C

Introduction à C

Ce document présente rapidement les bases du langage C, à travers divers exemples, en supposant connues les notions de base de programmation.

Contents

1 Hello, C !

1.1 Un essai

1. Pour commencer, utilisez un *éditeur de textes* pour placer les lignes suivantes dans un fichier source nommé `hello.c` :

```c
// fichier hello.c, premier exemple

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

int main(void)
{
    printf("Hello World!n");
    return EXIT_SUCCESS;
}
```

2. puis lancez la **compilation** de ce fichier source avec la commande "`gcc hello.c`". Si il y a des erreurs, vérifiez soigneusement ce que vous avez tapé, et corrigez.

3. Et enfin, lancez l'**exécution** du fichier `a.out` produit par le compilateur :

```
$ ./a.out
Hello World!
```

Bravo, votre premier programme C est un succès !

1.2 Explications

Le code source mérite quelques explications.

La fonction `main` :

- est par convention le point d'entrée du programme
- est déclarée sans paramètres (`void` entre parenthèses)
- retourne un entier qui indique si les choses se sont bien passées
- la valeur retournée est ici `EXIT_SUCCESS`, une constante déclarée dans le fichier d'entête de la bibliothèque standard `stdlib.h`

La fonction `printf` :

- fait partie des fonctions d'entrées-sorties standards (déclarées dans `stdio.h`)
- accepte un nombre variable de paramètres. Ici, uniquement le message à afficher (`n` représente le caractère "saut de ligne").

Les directives `#include` :

- désignent les fichiers d'entête (headers) contenant les déclarations à importer
- recherchent les fichiers dans les répertoires des **bibliothèques systeme** (en général `/usr/include`), parce qu'ils sont cités entre "<>"

1.3 Options de compilation

La ligne de commande indiquée plus haut ne comporte pas les options indispensables pour travailler sérieusement. À la place, il faut taper :

```
$ gcc -o hello  -std=c11  -Wall -Wextra -pedantic hello.c
```

qui

- fabrique un exécutable nommé `hello` (au lieu de `a.out`)
- se conforme au dernier standard (ISO C11) du langage (sinon, avec le compilateur `gcc` 6.3.0, ca sera `c11`, une version modifiée du standard C11).
- vous signale un maximum d'erreurs, d'avertissements et de conseils.

1.4 Utilisez un `Makefile`

La commande `make` peut vous simplifier énormément la vie. Dans un fichier nommé `Makefile`, mettez les lignes

```
CC     = gcc
CFLAGS = -std=c11 -Wall -Wextra -pedantic

all:   hello
hello: hello.o
```

Désormais, pour (re)-fabriquer l'exécutable `hello`, vous aurez simplement à taper `make`, qui lancera les commandes nécessaires

```
$ make
gcc -std=c11 -Wall -Wextra -pedantic   -c -o hello.o hello.c
gcc   hello.o   -o hello
```

et remarquez que si vous relancez `make`, la commande vous dit qu'il n'y a rien à faire :

```
$ make
make: Nothing to be done for 'all'.
```

En effet, `make` "sait" -parce que vous l'avez indiqué) que `hello` a été fabriqué à partir de `hello.o` lui-même obtenu par compilation de `hello.c` (dépendance implicite). Donc si on n'a pas modifié `hello.c`, l'exécutable est à jour et il n'est pas nécessaire de le reconstruire.

Quand vous ferez des programmes d'une certaine ampleur avec des dizaines de fichiers sources, `make` ne recompilera que ce qui est strictement nécessaire, ce qui vous fera gagner du temps.

Essentiellement, les "gestionnaires de projets" intégrés aux EDI font la même chose.

2 Exemple 2 : les copies

2.1 Objectifs, début du source

Voici le début du source d'un programme C dont l'intention doit être évidente

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

#define MAX_LONGUEUR_NOM 10

struct Copie {
    char nom[MAX_LONGUEUR_NOM];
    float note;
};

// déclarations des fonctions auxiliaires

int  demander_nombre_copies();
void lire_copies(struct Copie tableau[], int nombre);
void afficher_copies(char titre[], struct Copie tableau[], int nombre);

// ------------------------------------------------

int main(void)
{
    int nb_copies = demander_nombre_copies();
    struct Copie copies[nb_copies];

    lire_copies(copies, nb_copies);
    afficher_copies("Les notes reçues", copies, nb_copies);

    // ordonner_copies_par_note(copies, nb_copies);
    // afficher_copies("Notes décroissantes", copies, nb_copies);

    // ordonner_copies_par_nom(copies, nb_copies);
    // afficher_copies("Par ordre alphabétique", copies, nb_copies);

    return EXIT_SUCCESS;
}
```

C'est un programme interactif qui va

- lire une série de noms et de notes,
- les faire afficher dans l'ordre de saisie
- puis (quand la fonction sera écrite) dans l'ordre des notes
- puis dans l'ordre alphabétique.

Remarquez les **éléments nouveaux** dans ce programme :

- la ligne `#define` qui définit une constante. C'est une **constante de macro-processeur** : partout où apparaitra le texte `MAX_LONGUEUR_NOM` dans le source, il sera *remplacé* par la chaine `10`.
- La définition d'un *type de données* : `struct Copie`. Les instances seront des entités avec deux attributs ("membres" ou "champs", dans la terminologie C), le premier étant un tableau de 10 octets et le second un nombre flottant.
- Viennent ensuite les **déclarations des fonctions** qui sont utilisées dans la suite. Une fonction doit avoir été déclarée (dans le source ou dans un header) avant d'être employée. Une déclaration ne comporte que le **prototype** de la fonction, c'est à dire le type de retour, le nom et la liste de paramètres, suivi d'un point-virgule.
- Ne pas confondre la déclaration et la **définition des fonctions**, qui sera faite plus loin.
- le premier paramètre de `afficher_copies` est un tableau d'octets, de taille non spécifiée. Nous verrons plus loin que ça correspond à une chaine de caractères.
- le second paramètre est un *tableau* de `struct Copie`s. Le troisième paramètre indique combien il y a de `struct Copie` à considérer à cet endroit.

2.2 Fonctionnement du `main`

Quand le `main` est lancé,

1. la fonction `demander_nombre_copies` est exécutée et retourne un nombre entier (par exemple 8)
2. La définition de la variable `copie` réserve alors, dans la pile d'exécution, une zone pour contenir 8 instances de `struct Copie`, soit 8 x (10 + 4) = 112 octets consécutifs, sur les machines où les `float` occupent 32 bits = 4 octets.

Notez bien la différence avec Java, où une déclaration `Copie[] copies` ne ferait que déclarer une référence, et l'instruction `copies = new Copie[8];` réserverait un tableau de 8 *réferences*, mais pas les instances elles-mêmes. En C, les instances sont des parties du tableau, à qui de l'espace est alloué dès sa déclaration.

3. la fonction `lire_copies` est appelée, avec comme paramètres le tableau et un entier. Attention, le tableau n'est pas un conteneur, c'est une indication de l'*emplacement* où on trouve des données. Ce qu'on fournit à `lire_copie`, c'est la valeur de l'**adresse** en mémoire où la lecture doit ranger les données.
4. Ensuite appel de `afficher_copies`, avec comme premier paramètre un tableau d'octets. là aussi, ce qui est transmis, c'est l'adresse du premier octet du message.
5. Et pour finir, retour au "runtime" qui a lancé la fonction `main`.

Important

- En C, un appel de fonction passe toujours des **valeurs** en paramètres
- quand on passe un tableau, ce qui est transmis c'est la **valeur de l'adresse** de son premier élément

2.3 La fonction `demander_nombre_copies`

```c
int demander_nombre_copies()
{
    int nombre_copies;
    do {
        printf("nombre de copies : ");
        scanf("%d", & nombre_copies);
    } while (nombre_copies <= 0);
    return nombre_copies;
}
```

Ici, le point notable est la lecture d'un entier en utilisant la fonction standard `scanf`.

- le premier paramètre est une "chaine de spécification de format", qui indique ici (`%d`) qu'on veut lire la représentation décimale d'un entier.
- le second paramètre indique l'endroit où il faudra ranger cet entier. Il s'agit donc d'indiquer **l'adresse** de la variable `nombre_copies`. D'où la présence de l'opérateur "`&`" (*address of*).

Très important : rappelez-vous qu'en C, le passage de paramètre se fait par **valeur**. Si on écrivait

```c
scanf("%d",  nombre_copies);   // ne faites surtout pas ça.
```

la fonction `scanf` recevrait la *valeur actuelle de l'entier* contenu dans `nombre_copies`. Ca ne fait pas l'affaire du tout. Ce qu'on veut, c'est dire à `scanf` **où** elle doit mettre la valeur lue.

Attention cette fonction n'est pas protégée contre les erreurs de saisie. Rien n'est prévu pour détecter le cas où on taperait des caractères non numériques, ou des valeurs anormalement grandes.

2.4 La fonction `lire_copies`

Une surprise : la lecture du code suivant va vous étonner :

```
void lire_copies(struct Copie tableau[], int nombre_copies)
{
    for (int i = 0; i < nombre_copies; i ++) {
        printf("Copie %d/%dn- nom  : ", i+1, nombre_copies);
        scanf("%s", tableau[i].nom);
        printf("- note : ", );
        scanf("%f", & tableau[i].note);
    }
}
```

Le problème : On vient d'expliquer que pour lire un entier, on mettait un "&" pour transmettre à `scanf` l'adresse de la destination. C'est ce qu'on fait encore dans

```
    scanf("%f", & tableau[i].note);
```

mais alors pourquoi fait-on ainsi la lecture du nom ?

```
        scanf("%s", tableau[i].nom);      // pas de & ???
```

Explication :

- d'abord remarquer qu'il s'agit de **lire une chaine** de caractères (`%s` = string) que l'on va placer dans le champ `nom` de `tableau[i]`
- le second paramètre doit donc être l'adresse où on veut mettre la chaîne ;
- mais justement, `nom` est un *tableau* de d'octets, `tableau[i].nom` c'est l'adresse du premier octet de la copie numéro i.

Lors de la lecture d'une chaine, les caractères tapés par l'utilisateur sont placés dans le tableau d'octets indiqué. A la fin de la ligne, ou quand le "mot" lu est terminé, un caractère nul est ajouté pour marquer la fin de la chaine.

Autrement dit, une chaine de 3 caractères "Isa" occupera les 4 premiers octets dans le tableau `nom`, puis qu'une place est réservée pour la *valeur sentinelle* (`'0'`).

Il est clair qu'on aura des problèmes si on tape une chaine de plus de 9 octets : ça ne rentre pas, et ce qui est tapé en trop va déborder sur ce qui est situé plus loin dans la mémoire. Là aussi, il faudrait mettre en place un mimimum de protections.

A noter aussi la présence d'une chaine de spécification de format un peu plus élaborée dans

```
printf("Copie %d/%dn- nom  : ", i+1, nombre_copies);
```

C'est simple, les représentations décimales de `i+1` et `nombre_copies` vont prendre la place des "`%d`". Si par exemple `i` et `nombre_copies` valent respectivement 3 et 8, il s'affichera

```
Copie 4/8
- nom
```

La chaine de spécification de format est là pour donner un "modèle de présentation" du résultat, avec des "%" pour indiquer les "trous" où apparaitront les valeurs.

Attention la lecture par `scanf("%s", ...)` ne va fonctionner correctement que pour lire un "mot", et à condition qu'il ne soit pas trop grand (la taille du champ `nom` est limitée). Faire des lectures "blindées" en C est assez pénible, on s'en dispense dans ces exemples.

2.5 La fonction `afficher_copies`

Cette fois-ci pas de nouveauté fondamentale dans ce code

```
void afficher_copies(char titre[], struct Copie tableau[], int nombre)
{
    printf("n*** %s ***nn", titre);
    char tirets[] = "+------------+-------+n";
    printf(tirets);
    for (int i = 0; i < nombre; i++) {
        printf("| %-10s | %5.2f |n",  tableau[i].nom, tableau[i].note);
    }
    printf(tirets);
    printf("n");
}
```

qui comporte des options dans les spécifications de format

- "`%-10s`" pour une chaine (%s) complétée à 10 octets par des espaces, et cadrée à gauche (-).
- "`%5.2f`" pour un flottant représenté sur 5 positions, dont 2 après la virgule.

Exemple d'exécution

```
$ ./main
nombre de copies : 3
Copie 1/3
- nom  : Arthur
- note : 12.5
Copie 2/3
- nom  : Charles
- note : 7.75
Copie 3/3
- nom  : Bridget
- note : 18

*** Les notes reçues ***

+------------+-------+
| Arthur     | 12.50 |
| Charles    |  7.75 |
| Bridget    | 18.00 |
+------------+-------+
```

Attention là aussi : la spécification de largeur de format (`%-10s`) ne fonctionne correctement qu'avec les données en code ASCII et ISO-8859. Un caractère sur un octet, c'était le bon temps, mais c'était avant.

De nos jours, tout ceci déraille quand - situation fréquente - le terminal communique en UTF-8, ce qui est une situation fréquente maintenante, parce qu'un caractère y est codé par une séquence de 1 à 4 octets. À noter que Java gère très bien le problème, en établissant une séparation claire entre les 2 niveaux "fichiers en tant que suites d'octets" (`File`) et "lecteurs/ecrivains de suites de caractères" (`Scanner`/`PrintStream`).

En C, il faut alors rentrer dans les complications des "wide chars" (exemple fonction `wprintf`) qui sortent largement du cadre de cet exposé sur les bases de C.

2.6 Le tri par notes

En décommentant quelques lignes, on peut ajouter le classement des copies par notes décroissantes, par exemple avec un algorithme classique de tri par insertion

```
void ordonner_copies_par_note(struct Copie tableau[], int nombre_copies)
{
  /* tri par insertion */
  for (int i = 1; i < nombre_copies; i++) {
    // les élsments d'indice 0 à i-1 sont ordonnés,
    // on va y insérer tableau[i]

    // sauver
    struct Copie copie = tableau[i];
    // décaler les éléments précédents qui sont plus petits
    int j = i;
    while (   j > 0
           && tableau [j-1].note < copie.note) {   // comparaison
      tableau[j] = tableau[j-1];
      j--;
    }
    tableau[j] = copie;
  }
}
```

Sur le plan du langage, pas grand chose à en dire, sinon qu'on peut peut y repérer trois fois l'affectation d'une **structure** dans une autre de même type. Ceci a pour effet de copier les contenus des structures.

En Java, si on écrivait

```
Copie copie = tableau[i];
```

cela mettrait dans `copie` une référence à la même instance que `tableau[i]`, il n'y aurait pas de duplication de l'information. C'est très différent.

2.7 Le tri par ordre alphabétique

Pour ordonner les copies par nom, il faut remplacer la comparaison des notes

```
while (j > 0
    &&  tableau [j-1].note < copie.note ) {
...
}
```

par une comparaison des noms, qui sont des chaines de caractères. Pour cela, on utilise, la fonction `strcmp` (string compare) définie dans `string.h` (à ajouter dans les `#include`).

Ses deux paramètres sont les adresses des débuts des chaines à comparer, et elle retourne un entier qui 0 si les chaines sont identiques, ou positif (resp. négatif) si la première chaine est après (resp. avant) la seconde dans l'ordre lexicographique. Bref, on écrit

```
while (j > 0
  && strcmp(tableau [j-1].nom, copie.nom) > 0) {
  ...
}
```

Si tout se passe bien, on voit la liste dans le bon ordre

```
*** Par ordre alphabétique ***

+------------+-------+
| Arthur     |  9.75 |
| Bernard    | 14.50 |
| Charles    | 16.00 |
| Dora       | 10.00 |
| Eliza      | 17.25 |
+------------+-------+
```

3 Exemple 3 : les feuilles

Il est assez raisonnable d'écrire

```
#define MAX_COPIES_PAR_FEUILLE 20

struct Feuille {
  int nombre_copies;
  struct Copie tableau[MAX_COPIES_PAR_FEUILLE];
};
```

pour former une entité unique contenant à la fois le nombre de copies et leurs descriptions.

Une "`struct Feuille`" contiendra donc 20 `struct Copie` dont un certain nombre (indiqué par le champ `nombre_copies` seront effectivement occupées par des données).

Ceci nous permettra de définir des actions `lire_feuille`, `ordonner_feuille`, … auxquelles on passera un seul paramètre représentant la feuille de notes.

3.1 Le problème

Voila ce que donnerait, en gros, une version naïve de la fonction d'affichage d'une `struct Feuille`

```
void afficher_feuille_betement(char titre[],
                               struct Feuille feuille)
{
    ...
    for (int i = 0; i < feuille.nombre_copies; i++) {
        printf("| %-10s | %5.2f |n",
               feuille.tableau[i].nom,
               feuille.tableau[i].note);
    }
    ....
}
```

Le souci, c'est qu'appeler cette fonction gaspillerait du temps, parce que le passage d'une structure en paramètre transmet en fait une copie. Et fait la copie, ça prend du temps et de la place.

Or si on programme en C, qui n'est pas un langage de très haut niveau, c'est bien pour éviter les pertes de temps et d'espace pendant l'exécution.

Il y a pire : une fonction de lecture de Feuille comme celle-ci

```
void lire_feuille_stupidement(struct Feuille feuille) {
    scanf("%d",  & feuille.nombre_copies);
    ...
}
```

ne fonctionnerait pas du tout. En effet, les `scanf` agiraient sur la copie qui a été faite au moment de l'appel, pas du tout sur la `struct Feuille` dans laquelle on voulait lire

3.2 Solution : pointeurs contenant des adresses

La solution est de passer en paramètre **l'adresse de la structure** à lire/écrire, plutot que sa valeur actuelle. Le `main` ressemblerait alors à ceci

```
int main(void)
{
  struct Feuille feuille;

  lire_feuille(& feuille);                          // adresse de feuille
  afficher_feuille("Les notes reçues", & feuille);
  ...
}
```

Pour cela, il faut déclarer les fonctions en indiquant que le paramêtre est une **adresse** de structure, ce qui se fait ainsi, avec une étoile devant le nom de la variable qui contient l'adresse:

```
void lire_feuille(                  struct Feuille *adr_feuille);
void afficher_feuille(char titre[], struct Feuille *adr_feuille);
```

Remarques:

- une variable comme `adr_feuille`, qui contient une adresse de `Feuille`, est appelée un **pointeur de Feuille**
- la déclaration `struct Feuille *adr_feuille` se lit de la façon suivante

  `*adr_feuille` est une `struct Feuille`

En effet, si `p` est un pointeur, `*p` est la donnée "pointée" par `p`.

Il suffit de reprendre la version "naïve" et de remplacer les occurrences du paramètre `feuille` par `*adr_feuille`, quitte à ajouter quelques parenthèses pour les histoires de priorités d'opérateurs.

```
void afficher_feuille(char titre[],
                      struct Feuille *adr_feuille)
{
    ...
    for (int i = 0; i < (*adr_feuille).nombre_copies; i++) {
        printf("| %-10s | %5.2f |n",
               (*adr_feuille).tableau[i].nom,   // voir meilleure notation plus loin
               (*adr_feuille).tableau[i].note);
    }
    ....
}
```

3.3 La notation “flèche”

Qui plus est, il existe une notation “flèche” (`->`) qui facilite le codage. Si `p` est un pointeur sur une structure qui a un champ `x`, on peut écrire `p->x` au lieu de `(*p).x`.

Voila ce que ça donne :

```
void afficher_feuille(char titre[],
                      struct Feuille *adr_feuille)
{
    ...
    for (int i = 0; i < adr_feuille->nombre_copies; i++) {
        printf("| %-10s | %5.2f |n",
               adr_feuille->tableau[i].nom,
               adr_feuille->tableau[i].note);
    }
    ....
}
```

C’est plus “naturel” parce que `adr_feuille->tableau[i].note` se comprend comme suit

- `adr_feuille` pointe sur un truc (suivez la flèche)
- on regarde son champ `tableau`
- plus particulièrement la case d’indice `i`
- et on considère son champ `note`.

3.4 Exercices

1. Compléter `afficher_feuille`
2. Écrire `lire_feuille` qui

- reçoit comme paramètre une feuille à remplir
- demandera un entier, le nombre de copies entre 0 et `MAX_COPIES_PAR_FEUILLE`
- remplira la feuille à partir de données demandées à l’utilisateur

3. Écrire les fonctions `ordonner_feuille_par_note` et `ordonner_feuille_par_nom`.

3.5 Résumé

Nous venons de voir

- la notion de **pointeur**, dont les débutants se font toute une histoire, mais qui est bêtement une variable destinée à contenir l’**adresse**, la position en mémoire, d’une donnée.
- la **déclaration d’un pointeur**, qui a la forme `T *p`, où `T` est le type de la donnée pointée par `p`.
- le “**déréférencement d’un pointeur** `p`”, c’est à dire l’action de regarder le contenu de la donnée à l’emplacement qu’il “pointe”. Avec deux notations, l’une pour désigner la donnée entière (`*p`), l’autre pour un de ses membres (`p->x`).

4 Retour sur le tri (passages de fonctions en paramètres)

Dans le code, nous avons deux fonctions qui trient une `Feuille`. Elles utilisent le même algorithme de tri par insertion, tout ce qui diffère, c’est le critère de comparaison entre les `Note`, qui n’est qu’un élément dans une condition d’une boucle du tri

4.1 Objectif : Un tri paramétré par la comparaison

Il est donc raisonnable d'envisager de définir une fonction de tri générale (`ordonner_feuille`) qui prenne comme paramètre l'adresse du code de comparaison.

On pourrait donc écrire

```
ordonner_feuille(& feuille, & comparaison_par_nom_croissant);
afficher_feuille("Par ordre alphabétique", &feuille);

ordonner_feuille(& feuille, & comparaison_par_note_decroissant);
afficher_feuille("Par ordre de mérite", &feuille);
```

4.2 Les fonctions de comparaison

Le but d'une comparaison, c'est de prendre deux Copies, et de dire comment elles sont ordonnées. On suit le même principe que pour `strcmp` : on retourne un entier négatif, nul ou positif qui indique l'ordre.

Pour éviter de faire des copies inutiles, les fonctions de comparaison prendront en paramètre les *adresses* des structures à comparer.

Voici une des fonctions :

```
int comparaison_par_nom_croissant(struct Copie *p1, struct Copie *p2)
{
  return strcmp(p1->nom, p2->nom);
}
```

La fonction étant très courte, et son rôle évident, on s'est permis des abréviations radicales pour les noms des paramètres : `p1` est un pointeur sur la première `struct Copie`. Si vous préférerez `adr_premiere_copie`, c'est vous qui voyez.

Exercice : écrivez `comparaison_par_note_decroissante` qui est encore plus simple (rappelez-vous vos cours de philo : *comparer, c'est étudier les différences*).

4.3 La fonction de tri

Comme vu plus haut, la fonction de tri sera appelée avec l'adresse d'une fonction en paramètre.

```
ordonner_feuille(& feuille, & comparaison_par_nom_croissant);
```

Il nous reste à voir deux choses :

- dans le prototype de `ordonner_feuille`, comment déclarer le second paramètre

```
void ordonner_feuille(struct Feuille *feuille,
                      ..... comparaison....);
```

- dans le tri, comment faire appel à `comparaison`.

1. La déclaration est assez logique : puisque la variable `comparaison` va recevoir l'adresse d'une fonction de comparaison comme

```
int comparaison_par_nom_croissant(struct Copie *p1, struct Copie *p2)
```

on la déclare en remplaçant le nom de cette fonction particulière par `(*comparaison)`, ce qui donne

```
void ordonner_feuille(struct Feuille *feuille,
                      int (*comparaison)(struct Copie *p1, struct Copie *p2));
```

Ce n'est pas léger, mais au moins c'est logique.

2. Aucune difficulté pour l'appel : le pointeur de fonction s'utilise comme une fonction :

```
  while (   j > 0
         && comparaison(& tableau [j-1], & copie) {
    ...
```

4.4 Compléments

1. Lorsqu'on passe une fonction en paramètre, on peut omettre le "`&`" : le nom de la fonction représente aussi son adresse.

```
ordonner_feuille(& feuille, & comparaison_par_nom_croissant);
ordonner_feuille(& feuille,   comparaison_par_nom_croissant);
```

(les deux formes sont permises pour des raisons qui remontent aux début de C, quand le langage n'était pas normalisé. On a choisi d'exposer la forme "avec &" parce qu'elle permet de comprendre la logique de déclaration d'une fonction en paramètre).

2. Dans la déclaration d'un type de fonction, on peut omettre les noms des arguments.

```
void ordonner_feuille(struct Feuille *feuille,
                      int (*comparaison)(struct Copie *p1, struct Copie *p2));
void ordonner_feuille(struct Feuille *feuille,
                      int (*comparaison)(struct Copie *  , struct Copie *  ));
```

4.5 `typedef` pour simplifier les déclarations

La directive `typedef` de C permet de définir des **types** en faisant comme si on déclarait des variables. Ca va simplifier les écritures :

Exemple :

```
        int m      [3][3];  // déclaration d'une matrice de 3 x 3 entiers
typedef int Matrice[3][3];  // déclaration d'un type Matrice

Matrice rotation;           // une variable de type Matrice (= tableau 3x3).
```

Appliquons ceci aux comparaisons :

- le type "`ComparaisonCopies`" est défini par

```
typedef  int (*`ComparaisonCopies`)(struct Copie *  , struct Copie *  ));
```

- et la fonction de tri est déclarée :

```
void ordonner_feuille(struct Feuille *feuille, ComparaisonCopies comparaison);
```

5 Réorganisation du code : compilation séparée

Le programme commmence à prendre de l'ampleur. Il est temps de le réorganiser un peu, avant que ça devienne ingérable.

On va le décomposer en 3 "modules"

- ce qui concerne spécifiquement les `Copies` (déclarations et code),
- ce qui concerne les `Feuilles`,
- le programme principal,

que l'on pourra manipuler séparément.

5.1 Le programme principal

Commençons par le `main`. Ce qui nous intéresse c'est d'y trouver

```
int main(void)
{
  struct Feuille feuille;

  lire_feuille (& feuille);
  ...
  return EXIT_SUCCESS;
}
```

Il va donc falloir y "importer"

- les déclarations concernant les Feuilles (type de donnée et prototypes des fonctions),
- la constante `EXIT_SUCCESS`.

Dans le source `main.c` on met donc

```
#include <stdlib.h>
#include "Feuille.h"

int main(void)
{
  struct Feuille feuille;

  lire_feuille (& feuille);
  afficher_feuille("Les notes reçues", &feuille);
  ...
  return EXIT_SUCCESS;
}
```

5.2 Feuille.h

Dans ce fichier d'entête on doit trouver les déclarations

- de la structure `Feuille`
- des fonctions qui vont avec.

```
#define MAX_COPIES_PAR_FEUILLE 20

struct Feuille {
  int nombre_copies;
  struct Copie tableau[MAX_COPIES_PAR_FEUILLE];
};

void lire_feuille (struct Feuille *adr_feuille);
...
```

Comme la déclaration de la structure `Feuille` nécessite de connaitre la structure `Copie`, il faut aussi inclure `Copie.h`.

Voici le source :

```
#ifndef FEUILLE_H
#define FEUILLE_H

#include "Copie.h"

#define MAX_COPIES_PAR_FEUILLE 20

struct Feuille {
  ...
};

void lire_feuille                  (struct Feuille *adr_feuille);
...
}

#endif
```

Le rôle des 2 premières lignes et de la dernière sera expliqué plus loin.

5.3 Le fichier `Copie.h`

De la même façon, dans `Copie.h`, on trouve

```
#ifndef COPIE_H
#define COPIE_H

#define MAX_LONGUEUR_NOM 10

struct Copie {
  char nom[MAX_LONGUEUR_NOM];
  float note;
};

typedef int (*ComparaisonCopies)(struct Copie *p1, struct Copie *p2);

int comparaison_par_nom_croissant(struct Copie *p1, struct Copie *p2);
int comparaison_par_note_decroissante(struct Copie *p1, struct Copie *p2);

#endif
```

5.4 Copie.c : le code pour les copies

Maintenant, il nous faut un fichier `Copie.c` pour le code des fonctions sur les copies. Ce code s'appuie sur la définition du type `struct Copie`, qu'il faut inclure. Et on utilise aussi `strcmp` :

```
#include <string.h>

#include "Copie.h"

int comparaison_par_nom_croissant(struct Copie *p1, struct Copie *p2) {
  return strcmp(p1->nom, p2->nom);
}

...
```

5.5 Feuille.c

Et enfin, `Feuille.c` contient les définitions de fonctions sur les Feuilles, et inclue donc les déclarations nécessaires au code

```
#include <stdio.h>

#include "Copie.h"
#include "Feuille.h"

void lire_feuille(struct Feuille * adr_feuille)
{
  int nb;
  ...
}

void afficher_feuille(char titre[], struct Feuille *adr_feuille)
{
    printf("*** %s ***nn", titre);
    ...
}

void ordonner_feuille(struct Feuille *adr_feuille,
              ComparaisonCopies comparaison)
{
  /* tri par insertion */
 for (int i = 1; i < adr_feuille->nombre_copies; i++) {
 ...
}
```

Remarquez qu'on a inclu ici le fichier `Copie.h`, ce qui n'est pas indispensable, puisqu'il est lui-même inclus dans `Feuille.h`.

Sans précautions, ça poserait un problème à la compilation parce que le type `struct Copie` serait défini deux fois.

Heureusement on a eu la bonne idée de prendre les précautions standards contre les inclusions multiples.

5.6 Le préprocesseur de C

Pour comprendre ces précautions, il faut savoir que la compilation d'un fichier source se fait en deux temps

1. un programme utilitaire, le **préprocesseur** `cpp`, s'occupe d'abord de toutes les directives qui commencent par le caractère dièse : `#include`, `#define`, `#ifdef`, … et fabrique un nouveau texte source.
2. ensuite, le texte résultant est analysé et traduit.

5.6.1 Les directives

Dans la première phase,

- les directives `#include` sont remplacées par le texte contenu dans le fichier indiqué
- les définitions `#define` sont notées dans la "table des variables du préprocesseur" (exemple MAX_COPIES_PAR_FEUILLE)
- quand ces variables sont rencontrées dans le texte, elles sont remplacées par la chaine correspondante.

C'est ce qu'on appelle la phase d'**expansion**, qui comporte aussi des directives `#ifdef`, `#endif` pour faire de l'expansion conditionnelle.

5.6.2 Exemple

Regardons maintenant ce qui se passe pendant l'expansion de `Feuille.c`, qui commence par

```
#include <stdio.h>    // 1

#include "Copie.h"    // 2
#include "Feuille.h"  // 3
```

1. Tout d'abord, le contenu de `/usr/include/stdio.h` est inséré à la place de la ligne 1
2. Puis le préprocesseur va lire le fichier `Copie.h`. Il y trouve ceci :

```
#ifndef COPIE_H
#define COPIE_H

#define MAX_LONGUEUR_NOM 10

struct Copie {
  ...
};
...

#endif
```

3. la variable `COPIE_H` du préprocesseur n'étant pas encore connue le`#ifndef` (if not defined) réussit. La partie qui va jusqu'au `#endif` est donc traitée.

4. Ceci provoque la déclaration de la variable de préprocesseur `COPIE_H` (à qui est affectée une chaine vide) ainsi que de `MAX_LONGUEUR_NOM` (contenant `10`). Et bien sûr l'inclusion de la déclaration `struct Copie`, etc

5. A la fin de ce fichier, le préprocesseur revient donc s'occuper de la ligne 3 de `Feuille.c`. Il y trouve une directive `#ifndef FEUILLE_H` qui est franchie avec succès parce que `FEUILLE_H` n'est pas encore définie, ce qui est fait juste après.

6. Le préprocesseur retrouve un `#include "Copie.h"`. Ce fichier est re-consulté. Mais cette fois si la variable `COPIE_H` est définie, donc la partie entre le `#ifndef` et le `#endif` est ignorée, ce qui évite une double définition.

5.6.3 Précaution systématique

On utilisera systématiquement cette précaution pour tout fichier “toto.h”, en encadrant son contenu par les lignes

```
#ifndef TOTO_H
#define TOTO_H
....
#endif
```

Le nom de la “variable de garde” étant dérivée du nom du fichier (variantes `TOTO_H_INCLUDED`, `MODULE3_TOTO_H`, etc.)

5.7 Compiler les modules

Nous nous retrouvons maintenant avec 5 fichiers, correspondant à 3 modules : le programme principal, et les modules des feuilles, et des copies :

principal	feuilles	copies
`main.c`	`Feuille.c`	`Copie.c`
	`Feuille.h`	`Copie.h`

à partir desquels on veut fabriquer un exécutable `main`.

5.7.1 Compilation monolithique

Une première manière de faire est de compiler d'un seul coup tous les sources C, avec (les options sont omises pour faciliter la lecture) :

```
$ gcc  main.c Feuille.c Copie.c  -o main
```

Vous imaginez que ça va faire perdre beaucoup de temps sur un projet qui comportera des dizaines de modules, si on doit tout recompiler chaque fois qu'on change une ligne dans un des fichiers.

5.7.2 Compilation séparée

Pour gagner du temps, on compile séparement les sources (option `-c`)

```
$ gcc -c main.c
$ gcc -c Feuille.c
$ gcc -c Copie.c
```

ce qui produit un “module objet” (portant le suffixe `.o`) pour chacun d'eux.

Ensuite, on réunit ces modules (phase dite “d'édition des liens”) et la bibilothèque standard C pour fabriquer l'exécutable

```
$ gcc   main.o Feuille.o Copie.o   -o main
```

Il y a un gros avantage : si on a modifié seulement le fichier `main.c` par exemple, pour obtenir le nouvel exécutable il suffit de recompiler `main.c`, et de refaire l'édition des liens.

```
$ gcc -c main.c
$ gcc   main.o Feuille.o Copie.o   -o main
```

5.7.3 Makefile, gestion des dépendances

Un `Makefile` qui s'occuperait de ce "projet" commencerait par les lignes

```
CC     = gcc
CFLAGS = -std=c11 -Wall -Wextra -pedantic

all:   main
main:  main.o Copie.o Feuille.o
```

- La dernière ligne indique que `main` s'obtient à partir des trois "modules objets" ;
- la commande `make` voit que l'exécutable `main` doit être produit à partir de `main.o` et d'autres fichiers. Elle en déduit qu'il faut faire une édition des liens de programmes C (il y a des règles de fabrication implicites) ;
- de même, comme on lui demande de fabriquer `main.o` ET qu'il existe un fichier `main.c` dans le répertoire, il fait appel au compilateur C (avec les options de `CFLAGS`). Et pareillement pour les autres modules.

Par contre, il y a des dépendances qui ne sont pas prises en compte automatiquement. Si on modifie le fichier d'entête `Feuille.h`, il faudra recompiler `Feuille.c` (qui en fait l'inclusion) ainsi que `main.c`. Autrement dit, les modules `main.o` et `Feuille.o` **dépendent** de l'entête `Feuille.h`.

On ajoute donc, à la fin du `Makefile`, des règles pour les "dépendances supplémentaires" (la dépendance de chaque module envers son source est implicite):

```
# dépendances supplémentaires

main.o    :  Feuille.h Copie.h
Feuille.o :  Feuille.h Copie.h
Copie.o   :  Copie.h
```

Ce qui garantit que le lancement de "`make`" déclenche uniquement les compilations strictement nécessaires.

Note : il est possible de générer automatiquement ces dépendances plutot que de les écrire à la main, mais c'est un peu trop long pour l'expliquer ici.

6 Exemple 3 : feuilles de taille illimitée (allocation dynamique)

On souhaite maintenant disposer de Feuilles avec un nombre arbitraire de Copies, ans être limité par une une constante `MAX_COPIES_PAR_FEUILLE` qui conduit à du gaspillage.

En effet, si on donne la valeur 100 à cette constante, **toutes** les feuilles occuperont l'espace d'une feuille de 100 notes. Y compris les feuilles qui ont peu de note.

6.1 La nouvelle `struct Feuille`

On va donc procéder autrement : une `struct Feuille` ne contiendra plus directement un tableau de Copies, mais un **pointeur** vers une zone qui contiendra le tableau.

```
struct Feuille {
    int nb_copies;
    struct Copie *tableau;          // modification
};
```

Remarque : la taille de la structure `Feuille` est maintenant la taille d'un entier + la taille d'un pointeur, soit 8 ou 16 octets seulement sur les machines 32-bits ou 64-bits. Mais c'est parce que les informations des Copies n'y sont pas stockées directement : on a seulement l'*emplacement* où elles se trouvent.

6.2 Allocation dynamique

On utilise la fonction `malloc` (memory allocation, de la bibliothèque `memory.h`), pour **allouer dynamiquement** (réserver) cette zone.

La séquence suivante initialise une feuille de 20 copies :

```
struct Feuille f;
f.nb_copies = 20;
f.tableau = malloc(n * sizeof(struct Copie));
```

Le paramètre est le nombre d'octets à réserver : ici c'est le nombre de copies multiplié par la taille (en nombre d'octets) d'une `struct Copie`.

Le résultat de `malloc` est un **pointeur non typé** (de type `void*`), qui est converti implicitement en `struct Copie *` par l'affectation dans le champ `tableau`. Il indique où se trouve le premier octet de la zone, qui correspond à l'adresse du premier élément du tableau.

Attention, l'allocation peut retourner `NULL` en cas d'échec (allocation impossible parce que toute la mémoire est occupée). En principe, il faut le vérifier.

6.3 Modularité

6.3.1 Fonction d'initialisation

De cette séquence, on peut faire une fonction qui initialise une feuille (dont l'adresse est reçue en paramètre) pour qu'elle contiennent un certain nombre de `Copies` :

```
void initialiser_feuille(struct Feuille * adr_feuille, int nb)
{
    adr_feuille->nb_copies = nb;
    adr_feuille->copies    = malloc(nb * sizeof(struct Copie));
}
```

Usage :

```
struct Feuille maths;
initialiser_feuille(& maths, 20);
```

6.3.2 Libération

Les zones qui ont été allouées dynamiquement doivent être libérées par un appel à `free` quand on n'en a plus besoin, sinon on court le risque d'une **fuite mémoire**.

On peut définir une autre fonction pour s'en occuper

```
void liberer_feuille(struct Feuille * adr_feuille)
{
    free(feuille->copies);
    feuille->copies = NULL;  // précaution
}
```

Un petite précaution a été prise parce que libérer plusieurs fois la même zone provoque une erreur. On passe donc l'adresse à `NULL` lors du premier appel à `libérer_feuille`. Les appels suivant feront des `free(NULL)`, ce qui est simplement sans effet.

6.4 Les autres fonctions

Les autres fonctions, par exemple l'affichage sont inchangées

```
for (int i = 0; i < adr_feuille->nombre_copies; i++) {
    printf("| %-10s | %5.2f |n",
           adr_feuille->tableau[i].nom,
           adr_feuille->tableau[i].note);
}
```

parce que la notation "[…]" est utilisable avec les pointeurs autant qu'avec les tableaux ;

- `adr_feuille->tableau` est l'adresse de l'emplacement du début du tableau
- `adr_feuille->tableau[0]` est la première `struct Copie` du tableau
- `adr_feuille->tableau[i]` est la `struct Copie` d'indice i (la i+1 ième).