortie = fopen(nom2, "a");
/* tests de validité */
...
 /* lit un entier dans le fichier "entree" */
fscanf(entree, "%d", &i);

/* et l'écrit dans le fichier "sortie" */
fprintf(sortie, "%d\n", i);
```


Jean-Cédric Chappelier

Utilisation des flots (2)

Une fonction utile pour tester si la lecture d'un fichier est terminée est la fonction `feof(FILE*)`, qui retourne une valeur non nulle si la fin de fichier a été atteinte.

`ferror(FILE*)` de son coté teste le statut d'erreur du fichier et retourne une valeur non-nulle si une erreur s'est produite sur le fichier
(atteindre la fin de fichier **n'**est **pas** une erreur).

Exemple :

```
while ( !feof(entree) && !ferror(entree) )  {
   ...
   /* lecture du fichier entree */
}
```

Fermeture des flots

La fermeture du flot se fait par la fonction

`fclose(FILE*)`

Exemple :

```
fclose(entree);
```

Attention ! NE PAS oublier de fermer tout fichier ouvert !
En particulier en écriture : vous risqueriez sinon d'avoir des surprises...

Exemple de lecture à partir d'un fichier

Exemple de programme de lecture d'un fichier texte de nom « `test` » :

```c
#include <stdio.h>
int main(void) {
  char const nom_fichier[] = "test";
  FILE* entree = NULL;

  entree = fopen(nom_fichier, "r");
  if (entree == NULL) {
    fprintf(stderr,
            "Erreur : impossible de lire le fichier %s\n",
            nom_fichier);
  } else {
#define TAILLE_MAX 10
    char mot[TAILLE_MAX+1] = "";

    while (fscanf(entree, "%10s", mot) == 1) {
      printf("lu : ->%-*s<-\n", TAILLE_MAX, mot);
    }
    fclose(entree) ;
  }
  return 0; /* pour le moment... en attendant le cours 10 */
}
```


Jean-Cédric Chappelier

Écriture dans un fichier (1/2)

Exemple de programme d'écriture dans un fichier texte :

```c
#include <stdio.h>
#include <string.h>

int main(void) {
  char nom_fichier[FILENAME_MAX+1] = "";
  FILE* sortie = NULL;

  {
    size_t len = 0;
    do {
      printf("Dans quel fichier voulez vous écrire ?\n");
      fgets(nom_fichier, FILENAME_MAX+1, stdin);
      len = strlen(nom_fichier);
      if ((len != 0) && (nom_fichier[--len] == '\n')) {
        nom_fichier[len] = '\0';
      }
    } while ((len == 0) && !feof(stdin) && !ferror(stdin) );
  }

  // ... à suivre
```


Jean-Cédric Chappelier

Écriture dans un fichier (2/2)

```
// ... suite ...
  if (nom_fichier[0] != '\0') {
    sortie = fopen(nom_fichier, "a");
    if (sortie == NULL) {
      fprintf(stderr,
              "Erreur : impossible d'écrire dans le fichier %s\n",
              nom_fichier);
    } else {
#define MAX_PHRASE 513    // nombre arbitraire plus 1
      char phrase[MAX_PHRASE] = "";

      printf("Entrez une phrase :\n");
      fgets(phrase, MAX_PHRASE, stdin);
      fputs(phrase, sortie);
      fclose(sortie);
    }
  }

  return 0; /* pour le moment... en attendant le cours 10 */
}
```


Jean-Cédric Chappelier

Flots standards

On a donc `printf(` = `fprintf(stdout,`
et `scanf(` = `fscanf(stdin,`

Une façon élégante de pouvoir choisir l'entrée (ou la sortie) est de faire quelque chose comme :

```
FILE* sortie = NULL; /* sortie générique */

if (choix == ...) /* en fonction du choix */ {
  sortie = fopen(...)
  ...
} else { /* choix final : stdout */
  sortie = stdout;
}

... /* et dans toute la suite ne faire que des : */
fprintf(sortie, ...);
```

De même, tous les messages d'erreur devront s'écrire :

`fprintf(stderr, message...);`

Jean-Cédric Chappelier

Fichiers binaires

Comment faire si l'on veut sauvegarder directement les valeurs stockées en mémoire et non pas leurs écritures en décimal ?

☞ sauvegarde en binaire

Il faut pour cela :

- ouvrir le fichier pour une écriture en binaire
  `sortie = fopen(nom_fichier, "wb");`
- utiliser la commande `fwrite` au lieu de `fprintf` :

```
size_t fwrite(const void* ptr, size_t taille_el,
              size_t nb_el, FILE* fichier);
```

`fwrite` écrit dans le fichier *fichier*, *nb_el* éléments, chacun de taille *taille_el*, stockés en mémoire à la position pointée par *ptr*.
`fwrite` retourne le nombre d'éléments effectivement écrits.

Jean-Cédric Chappelier

fwrite

Exemple :

```c
#include <stdio.h>
#define TAILLE 12
int main(void) {
  int tab[TAILLE];
  for (size_t i = 0; i < TAILLE; ++i) tab[i] = i*i - 6*i;

  FILE* sortie = fopen("test-fwrite.bin", "wb");
  if (sortie != NULL) {
    size_t nb_ok = fwrite(tab, sizeof(int), TAILLE, sortie);
    fclose(sortie);
    if (nb_ok != TAILLE) {
      fprintf(stderr, "Unable to write %zu elements, "
              "wrote only %zu", TAILLE, size_ok);
    }
  }
  return 0;
}
```

Fichiers binaires (lecture)

La lecture d'un fichier binaire se fait de façon similaire en utilisant

```
size_t fread(void* ptr, size_t taille_el, size_t nb_el, FILE* fichier);
```

Exemple :

```
#include <stdio.h>
#define TAILLE 12
int main(void) {
  FILE* entree = fopen("test-fwrite.bin", "rb");
  if (entree != NULL) {
    int tab[TAILLE];
    size_t nb_ok = fread(tab, sizeof(int), TAILLE, entree);
    fclose(entree);
    if (nb_ok != TAILLE) {
      fprintf(stderr, "Unable to read %zu elements, "
              "read only %zu", TAILLE, nb_ok);
    } else for (size_t i = 0; i < TAILLE; ++i)
        printf("tab[%2zu] = % 3d\n", i, tab[i]);
  }
  return 0;
}
```


Jean-Cédric Chappelier

Diverses manipulations de fichier

`int fseek(FILE* fichier, long offset, int depart)`

repositionne la « tête de lecture ».

La nouvelle position est *offset* octets à partir de *depart*, qui peut valoir :

- `SEEK_SET` : début du fichier
- `SEEK_CUR` : position courante
- `SEEK_END` : la fin du fichier

`long ftell(FILE* fichier)`

retourne la position de la « tête de lecture » (à partir du début de fichier)

`void rewind(FILE* fichier)`

= `(void)fseek(fichier, 0L, SEEK_SET)`

`int ferror(FILE* fichier)`

teste le statut d'erreur du *fichier* : retourne une valeur non-nulle si une erreur s'est produite sur le fichier.

`void clearerr(FILE* fichier)`

remet à zéro le statut d'erreur du *fichier*

Flots dans des chaînes

Supposons que l'on écrive un programme avec une interface graphique que nous gérons nous-même dans le programme.

On ne voudra évidement pas envoyer nos messages sur `stdout`, mais dans notre interface.

Comment faire ?

☞ avec des **flots dans des chaînes de caractères**

Ils se manipulent comme les flots vus précédemment sauf que l'on utilise `sprintf` (`sscanf` existe aussi ☞ bon moyen pour convertir une chaîne en un autre type)

```
sprintf(chaine, FORMAT, variables...);
```

Exemple :

```
char adresse[129];
sprintf(adresse, "%5d %.122s", code, ville);
```


Jean-Cédric Chappelier

Recommandation

Quelques soient les cas, **toujours faire attention à la taille des objets manipulés !**

En particulier quand on lit de sources extérieures (fichier ou terminal) : toujours imposer une borne sur la taille lecture

Autre exemple : quand on copie des chaînes de caractères
(préférer `strncpy` à `strcpy`)

POURQUOI ?

Sinon c'est la porte ouverte à des débordements de mémoire (« buffer overflow ») et des attaques possibles (cf cours semaine 9).

Les entrées/sorties

Clavier / Terminal : `stdin` / `stdout` et `stderr`

Fichier de définitions : `#include <stdio.h>`

Utilisation :

écriture : `int printf("FORMAT", expr1, expr2, ...);`

lecture : `int scanf("FORMAT", ptr1, ptr2, ...);`

Saut à la ligne : `'\n'`

Lecture d'une ligne entière :

```
char* fgets(char* s, int size, FILE* stream);
```

Les entrées/sorties (fichiers)

Type : `FILE*`

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

Mode :
`"r"` en lecture, `"w"` en écriture (écrasement), `"a"` en écriture (à la fin), suivit de '`+`' pour ouverture en lecture et écriture, et/ou de '`b`' pour fichiers en binaires

Écriture :
`fprintf(FILE*, ...)` pour fichiers textes
`size_t fwrite(const void* adr_debut, size_t taille_el, size_t nb_el, FILE*);` pour les fichiers binaires

Lecture :
`fscanf(FILE*, ...)` pour fichiers textes
`size_t fread(void* adr_debut, size_t taille_el, size_t nb_el, FILE*);` pour les fichiers binaires

Test de fin de fichier : `feof(FILE*)`

Fermeture du fichier : `fclose(FILE*)`

Jean-Cédric Chappelier

[hors cours] Monsieur, et en C++... ? [hors cours]

L'intégration des entrées/sorties est nettement mieux faite en C++ (pas de « langage dans le langage ») via les opérateurs `<<` et `>>` (lesquels peuvent être surchargés !).

Par ailleurs, `stdin` s'appelle `std::cin` et `stdout`, `std::cout`.
(Note : on peut simplifier en utilisant la directive « `using namespace std;` »)

Exemple :

```
cout << "Un message, et un entier : i=" << i << endl;
cin >> i;
```

Conseil : ne **jamais** utiliser `printf` ni `scanf` en C++.

[hors cours] Monsieur, et en C++... ? (2) [hors cours]

Concernant les fichiers, la syntaxe est également plus simple qu'en C et utilise les même opérateurs `<<` et `>>`.

`ifstream` (défini dans `<iostream>`) est le type (la classe) qui représente un flot d'*entrée* depuis un fichier (similaire à `cin`) et `ofstream`, un flot de *sortie* vers un fichier (similaire à `cout`).

Exemple :

```
string nom_fichier("test");
ifstream entree(nom_fichier);

if (entree.fail()) {
    cerr << "Erreur : impossible de lire le fichier "
         << nom_fichier << endl;
} else {
    string mot;

    while (!entree.eof()) {
        entree >> mot ;
        cout << mot << endl;
    }
    entree.close() ;
}
```

Ce que j'ai appris aujourd'hui

- à faire communiquer mon programme avec le « monde extérieur » pour
  - afficher des résultats ;
  - saisir des données ;
  - sauvegarder et relire des données ;
- à créer et lire des fichiers ;
- à formater mes résultats.

☞ je peux maintenant écrire des programmes pouvant **interagir** avec l'utilisateur, mais aussi **manipuler des fichiers** et donc donner de la **persistance** aux données créées.