

5. Allocation dynamique

- 5.1 Pointeurs
- 5.2 Passage par adresse
- 5.3 Allocation dynamique
- 5.4 Tableaux dynamiques

Tableaux dynamiques

Un tableau dynamique de `N` valeurs de type `T` est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

– Exemple –

```
int *tab;  
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

On **lit** et **écrit dans un tableau dynamique** de la même manière que dans un tableau statique. En effet, `tab` pointe vers la première case du tableau, et pour tout $0 \leq i < 66$, `(tab + i)` pointe vers la case d'indice `i`.

La fonction `free` vue précédemment permet de désallouer un tableau. On utilise donc

```
free(tab);
```

pour libérer la zone mémoire occupée par `tab` (c.-à-d. les `66` entiers situés à partir de l'adresse `tab`).

Création d'un tableau dynamique à deux dimensions

Un tableau à deux dimensions est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**. Ceci se généralise dans le cas des tableaux à plus de deux dimensions.

– Exemple –

Les instructions ci-contre permettent de créer un tableau dynamique à deux dimensions de taille 4×3 de valeurs de type `int` initialisées à 0.

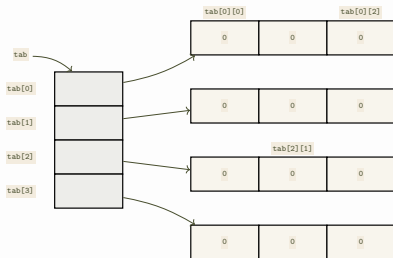
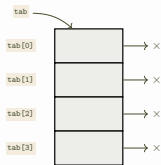
```
int i, j, **tab;
tab = (int **) malloc(sizeof(int *) * 4);
if (tab == NULL) exit(EXIT_FAILURE);
for (i = 0 ; i < 4 ; ++i) {
    tab[i] = (int *) malloc(sizeof(int) * 3);
    if (tab[i] == NULL) exit(EXIT_FAILURE);
    for (j = 0 ; j < 3 ; ++j)
        tab[i][j] = 0;
}
```

Cet exemple est à bien connaître et à généraliser aux dimensions supérieures.

Important : ici, des `exit(EXIT_FAILURE)` sont placés pour quitter l'exécution en cas de problème. Ceci est correct uniquement si ces instructions sont dans le `main`. Si cette création est dans une fonction, un code d'erreur sera renvoyé sans interrompre l'exécution.

Création d'un tableau dynamique à deux dimensions

tab → ×



Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.

Juste après la 1^{re} allocation dynamique, `tab` est dans cet état. L'espace de mémoire de 4 pointeurs sur des entiers a été réservé.

Après les allocations dynamiques des tableaux à une dimension de taille 3 et des initialisations des cases d'entiers, `tab` est dans cet état.

Destruction d'un tableau dynamique à deux dimensions

Pour **libérer** l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

– Exemple –

Les instructions ci-contre permettent de libérer l'espace mémoire occupé par un tableau `tab` à deux dimensions de taille $4 \times N$ de valeurs de type `T` où `N` est un entier strictement positif quelconque et `T` est un type.

```
for (i = 0 ; i < 4 ; ++i) {  
    free(tab[i]);  
    tab[i] = NULL;  
}  
free(tab);  
tab = NULL;
```

Remarque : la connaissance de la seconde dimension du tableau (`N`) n'est pas utile et n'intervient pas dans la suite d'instructions ci-dessus.

La fonction `realloc`

Il est possible de **modifier la taille** d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, int size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

Attention : l'adresse de la zone de la mémoire réallouée peut être différente de son adresse d'origine.

– Exemple –

```
int *tab;  
tab = (int *) malloc(sizeof(int) * 150);  
printf("%p\n", tab);  
tab = (int *) realloc(tab, sizeof(int) * 250000);  
printf("%p\n", tab);
```

affiche

```
0x1205010  
0x7fb123f9d010
```

Exemple d'utilisation de `realloc`

– Exemple –

Ceci construit une chaîne de caractères, caractère par caractère, dans un tableau dynamique dont la **taille est adaptative**. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
chaine = (char *) realloc(chaine, t_reelle + 1); /* Diminution. */
printf("%s \n", chaine);
```

Il faut ajouter la bonne gestion des erreurs en vérifiant ce qui est renvoyé par le `malloc` et les `realloc`, ainsi que par ce qui est renvoyé par le `scanf`.

6. Entrées et sorties

6. Entrées et sorties

6.1 Sortie

6.2 Entrée

6.3 Fichiers

6.4 Fichiers binaires

Écriture formatée

La fonction

```
int printf(char *format, V_1, ..., V_N);
```

permet de réaliser une écriture formatée sur la sortie standard `stdout`.

Rappel : cette fonction renvoie le **nombre de caractères affichés**. Si une erreur a lieu, elle renvoie un entier négatif.

– Exemple –

```
int num;  
num = printf("%s+%d\n", "test", 200);  
printf("%d\n", num);
```

Ces instructions affichent

```
test+200  
9
```

Indicateurs de conversion

On utilise les indicateurs de conversion suivants pour interpréter chaque valeur à écrire (ou à lire, voir la partie suivante) de manière adéquate :

Indicateur de conversion	Affichage
d, i	Entier en base dix
u	Entier non signé en base dix
x, X	Entier en hexadécimal
c	Caractère
e, E	Flottant en notation scientifique
f, g	Flottant
s	Chaîne de caractères
p	Pointeur

Caractères spéciaux

Certains caractères **ne s'affichent pas** mais produisent un **effet** sur la sortie. Ce sont des caractères spéciaux.

Caractère spécial	Rôle
<code>\n</code>	Passage à la ligne suivante
<code>\b</code>	Retour en arrière d'un caractère
<code>\f</code>	Passage à la ligne suivante avec alinéa
<code>\r</code>	Retour chariot
<code>\t</code>	Tabulation horizontale
<code>\v</code>	Tabulation verticale

Caractères d'attribut

Il est possible de faire suivre le `%` d'un indicateur de conversion de caractères d'attribut pour réaliser un formatage avancé.

Caractère d'attribut	Rôle
<code>ON</code>	Affichage du nombre sur <code>N</code> chiffres (ajout de « <code>0</code> »)
<code>N</code>	Affichage du nombre sur <code>N</code> chiffres (ajout d'espaces)
<code>+</code>	Force l'affichage du signe d'un nombre
<code>-</code>	Justifie à gauche un nombre (à droite par défaut)

– Exemples –

```
printf("%+5d\n", 23);
```

affiche

```
  23
```

```
printf("%+-5d %d\n", 23, 5);
```

affiche

```
+23  5.
```

Écriture caractère par caractère

La fonction

```
int putchar(int c);
```

permet d'**afficher un caractère** sur la sortie standard.

Cette fonction renvoie le caractère écrit (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) si une erreur a lieu.

– Exemple –

```
int ret;  
ret = putchar('a');  
if (ret == EOF)  
    /* Gestion de l'erreur. */
```

Ces instructions affichent `a`. Un test est réalisé pour détecter une erreur éventuelle.

6. Entrées et sorties

6.1 Sortie

6.2 Entrée

6.3 Fichiers

6.4 Fichiers binaires

Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une lecture formatée sur l'entrée standard `stdin`.

Cette fonction renvoie le **nombre** d'éléments lus **correctement assignés**.

– Exemples –

```
int num, ret;  
char chaine[128];  
  
chaine[0] = '\0';  
ret = scanf("%d W %s", &num, chaine);  
printf("%d %d %s\n", ret, num, chaine);
```

Ces instructions lisent sur l'entrée standard un entier, un caractère `'W'`, puis une chaîne de caractères.

1. `25 W abc\n`
→ 2 25 abc

2. `25 W abc\n`
→ 2 25 abc

3. `25Wabc\n`
→ 2 25 abc

4. `25 abc\n`
→ 1 25

5. `xy W abc\n`
→ 0 ?

6. `25 w abc\n`
→ 1 25

Exemple d'utilisation de `scanf`

– Exemple –

Ce programme, lorsque exécuté, lit sur l'entrée standard une chaîne d'au plus sept caractères (format `%Ns`), une espace, puis un entier.

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

int main() {
    char prenom[8];
    int age, ret;

    ret = scanf("%7s %d", prenom, &age);
    while (ret != 2) {
        printf("%lu\n", strlen(prenom));
        ret = scanf("%7s %d", prenom, &age);
    }
    return 0;
}
```

Bien noter que `prenom` est un tableau statique de taille `8 = 7 + 1` pour avoir assez de place pour accueillir le caractère de fin de chaîne `'\0'`.

Lecture caractère par caractère

La fonction

```
int getchar();
```

permet de **lire un caractère** sur l'entrée standard.

Cette fonction renvoie le caractère lu (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) lorsque la fin de fichier est détectée (touche Ctrl + D).

– Exemple –

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Ces instructions affichent chaque caractère lu en entrée, tant que `EOF` n'est pas rencontré.

Le tampon d'entrée

Considérons les instructions

```
char c;  
while ((c = getchar()) != EOF)  
    printf("%c", c);
```

Il est possible de saisir plusieurs caractères avant le `'\n'` (touche Entrée).

Avant d'être lus par le `getchar` de la boucle, ils sont situés temporairement dans le tampon d'entrée.

Le tampon d'entrée se comporte comme un **tableau de caractères**.

Chaque appel à `getchar` considère le tampon d'entrée et

- s'il est vide, l'exécution est mise en pause jusqu'à la saisie d'au moins un caractère sur l'entrée standard suivie d'Entrée ;
- s'il contient au moins un élément, ce caractère est supprimé du tampon d'entrée et renvoyé, **sans mettre en pause l'exécution**.

6. Entrées et sorties

6.1 Sortie

6.2 Entrée

6.3 Fichiers

6.4 Fichiers binaires

Descripteurs de fichiers et tête de lecture/écriture

Tout fichier est manipulé par un pointeur sur une variable de type `FILE`.

C'est un type structuré déclaré dans `stdio.h` qui contient diverses informations nécessaires et relatives au fichier qu'il adresse.

Toute variable de type `FILE *` est un descripteur de fichier.

La lecture/écriture dans un fichier se fait par l'intermédiaire d'une tête de lecture.

Celle-ci désigne un caractère du fichier considéré comme le **caractère observé** à un instant donné.

Les fonctions de lecture/écriture ont pour effet (en particulier) de mettre à jour cette tête de lecture en avançant dans son positionnement dans le fichier.

Tête de lecture/écriture

Il existe trois fonctions de `stdio.h` pour **déplacer manuellement la tête de lecture** ou obtenir des informations à son sujet :

- `int ftell(FILE *f);` qui renvoie l'indice de la tête de lecture du fichier pointé par `f`.

Cette fonction est sans effet de bord;

- `int fseek(FILE *f, int decalage, int mode);` qui décale la tête de lecture du fichier pointé par `f` de `decalage` caractères selon le mode `mode`.

Ce paramètre explique à quoi le décalage est relatif (`SEEK_SET` : début du fichier, `SEEK_CUR` : position courante, `SEEK_END` : fin du fichier).

Cette fonction renvoie `0` si elle s'est bien exécutée et `-1` sinon;

- `void rewind(FILE *f);` place la tête de lecture au début du fichier. L'appel `rewind(f)` est ainsi équivalent à `fseek(f, 0, SEEK_SET)`.

Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

Cette fonction renvoie un descripteur de fichier sur le fichier de chemin `chemin` (relatif par rapport à l'exécutable).

La tête de lecture est positionnée sur son 1^{er} caractère.

Le paramètre `mode` désigne le **mode d'ouverture** désiré.

Attention : `fopen` renvoie `NULL` si l'ouverture s'est mal passée. Il faut donc toujours tester la valeur de retour de `fopen`.

Modes d'ouverture

Il existe plusieurs modes d'ouverture. Chacun répond à un besoin particulier :

- "r" : lecture seule.
- "w" : écriture seule. Si le fichier n'existe pas, il est créé.
- "a" : écriture en ajout. Permet d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.
- "r+" : lecture et écriture.
- "w+" : lecture et écriture avec suppression préalable du contenu du fichier. Si le fichier n'existe pas, il est créé.
- "a+" : lecture et écriture en ajout. Permet de lire et d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.

Fermeture de fichiers

La fonction

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

permet de **fermer un fichier**.

Cette fonction permet de mettre à jour le fichier pointé par `f` de sorte que toutes les modifications effectuées soient prises en compte. Le pointeur `f` n'est alors plus utilisable pour accéder au fichier.

Cette fonction renvoie `0` si la fermeture s'est bien déroulée et `EOF` dans le cas contraire.

Attention : toute ouverture d'un fichier lors de l'exécution d'un programme doit s'accompagner tôt ou tard de la fermeture future du fichier en question.

Écriture dans un fichier

La fonction

```
int fprintf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** dans le fichier pointé par `f`.

Cette fonction se comporte comme `printf`, à la différence que cette dernière travaille sur le fichier `stdout`.

– Exemple –

```
FILE *f;
int ret;
f = fopen("fic.txt", "w");
if (f == NULL)
    /* Gestion de l'erreur. */
fprintf(f, "abc\n");
ret = fclose(f);
if (ret == EOF)
    /* Gestion de l'erreur. */
```

Ces instructions déclarent un pointeur sur le fichier `fic.txt` et écrivent `"abc\n"` dans `fic.txt`.

Lecture dans un fichier

La fonction

```
int fscanf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **lecture formatée** depuis le fichier pointé par `f`.

Cette fonction se comporte comme `scanf`, à la différence que cette dernière travaille sur le fichier `stdin`.

6. Entrées et sorties

6.1 Sortie

6.2 Entrée

6.3 Fichiers

6.4 Fichiers binaires

Fichiers binaires — introduction

À la différence des fichiers textes qui sont constitués de caractères (et sont donc lisibles par un être humain directement), les fichiers binaires contiennent des suites de bits **non formatées**.

– Exemple –

Si `ft` pointe vers un fichier texte, l'instruction

```
fprintf(ft, "%d", 261)
```

place les caractères `'2'`, `'6'` et `'1'` dans le fichier en question. Ceci écrit donc les trois octets `00110010`, `00110110` et `00110001`, obtenus en interprétant en base deux les valeurs ASCII de ces caractères décimaux.

En revanche, si `fb` pointe vers un fichier binaire, on emploiera l'instruction d'écriture `fwrite` et, si `v` est une variable entière de valeur `261`,

```
fwrite(&v, sizeof(int), 1, fb)
```

écrit dans le fichier les 4 (= `sizeof(int)`) octets `00000000`, `00000000`, `00000001`, `00000101` à la suite (développement en base deux de la valeur).

Ainsi, on écrit/lit dans/depuit un fichier binaire directement des valeurs, sans se soucier du format à adopter.

Ouverture de fichiers en mode binaire

On utilise les modes d'ouverture habituels avec un `b` en plus pour signaler l'**ouverture en binaire**.

- `"rb"` : lecture binaire seule.
- `"wb"` : écriture binaire seule. Si le fichier n'existe pas, il est créé.
- `"ab"` : écriture binaire en ajout. Permet d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.
- `"r+b"` : lecture binaire et écriture binaire .
- `"w+b"` : lecture binaire et écriture binaire avec suppression préalable du contenu du fichier. Si le fichier n'existe pas, il est créé.
- `"a+b"` : lecture binaire et écriture binaire en ajout. Permet de lire et d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.

Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par `f` ouvert en **mode binaire**.

Cette fonction écrit les `nb` valeurs de taille `taille` octets pointées par le pointeur `ptr` dans le fichier pointé par `f`.

Cette fonction renvoie le nombre de valeurs écrites.

Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

– Exemple –

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;
```

```
int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");
    fwrite(&coul, sizeof(Couleur), 1, f);
    fclose(f);

    return 0;
}
```

Remarque : il n'a pas été nécessaire ici de définir un format d'écriture (ce qui aurait été imposé en cas d'écriture dans un fichier texte).

Lecture dans un fichier binaire

On utilise la fonction

```
int fread(void *ptr, int taille, int nb, FILE *f);
```

pour **lire** dans un fichier pointé par `f` ouvert en **mode binaire**.

Cette fonction lit `nb` valeurs de taille `taille` octets dans le fichier pointé par `f` et les place à l'adresse pointée par le pointeur `ptr`.

Cette fonction renvoie le nombre de valeurs lues.

Exemple d'utilisation de `fread`

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

– Exemple –

```
#include <stdio.h>
int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;
```

```
    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);
    fclose(f);

    f = fopen("fic", "rb");
    fread(tab_2, sizeof(int), 12, f);
    fclose(f);

    return 0;
}
```

Remarque : il n'a pas été nécessaire de se baser sur un format pour la lecture (ce qui aurait été imposé en cas de lecture depuis un fichier texte).

7. Types

7. Types

- 7.1 Notion de type
- 7.2 Types scalaires
- 7.3 Types construits

Types

Un type peut être vu comme un ensemble (fini ou infini) de valeurs.

Dire qu'une variable `x` **est de type** `T` signifie que la valeur de `x` est dans `T`.

Il existe deux sortes de types :

1. les types scalaires, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les types composites, qui sont des assemblages de types scalaires ou de types composites par le biais des constructions `struct`, `enum` ou tableau.

Type d'une variable

Le type d'une variable indique comment **interpréter** la zone mémoire qui lui est attribuée ainsi que sa **taille**.

L'opérateur `sizeof` permet de connaître la taille en octets d'un type. Il est aussi possible de l'appliquer à une valeur ou à une variable, mais nous **interdisons ces usages**.

– Exemple –

`sizeof(int)` et `sizeof(33)` valent 4.

```
int t;  
char c;  
  
printf("%d ", sizeof(int));  
printf("%d ", sizeof(t)); /* Interdit. */  
printf("%d ", sizeof(31)); /* Interdit. */  
  
printf("%d ", sizeof(char));  
printf("%d ", sizeof(c)); /* Interdit. */  
printf("%d\n", sizeof('a')); /* Interdit. */
```

Ceci affiche `4 4 4 1 1 4`.

L'expression `sizeof('a')` vaut 4. En effet, même si `'a'` est un caractère, c'est avant tout un entier.

La conversion est implicite.

Le type d'une variable est **attribué à sa déclaration** et ne peut pas être modifié.