Cours de Programmation 2 (Programmation C - structures de données) cours n°5 Compilation séparée, fichiers makefile Entrées-sorties (1re partie : fichiers de type texte)

Christophe Tollu
(Première version des diapos par Julien David)

A209 (poste 3691) - ct@lipn.univ-paris13.fr

2 avril 2024

Compilation sé

Séparer le code

La séparation

- Définition d'une structure de données `truc`
  → dans le **fichier d'entêtes** `truc.h`
- Déclaration des fonctions qui manipulent la structure `truc`
  → dans le même **fichier d'entêtes** `truc.h`
- Définition des fonctions qui manipulent la structure `truc`
  → dans le **fichier de fonctions** `truc.c`
- Le « programme » avec la fonction principale (`main`)
  → dans le **fichier principal** `main.c`

Exemple

On veut définir la notion de pixel

Un pixel se compose :

- d'un point (donné par ses coordonnées entières) ;
- d'une couleur.

Les structures

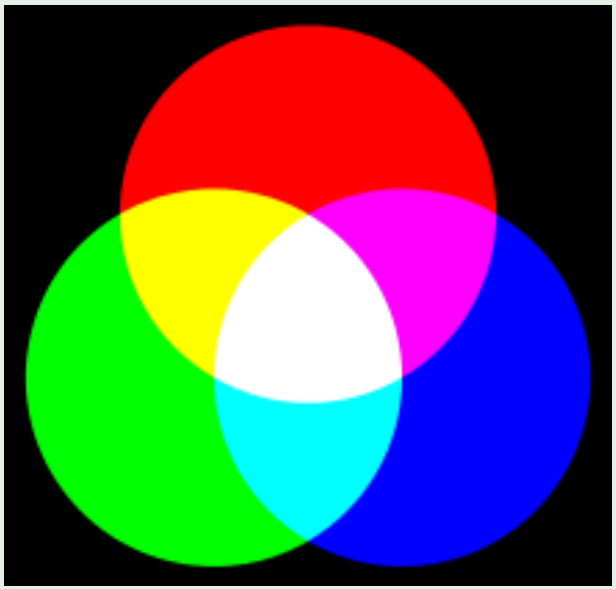
On aura donc besoin de définir :

- Un type structuré `point`.
- Un type structuré `couleur`.
- Un type structuré `pixel` qui utilise les deux premiers.

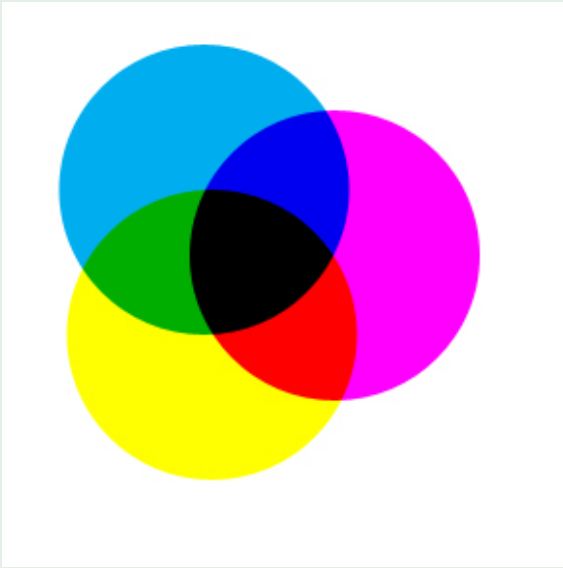
Une couleur ?

Il existe plusieurs façons de caractériser une couleur.

Modèle `RVB`



Modèle `CMJN`



Supposons que l'on choisisse le `RVB` .

Le modèle RVB

Modèle colorimétrique

- La couleur est représentée par des grandeurs numériques.
- Dans le modèle `RVB`, la couleur est obtenue par superposition de trois couleurs de base : le rouge, le vert et le bleu.
- L'intensité d'une couleur de base est codée par un entier (par exemple compris entre 0 et $2^8 - 1$, soit une valeur du type `unsigned char`).
- Le noir correspond aux intensités minimales des couleurs de base, le blanc aux intensités maximales.

Pour en savoir plus

- `https://fr.wikipedia.org/wiki/Rouge_vert_bleu`

Les structures

```c
/* Fichier point.h */
struct point_s{
  unsigned x;
  unsigned y;
};
typedef struct point_s point;
```

```c
/* Fichier couleur.h */
struct couleur_s{
  unsigned char rouge;
  unsigned char vert;
  unsigned char bleu;
};
typedef struct couleur_s couleur;
```

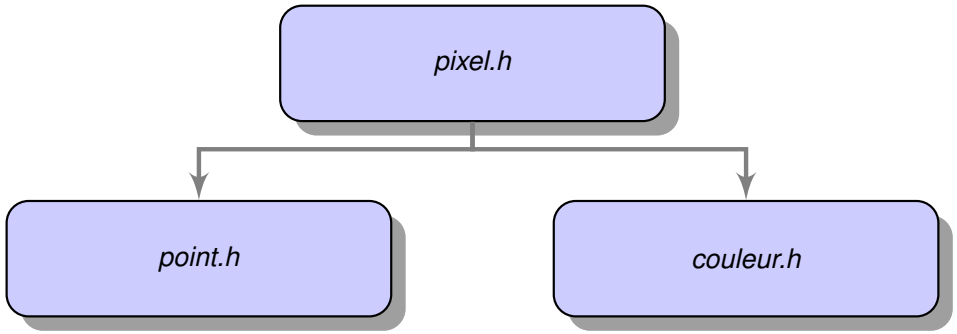
```c
/* Fichier pixel.h */
#include "point.h"
#include "couleur.h"

struct pixel_s{
  point coord;
  couleur col;
};
typedef struct pixel_s pixel;
```

Diagramme des dépendances

Dépendances

- La notion de `pixel` dépend de celle de `point` et de `couleur`,
- Ces dépendances sont identifiables grâce aux directives `#include`,
- On représente ces dépendances à l'aide d'un diagramme.



Les fonctions

Manipuler des structures

Lorsqu'on utilise un type de données (par exemple un structuré), **on peut souhaiter ne pas avoir besoin de connaître comment cette structure est définie pour la manipuler**.

Pour cette raison...

On va créer un ensemble de fonctions permettant de manipuler cette structure de données.

Déclarations des fonctions

```c
/* Fichier point.h */
/* Definition de la structure */
struct point_s{
  unsigned int x;
  unsigned int y;
};
typedef struct point_s point;

/* Declaration des fonctions */

point * creer_point(unsigned x, unsigned y);

int comparer_point(point p1, point p2);

void afficher_point(point p1);

void swap_point(point * p1, point * p2);
```

Déclarations des fonctions

```
1  /* Fichier couleur.h */
2  /* Definition de la structure */
3  struct couleur_s{
4    unsigned char rouge;
5    unsigned char vert;
6    unsigned char bleu;
7  };
8  typedef struct couleur_s couleur;
9
0  /* Declaration des fonctions */
1
2  couleur * creer_couleur (unsigned char r, unsigned char v, unsigned char b);
3
4  int comparer_couleur (couleur p1, couleur p2);
5
6  void afficher_couleur (couleur p1);
7
8  void swap_couleur (couleur * p1, couleur * p2);
```

Déclarations des fonctions

```c
/* Fichier pixel.h */
#include "point.h"
#include "couleur.h"

/* Definition de la structure */
struct pixel_s{
  point coord;
  couleur col;
};
typedef struct pixel_s pixel;

/* Declaration des fonctions */

pixel * creer_pixel (point pos, couleur c);

int comparer_point_pixel (pixel p1, pixel p2);

void afficher_pixel(pixel p1);

void swap_pixel(pixel * p1, pixel * p2);
```

Définitions des fonctions

Les fichiers `.c`

- Le fichier `truc.c` contient un `#include "truc.h"`.
- Si nécessaire, `truc.c` peut contenir d'autres `#include`.
- Chaque fonction déclarée dans `truc.h` doit être définie dans `truc.c`.

Par manque de place,

dans les diapos qui suivent,

- il manque les commentaires,
- on ne définit que certaines fonctions.

Définitions des fonctions

```c
/* Fichier point.c */
#include "point.h"
#include <stdio.h>
#include <stdlib.h>


point * creer_point (unsigned x, unsigned y) {
  point * res = malloc(sizeof(struct point_s));
  res.x = x;
  res.y = y;
  return res;
}

void afficher_point (point pt) {
  printf("X=%u Y=%u\n", pt.x, pt.y);
}
```

- On utilise la fonction `printf`, donc on inclut `stdio.h`
- On utilise la fonction `malloc`, donc on inclut `stdlib.h`

Définitions des fonctions

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

/* Definition des fonctions */

void afficher_pixel (pixel px) {
  afficher_point (px.coord);
  afficher_couleur (px.col);
}
```

- On peut faire appel à des fonctions définies dans d'autres fichiers.
- Ici, la déclaration de ces fonctions est accessible par `pixel.h`.

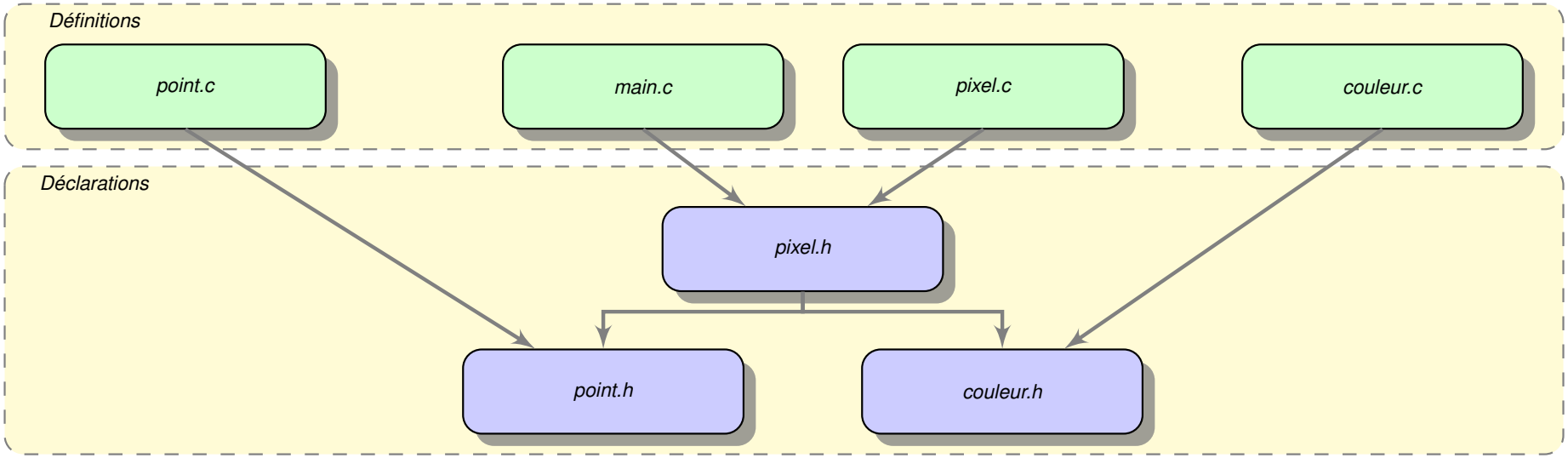
Définitions des fonctions

```c
/* Fichier main.c */
#include <stdlib.h>
#include "pixel.h"


int main() {
  pixel * px = creer_pixel (*creer_point(1,3), *creer_couleur(65,0,20));
  afficher_pixel(px);
  return EXIT_SUCCESS;
}
```

- On place toujours la fonction main dans un nouveau fichier.
- De cette façon on peut écrire plusieurs programmes en écrivant **plusieurs `main` dans autant de fichiers**.

Diagramme des dépendances



La compilation (1re phase)

Génération séparée de code objet

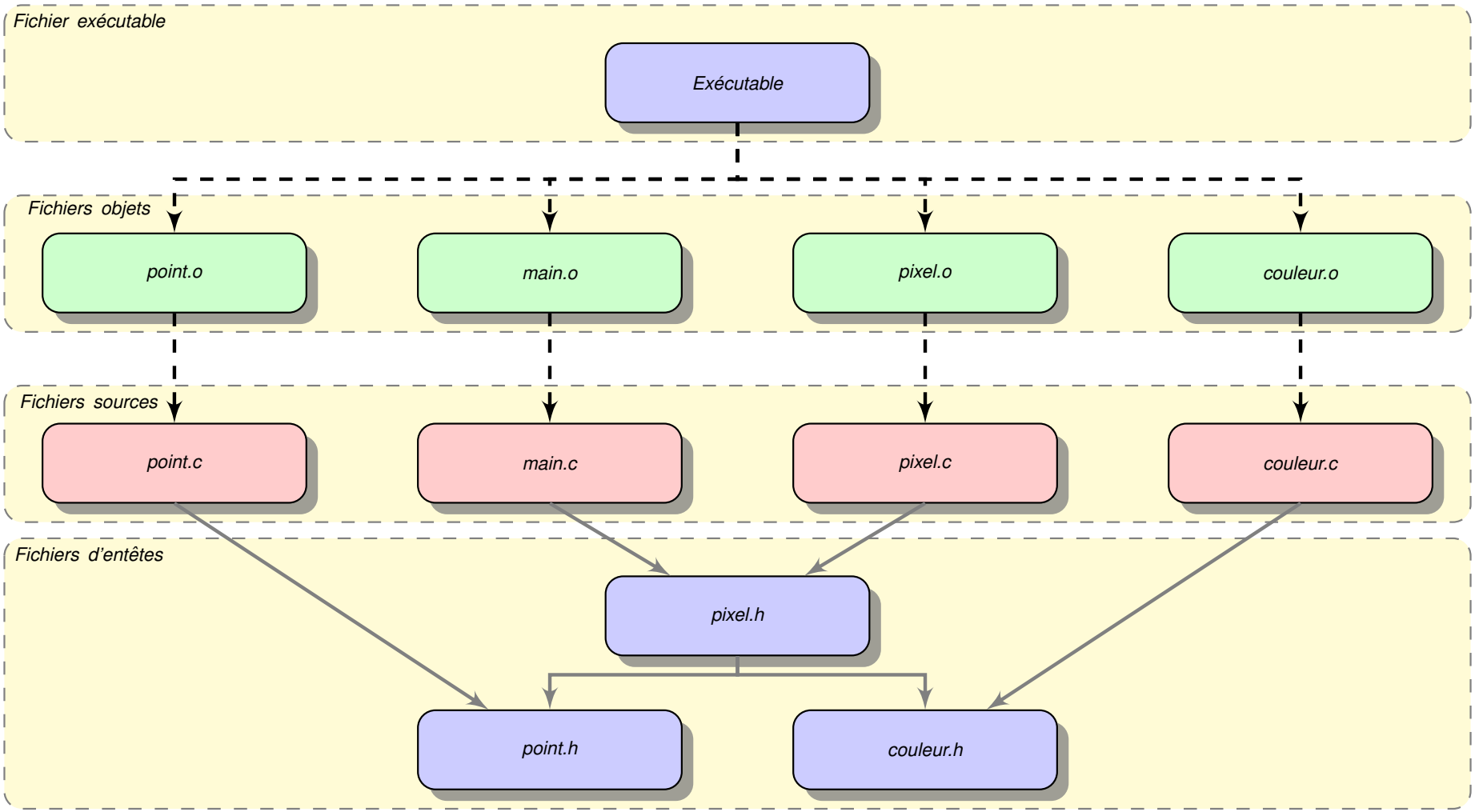
Pour compiler le programme,

- on compile (**partiellement**) chaque fichier `.c` séparément (exemple : `gcc -c -Wall -std=c99 point.c -o point.o` ou `gcc -c -Wall -std=c99 point.c`);
- pour chaque fichier `.c` on obtient un **fichier objet** `.o`;
- l'option `-c` dans la commande ci-dessus arrête la compilation après la **génération de code objet**;
- toutes les fonctions définies dans le fichier `.c` sont transformées en langage machine dans le fichier `.o`;

Attention

- l'option `-c` de `gcc` est indispensable pour arrêter la compilation après la génération de code objet !
- les fichiers objet **ne sont pas des fichiers exécutables** !

Diagramme des dépendances



La compilation (2nde phase)

Édition de liens

Pour compléter la compilation et engendrer le fichier exécutable,

- on lie l'ensemble des fichiers objets obtenus précédemment (**exemple** : `gcc point.o couleur.o pixel.o main.o -o programme`) ;
- on obtient un unique **fichier exécutable**.

Test

On peut **tester l'exécutable** en utilisant son nom comme une commande (avec les arguments quand ils sont requis) :
(exemple : `./programme` si on est dans le bon répertoire.)

Conflit(s) sur les inclusions

Gardes d'inclusion

- Certains fichiers `.h` peuvent être inclus plusieurs fois.
- Dans ce cas, à la compilation, il y aura des conflits d'inclusion.
- On va donc s'assurer que les fichiers `.h` soient inclus au plus une fois en insérant le contenu de chaque fichier `.h` dans une **directive conditionnelle** ; par exemple, le fichier `truc.h` sera encadré par

```
#ifndef _TRUC_H
#define _TRUC_H
...
#endif
```

Version finale des `.h` (gardes d'inclusion)

```
1  /* Fichier point.h */
2  #ifndef _POINT_H
3  #define _POINT_H
4  /* Definition de la structure */
5  struct point_s{
6    unsigned int x;
7    unsigned int y;
8  };
9  typedef struct point_s point;
0
1  /* Declaration des fonctions */
2
3  point * creer_point(unsigned x, unsigned y);
4
5  int comparer_point(point p1, point p2);
6
7  void afficher_point(point p1);
8
9  void swap_point(point * p1, point * p2);
20
21 #endif
```

Le Makefile

La commande `make`

La commande `make` permet de construire des programmes en composant différents fichiers sources.
Cette commande se configure grâce à un fichier : le `Makefile`.

Le Makefile

Le `Makefile`

Contient un ensemble d'informations permettant de construire l'exécutable :

- à partir de quels fichiers `.o` l'exécutable est-il construit ?
- comment convertir les fichiers `.c` en fichiers `.o` ? (on spécifie les options de compilation)

Si on a déjà construit le diagramme des dépendances,

- pour chaque flèche en pointillé dans le diagramme, on construit une **règle** dans le Makefile.

Le Makefile : syntaxe

Syntaxe d'une règle

`cible: sous-cible`$_1$ `sous-cible`$_2$ `...` `sous-cible`$_k$
`< TAB > commande à exécuter`

- Les **sous-cibles** sont des prérequis qui doivent être vérifiés avant l'exécution de la commande qui se trouve à la ligne suivante.
- Il est **obligatoire** de commencer la deuxième ligne par une **tabulation**.

Exemple

Pour compiler le fichier `point.c` et obtenir le fichier `point.o`

```
point.o: point.c point.h
< TAB > gcc -c -Wall -ansi point.c
```

Le Makefile : exemple naïf

```
1  all: point.o couleur.o pixel.o main.o
2    gcc point.o couleur.o pixel.o main.o -o programme
3
4  point.o: point.c point.h
5    gcc -c -Wall -std=c99 -Wfatal-errors point.c
6
7  couleur.o: couleur.c couleur.h
8    gcc -c -Wall -std=c99 -Wfatal-errors couleur.c
9
0  pixel.o: pixel.c pixel.h
1    gcc -c -Wall -std=c99 -Wfatal-errors pixel.c
2
3  main.o: main.c pixel.h
4    gcc -c -Wall -std=c99 -Wfatal-errors main.c
```

Pour compiler, il suffit d'exécuter la commande `make` (ou `make all`) depuis le répertoire où se trouve le `Makefile`.

Améliorer son Makefile

Les variables

- Il est possible de **déclarer des variables** dans le Makefile.
- On peut **définir d'autres cibles** dans la Makefile, qu'on « atteint » par la commande : `make <cible>`.
- Avec le fichier Makefile de la diapo suivante, la commande `make clean` aura pour effet de nettoyer le répertoire courant des fichiers `point.o couleur.o pixel.o main.o` et `programme`.

Le Makefile : exemple 2

```
1  # Indiquer le compilateur
2  CC=gcc
3
4  # Options du compilateur
5  CFLAGS=-Wall -std=c99 -Wfatal-errors
6
7  # Declarer ses propres variables
8  OBJETS=point.o couleur.o pixel.o main.o
9  PROG=programme
0
1  all: $(OBJETS)
2    gcc $(OBJETS) -o $(PROG)
3
4  point.o: point.c point.h
5    gcc -c $(CFLAGS) point.c
6
7  couleur.o: couleur.c couleur.h
8    gcc -c $(CFLAGS) couleur.c
9
20 pixel.o: pixel.c pixel.h
21   gcc -c $(CFLAGS) pixel.c
22
23 main.o: main.c pixel.h
24   gcc -c $(CFLAGS) main.c
25
26 clean:
27   rm $(PROG) $(OBJETS)
```

Améliorer son Makefile

Fichiers Makefile avancés

- On n'a vu que les bases de l'écriture de fichiers Makefile.
- Ils peuvent en réalité être beaucoup plus complexes et bien mieux décrits.
- À partir du prochain TP, vous écrirez un Makefile adapté à chacun de vos programmes.

Entrée Généralité

Entrées-sorties sous Linux

Fichiers

- Les entrées-sorties sous Linux sont traitées **uniformément par l'intermédiaire de fichiers**.
- Ces fichiers sont gérables par des **appels système** (primitives de bas niveau) et des **descripteurs** ou par des **fonctions C** (fonctions de haut niveau) et des **flux de données**.

Flux (de données) ?

- Les flux sont « une **abstraction** qui ajoutent automatiquement aux descripteurs de fichiers des tampons d'entrées-sorties, des verrous et des rapports d'état et d'erreur plus fins » (C. Blaess 2019).
- Un **tampon** (*buffer* en anglais) est un espace mémoire utilisé pour y stocker temporairement des données.

Flux de données

Flux (de données)

- Ils sont de **type FILE**, un type opaque défini dans `<stdio.h>` : il ne faut pas tenter d'accéder aux champs de la structure FILE ni utiliser d'objet de type FILE (on n'utilise que des pointeurs sur de tels objets).
- Les allocations et libérations de mémoire rendues nécessaires par la manipulation des flux sont régies par les fonctions C de manipulation des flux.

Trois flux standard

Ils sont ouverts au démarrage de tout programme exécuté sous Linux.

- `stdin` : flux d'**entrée standard**, ouvert en lecture seule ; par défaut, il s'agit du **clavier**.
- `stdout` : flux de **sortie standard**, ouvert en écriture seul ; par défaut, il s'agit de l'**écran**.
- `stderr` : flux d'**erreur standard**, ouvert en écriture seul ; il sert à afficher des informations sur l'exécution du programme ; par défaut, il s'agit aussi de l'**écran**.

Entrées-sorties standard

printf et scanf

- `printf("La somme de %u et de %u vaut %u", 4, 3, 4+3) ;`
- `scanf("%d", &val) ;`

fprintf et fscanf

- `fprintf(stdout, "La somme de %u et de %u vaut %u", 4, 3, 4+3) ;`
- `fscanf(stdin, "%d", &val) ;`

Ouverture / fermeture de fichiers en C

Les fichiers en C

Résumé

Tout comme pour les interactions avec le terminal :

- le programme n'interagit pas directement avec le disque dur ;
- c'est le système d'exploitation qui se charge de la communication entre les différents composants ;
- quand un programme (processus) veut accéder à un fichier, il demande au système de lui donner un accès ;
- le système va alors créer un espace mémoire où le programme pourra lire et/ou écrire ;
- cet espace mémoire est appelé **tampon** (*buffer* en anglais) ;
- on va voir aujourd'hui comment manipuler cet espace mémoire.

Les fichiers : un nouveau type de variable

Le type `FILE`

- Le type `FILE` permet de manipuler des fichiers.
- Il s'agit d'une structure (définie dans `<libio.h>`, incluse dans `<stdio.h>`) qui contient
  - l'adresse du tampon,
  - la position de la **tête de lecture/écriture** dans le fichier,
  - les droits que le programme possède sur le fichier.

En pratique

- On utilise toujours une variable de type `FILE *` .
- Un objet de type `FILE *` est (aussi) appelé un *flux (de données)* (*stream* en anglais).

Les fonctions : ouverture d'un fichier

Ouvrir un fichier

- `FILE *fopen(const char *path, const char *mode);`

Arguments

- `const char *path` : nom du fichier à ouvrir (comprenant le chemin absolu, si `path` commence par "/", ou relatif vers ce fichier).
- `const char *mode` : mode d'ouverture du fichier.

Les fonctions : ouverture d'un fichier

Commmande	Mode	Position	Fichier inexistant
"r"	lecture	Début	renvoie NULL
"r+"	lecture+écriture	Début	renvoie NULL
"w"	écriture	Début	Crée le fichier.
"w+"	lecture+écriture	Début	Crée le fichier.
"a"	écriture	Fin	Crée le fichier.
"a+"	lecture+écriture	R : Début W :Fin	Crée le fichier.

Les fonctions : ouverture d'un fichier

Valeur de retour de `fopen`

- `fopen` renvoie le flux de données associé au fichier ouvert si l'appel a réussi à ouvrir le fichier.
- `fopen` renvoie `NULL` sinon.
- Il faut **toujours tester la valeur de retour de cette fonction**.

Ouverture d'un fichier

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

int main(){
  FILE * f;
  if((f=fopen("test.txt","r"))!=NULL){
    fprintf(stderr, "Le fichier existe\n");
    fclose(f);
    return EXIT_SUCCESS;
  }
  else {
    fprintf(stderr, "Le fichier n'existe pas\n");
    return EXIT_FAILURE;
  }
}
```

Les fonctions : fermeture d'un fichier

La fonction `fclose`

```
int fclose(FILE *fp);
```

- La fonction `fopen` exécute un `malloc` pour créer le tampon/buffer qui va servir à interagir avec le disque dur.
- La fonction `fclose` permet de **libérer ce bloc mémoire** et si besoin, force le système à écrire sur le disque les données du tampon qui n'ont pas encore été transférées.
- Elle renvoie 0 si l'opération s'est déroulée normalement, EOF (souvent -1) dans le cas contraire.

La manipulation de texte en C

Les entrées-sorties formatées : écriture dans un fichier

La fonction `fprintf`

- Fonctionne (presque) comme la fonction `printf` : le premier argument de la fonction est le flux associé au fichier dans lequel on souhaite écrire.
- Son entête est
  ```
  int fprintf(FILE *flux, const char *format, ...)
  ```
- Elle **renvoie le nombre total de caractères écrits dans le tampon** en cas de succès, un nombre négatif en cas d'échec.

Écriture dans un fichier

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

int main(){
  FILE * fp;
  char c;

  fp = fopen ("nouveau.txt", "a");
  fprintf(fp, "%s %s %s %d", "Nous", "sommes", "en", 2021);
  fclose(fp);

  fp = fopen("nouveau.txt","r");
  fscanf(fp,"%c",&c);
  while (!feof(fp)){
    printf("%c", c);
    fscanf(fp,"%c",&c);
  }
  printf("\n");
  fclose(fp);
  return EXIT_SUCCESS;
}
```

Les entrées-sorties formatées : lecture à partir d'un fichier

La fonction `fscanf`

- Fonctionne (presque) comme la fonction `scanf` : le premier argument de la fonction est le fichier dans lequel on souhaite lire.
- Son entête est
  ```
  int fscanf(FILE *flux, const char *format, ...)
  ```
- Elle **renvoie le nombre de valeurs convenablement lues**.

Lecture et écriture dans un fichier

Les limites de l'écriture

- On peut écrire dans le fichier tant qu'il y a de la place sur le disque dur.
- On s'autorise à considérer qu'on ne sera jamais à court de place.

Les limites de la lecture

- On lit dans le fichier jusqu'à ce qu'il soit terminé.
- La fonction `int feof(FILE *f)` renvoie une valeur non nulle si la fin du fichier a été atteinte.

Lecture et écriture dans un fichier

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

int main(){
  FILE * f;
  char test[10];

  if((f=fopen("test.txt","w+"))!=NULL){
    printf("Le fichier existe\n");
    fprintf(f,"Je suis un fichier test\n");
    fclose(f);
  }
  else
    printf("Le fichier n'existe pas\n");

  if((f=fopen("test.txt","r"))!=NULL){
    while(!feof(f)){
      if(fscanf(f,"%s",test)==1)
        printf("%s\n",test);
    }
    fclose(f);
  }
  else
    printf("Le fichier n'existe pas\n");

  return EXIT_SUCCESS;
}
```

Fichiers texte binaire

Les différents types de fichiers

Les fichiers texte

- Les caractères comme le retour à la ligne, la tabulation, etc. y sont interprétés comme tels lors de la lecture et de l'écriture.
- Ils peuvent être lus et modifiés directement par un éditeur.
- Les fonctions d'entrées-sorties (`fprintf` et `fscanf`) **lisent et écrivent des chaînes de caractères formatées**.

Les fichiers binaires

- Les caractères comme le retour à la ligne, la tabulation, etc. n'y sont pas interprétés.
- Il est très compliqué de les lire avec un éditeur sans avoir d'informations complémentaires.
- Les données peuvent y être mieux compressées.
- Des fonctions d'entrées-sorties binaires **lisent ou écrivent le contenu intégral d'un bloc mémoire sans se soucier de son interprétation**.

Fichiers textes

Fichiers texte

- Dans un fichier texte, **chaque octet est interprété comme un caractère ASCII**.

Fonctions spécifiques pour la manipulation des fichiers textes

- Les fonctions `fprintf` et `fscanf`.
- La fonction `int fgetc(FILE *flux)` lit un caractère (`unsigned char`) du flux et le renvoie (converti en `int`) ; elle renvoie EOF si la tête de lecture a atteint la fin du fichier ou en cas d'échec.
- La fonction `int fputc(int car, FILE *flux)` écrit le caractère passé en argument dans le flux et le renvoie ; elle renvoie EOF en cas d'échec de l'écriture.

Les fonctions d'entrée-sortie

`printf` et `fprintf`

- Ces fonctions **écrivent toujours les caractères ASCII des valeurs passées en arguments**..
- Pour les chaînes de caractères, c'est assez intuitif.
- Pour les nombres. . . voyons avec un exemple.

```
int x=5000000;
printf("%d",x);
```

  La fonction va afficher 7 caractères, correspondant aux 7 chiffres.
- Si on effectue la même opération avec `fprintf`
  - on écrira 7 caractères dans un fichier.
  - on écrira donc 7 octets de données.
- Or un `int` occupe 4 octets et peut stocker des valeurs bien plus grandes.

Fichiers texte / fichiers binaires

Format de stockage

Pour les données numériques en particulier :

- on stocke directement la représentation binaire des valeurs
- plutôt que le code ASCII de chaque chiffre.

On obtient un fichier qui occupe moins d'espace en mémoire.