

# Structures de données (LU2IN006) - Cours 2

## Quelques rappels de C : Entrées/sorties et compilation séparée

Nawal Benabbou

Licence Informatique - Sorbonne Université

2021-2022



## Entrées/sorties en langage C

On appelle entrées/sorties les échanges de données entre la mémoire de la machine et ses périphériques (clavier, écran, disque dur, imprimante, réseau...). Le type `FILE` permet de stocker les informations relatives à la gestion d'un flux de données, comme par exemple la position actuelle de la tête de lecture/écriture, le type d'accès, l'état d'erreur etc.

## Entrées/sorties standards

Trois flux pré-ouverts :

- `stdin` : flux standard d'entrée de l'application (clavier par défaut).
- `stdout` : flux standard de sortie (console de l'application par défaut).
- `stderr` : flux standard d'erreur (console de l'application par défaut).

# Saisie au clavier : scanf

## La fonction scanf

La fonction `scanf` est une fonction de la bibliothèque `<stdio.h>` permettant de lire des données *formatées*, en provenance du flux `stdin` (entrée standard). Cette fonction prend en paramètre une chaîne de caractères décrivant le format du flux entrant attendu, ainsi qu'une liste de pointeurs pour stocker le résultat de la lecture. L'acquisition s'arrête après avoir rencontré le caractère `'\n'` (saut de ligne). Pour chaque conversion réussie, `scanf` place l'objet converti dans l'espace pointé par le pointeur associé.

**Exemple :** lecture d'un entier suivi d'une chaîne de caractères.

```
1  int i;  
2  char chaine[20];  
3  scanf("%d_%s", &i, chaine);
```

**Observation :** la fonction `scanf` est plus complexe qu'il ne paraît.

Par exemple, on peut avoir des problèmes quand :

- on fait une petite erreur de saisie (interruption de lecture).
- on souhaite lire une chaîne de caractères avec des espaces (problèmes liés aux caractères "blanc").
- on entre une chaîne de caractères trop longue (saisie non sécurisée).

# Scanf et les erreurs de saisie

Si en cours d'analyse, une des conversions échoue, la fonction `scanf` s'arrête brutalement, sans effectuer les conversions suivantes. Le morceau de chaîne de caractères non traité reste dans la mémoire tampon. Par exemple :

```
1 int code, id;
2 char nom[20];
3 printf("Entrer_votre_code_postal_et_identifiant.\n");
4 scanf("%d%d",&code,&id); //Saisie : 75001 258p5 (Entree)
5 printf("Saisie_: %d%d\n", code, id); //Affiche : 75001 258
6 printf("Entrer_maintenant_votre_nom_de_famille.\n");
7 scanf("%s",nom); // le programme ne s'arrete pas...
8 printf("Bonjour_%s\n", nom); //Affiche "Bonjour p5"
```

**Solution possible :** vider la mémoire tampon après chaque appel à la fonction `scanf`, et vérifier le nombre de conversions réussies.

```
1 int n = scanf("%d%d",&code,&id);
2 char c = getchar(); //lit un caractere dans la memoire tampon
3 while (c!='\n'){
4     c = getchar();
5 }
6 if (n!=2){
7     printf("Erreur_durant_la_lecture");
8     return 0; //on pourrait demander une nouvelle saisie
9 }
```

**Remarque :** Sur l'exemple, le test en ligne 6 ne suffit pas à détecter l'erreur...

# Scanf et la gestion des blancs

## Différences entre les formateurs %s et %c

Le formateur %s :

- s'arrête au premier caractère "blanc" (espace, tabulation, saut de ligne).
- supprime les caractères "blanc" en début de chaîne.
- ajoute le caractère de fin '\0'.

Le formateur %c :

- n'ignore pas les caractères "blanc".
- nécessite de connaître la taille exacte de la chaîne.
- n'ajoute pas le caractère de fin '\0'.

Exemple :

```
1 char nom[20];
2 printf(" Entrer votre nom et prénom.\n");
3 scanf("%s", nom); //Saisie : abc def (Entree)
4 printf("%s\n", nom); //Affiche "abc"
5
6 char ville[20];
7 printf(" Entrer votre ville de naissance.\n");
8 scanf("%3c", ville); //le programme ne s'arrete pas
9 printf("%s\n", ville); //Affiche " de<:V"
```

# Scanf et la saisie non sécurisée

## Dépassement mémoire

Lors de la saisie d'une chaîne de caractères, la fonction `scanf` ne fait aucune vérification sur la taille de l'espace mémoire disponible pour stocker la chaîne. Si la chaîne à stocker est trop grande, `scanf` écrit les caractères en trop à la suite. La fonction écrit alors dans des zones mémoires qui n'étaient pas prévues pour cela, écrasant d'autres informations en mémoire potentiellement importantes pour le programme ou pour d'autres. On parle de *dépassement mémoire*, ou encore de *buffer overflow* en anglais.

## Exemple :

```
1 char nom[5]; //4 lettres maximum (pour le '\0' final)
2 printf("Entrez votre nom\n");
3 scanf("%s", nom); //Saisie: abcdefgh (Entree)
4 printf("Bonjour %s\n", nom); //Affiche : "Bonjour abcdefgh"
5 //Affiche a la fin "*** stack smashing detected ***"
6 //Pas de warning si l'allocation etait dynamique...
```

**Une solution possible, mais peu satisfaisante :** allouer un très grand tableau de caractères (ne protège pas des attaques dites "par buffer overflow" ...).

⇒ Pour une version plus sécurisée, on peut utiliser la fonction `fgets` en remplacement de la fonction `scanf` (ou de `fscanf` qui permet notamment de lire dans un fichier, avec encore d'autres défauts...).

# Saisie au clavier avec la fonction fgets

Signature : `char* fgets(char* chaine, int tailleMax, FILE* flux );`

La fonction `fgets` est une fonction de la librairie standard qui permet de lire des données (non formatées) provenant d'un flux (`stdin`, ou autre). La lecture s'interrompt après :

- avoir lu `tailleMax-1` caractères (pour pouvoir ajouter le `'\0'` final),
- ou avant si la fonction rencontre un caractère de saut de ligne (`'\n'`) ou un caractère fin de flux (EOF).

```
1 char nom[20];  
2 printf("Entrer votre nom et prénom.\n");  
3 fgets(nom, 20, stdin); //Saisie : a bc (Entree)  
4 printf("%s\n", nom); //Affiche bien "a bc" mais...
```

**Petit hic :** `fgets` ajoute un saut de ligne à la fin de la chaîne lue, correspondant à l'appui sur la touche "Entrée".

**Une solution possible :** remplacer le `'\n'` par le caractère de fin `'\0'`.

```
1 char nom[20];  
2 printf("Entrer votre nom et prénom.\n");  
3 fgets(nom, 20, stdin); //Saisie : a bc (Entree)  
4 nom[strlen(nom)-1] = '\0'; // il faut inclure <string.h>  
5 printf("%s\n", nom); //Affiche "a bc" sans saut
```

# La fonction fgets et la mémoire tampon

## Observation : fgets et mémoire tampon

Bien que fgets permette d'éviter les dépassements mémoire, rien n'empêche l'utilisateur de taper trop de caractères. Dans ce cas, les caractères en trop se retrouvent dans la mémoire tampon après l'appel à la fonction (dans le cas où le flux est stdin). Si on ne souhaite pas les récupérer, il faut penser à vider la mémoire tampon juste après l'appel (comme avec scanf).

```
1 char nom[5];
2 printf("Entrer votre nom.\n");
3 fgets(nom,5,stdin); //Saisie : abcdefghijklmn (Entree)
4 printf("%s\n", nom); //Affiche "abcd" sans saut
5 char c = getchar();
6 while (c!='\n' && c!=EOF){
7     c = getchar();
8 }
```



# La fonction fgets et la lecture de texte formaté

## Lecture d'un texte formaté avec fgets et sscanf

Pour pouvoir lire un texte formaté, on peut utiliser la fonction fgets suivie de la fonction sscanf, qui elle lit du texte formaté dans la chaîne de caractères qu'on lui fournit en paramètre.

```
1 char lus [20];  
2 char nom[20];  
3 char prenom[20];  
4 printf("Entrer votre nom et prenom.\n");  
5 fgets(lus,20,stdin); //Saisie : ab cde (Entree)  
6 printf("%s\n", lus); //Affiche "ab cde" avec saut  
7 sscanf(lus,"%s%s", nom, prenom);  
8 printf("%s %s\n", nom, prenom); //Affiche "ab cde" sans saut
```

**Remarque :** le saut de ligne contenu dans la chaîne lus est ignoré par le sscanf, donc pas besoin de le retirer.

**Attention :** sscanf peut aussi conduire à des débordements mémoire... par exemple ici, que se passe-t-il quand prenom est de taille 2 ? (sur l'ordinateur de Nawal, aucun problème détecté...).

**Note :** la fonction fgetc ne lit pas du texte formaté, mais permet de lire caractère par caractère (permet de faire des vérifications).

## Ouverture de fichier

L'ouverture se fait avec la fonction `fopen`. Différents modes d'ouverture :

- "r" : lecture seule (le fichier doit exister).
- "w" : écriture seule (si le fichier n'existe pas, il est créé).
- "a" : ajout en fin (si le fichier n'existe pas, il est créé).
- "r+" : lecture et écriture (le fichier doit exister).
- "w+" : lecture et écriture (si le fichier existe, destruction avant création).
- "br" : lecture en binaire (le fichier doit exister).
- etc.

La fonction `fopen` retourne `NULL` si l'ouverture échoue.

## Fermeture de fichier

Tout fichier ouvert doit être fermé avec la fonction `fclose` (comme `malloc` et `free`), qui libère la mémoire en le supprimant de la mémoire vive. La fonction `fclose` retourne 0 si la fermeture s'est bien passée, et EOF sinon.

# Lecture de fichier avec la fonction fgets

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int lecture(char* nomFichier){
5      char ligne[256];
6      char nom[256];
7      int id;
8      FILE* fic = fopen(nomFichier, "r"); // "r" pour read
9      if (fic == NULL){
10         printf("Probleme d'ouverture de fichier");
11         return 0;
12     }
13     while (fgets(ligne, 256, fic)!=NULL){ // lit une ligne
14         /* Traitement de la ligne avec sscanf */
15         if (sscanf(ligne,"%s%d", nom, &id) == 2){
16             printf("Nom: %s, ID: %d", nom, id);
17         }else{
18             printf("Format de ligne incorrect");
19             return 0;
20         }
21     }
22     fclose(fic); //on ferme le fichier avant de quitter
23     return 1;
24 }
```

**Remarque :** la fonction fgets permet aussi de lire dans un fichier.

# Écriture dans un fichier

## Fonctions d'écriture : fputc, fputs et fprintf

La fonction fputc permet d'écrire un caractère à la fois, tandis que fputs permet d'écrire une chaîne de caractères. La fonction fprintf permet d'écrire une chaîne formatée dans un flux.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int ecriture(char* nomFichier){
5      int i=0;
6      FILE* fic = fopen(nomFichier, "w"); // "w" pour write
7      if (fic == NULL){
8          printf("Probleme d'ouverture du fichier");
9          return 0;
10     }
11     fputs("Debut du Fichier\n",fic);
12     while (i<100){
13         fprintf(fic, "Ligne %d",i);
14         fputc('\n',fic); // '\n' peut etre ajoute dans fprintf
15         i=i+1;
16     }
17     fclose(fic); //on ferme le fichier avant de quitter
18     return 1;
19 }
```

# Compilation et programmation modulaire

## Programmation modulaire

Pour un programme, la modularité est le fait d'être composé en plusieurs parties, relativement indépendantes les unes des autres. La programmation modulaire offre la possibilité de découper un gros programme en modules (réunissant des fonctionnalités particulières) répartis dans différents fichiers.

## Avantages

La programmation linéaire :

- rend le code plus compréhensible,
- permet la réutilisation de code,
- supprime les risques d'erreurs en reprogrammant la même chose,
- permet de développer et d'améliorer des parties de code indépendamment,
- facilite le travail collaboratif,
- simplifie la compilation en se concentrant sur les fichiers modifiés.

⇒ Nécessite une réflexion sur le découpage du code avant de se lancer.

# Module en langage C et exemple

## Module

Un module est composé de :

- un fichier d'en-tête (extension ".h"), décrivant l'interface du module . Il contient des déclarations de fonctions, de type (struct), de variables et éventuellement des inclusions de bibliothèques.
- un fichier source (extension ".c") contenant le code des fonctions déclarées dans le fichier d'en-tête.

```
/* Fichier tableau.h */
#ifndef TABLEAU_H
#define TABLEAU_H

typedef struct tableau {
    char* tab;
    int taille;
} Tableau;

int recherche(int x, Tableau* t);

#endif
```

```
/* Fichier tableau.c */
#include "tableau.h"

int recherche(int x, Tableau* t){
    int i=0;
    while (i<t->taille){
        if (t->tab[i]==x){
            return i;
        }else{
            i=i+1;
        }
    }
    return -1;
}
```

# Exemple

```
1  /* Fichier lecture.h */
2  #ifndef LECTURE_H
3  #define LECTURE_H
4  #include "tableau.h"
5  #include <stdio.h>
6
7  Tableau* lire (int n, FILE* f);
8
9  #endif
```

```
1  /* Fichier lecture.c */
2  #include "lecture.h"
3  #include <stdlib.h>
4
5  Tableau* lire (int n, FILE* f){
6      Tableau* p = (Tableau*)malloc(sizeof(Tableau));
7      p->tab = (char*)malloc(sizeof(char)*n);
8      p->taille=n;
9      if (fgets(p->tab,n,f) != NULL){
10         return p;
11     }else{
12         printf("Probleme de lecture");
13         return NULL;
14     }
15 }
```

# Exemple

```
1  /* Fichier main.c */
2  #include "tableau.h"
3  #include "lecture.h"
4  #include <stdio.h>
5
6  void main(){
7      FILE* f = fopen("nomFic.txt", "r");
8      if (f!=NULL){
9          Tableau* p = lire(256, f);
10         printf("%s", p->tab);
11     }
12 }
```

## Compilation séparée

On peut maintenant compiler les fichiers sources séparément, puis créer l'exécutable à partir des fichiers compilés. Sur l'exemple, on peut faire :

- gcc -c tableau.c (crée le fichier "tableau.o")
- gcc -c lecture.c (crée le fichier "lecture.o")
- gcc -c main.c (crée le fichier "main.o")
- gcc -o main main.o tableau.o lecture.o (crée l'exécutable main)



## L'outil make

L'outil `make` est un programme permettant d'automatiser la génération de fichiers à partir de code source, en exécutant un certain nombre de commandes `shell`. Contrairement à un simple script `shell`, `make` exécute les commandes uniquement si elles sont nécessaires. Dans notre cas, cela permet de recompiler uniquement les fichiers qui ont été modifiés depuis la dernière compilation.

## Makefile

L'outil `make` utilise des fichiers, appelés `Makefile`, qui spécifient les règles à suivre pour créer les fichiers désirés. Les règles sont sous la forme :

```
1 nom_cible : dependances
2     commandes
```

où `nom_cible` est le nom du fichier à créer, `dependances` est l'ensemble des fichiers nécessaires à sa création, et `commandes` donne les commandes à exécuter (précédées d'une tabulation).

# Fonctionnement de l'outil make

Quand on tape la commande `make nom_cible` dans le terminal, l'outil `make` évalue la règle permettant de créer `nom_cible`. Si `nom_cible` n'est pas donné en argument de `make`, alors l'outil évalue la première règle rencontrée dans le `makefile`.

## Fonctionnement : une évaluation récursive

Évaluer une règle  $R$  se fait de récursivement en analysant ses dépendances de la manière suivante :

- Toute dépendance qui est la cible d'une autre règle doit être évaluée avant de pouvoir exécuter les commandes de la règle  $R$ .
- Une fois que toutes les dépendances de  $R$  ont été évaluées, les commandes de la règle  $R$  sont exécutées uniquement si la cible est plus ancienne que l'une de ses dépendances (au moins).

# Retour sur l'exemple

```
1 all : main //pas necessaire ici car un seul executable
2
3 main : main.o lecture.o tableau.o
4     gcc -o main main.o lecture.o tableau.o
5
6 main.o : main.c lecture.h tableau.h
7     gcc -c main.c
8
9 lecture.o : lecture.c lecture.h tableau.h
10    gcc -c lecture.c
11
12 tableau.o : tableau.c tableau.h
13    gcc -c tableau.c
14
15 clean :
16    rm -f *.o main
```