Pour en finir avec les pointeurs en C

Michel Billaud

07-05-2022

Table des matières

Pourquoi ce document ? Les débutants se prennent souvent la tête avec les pointeurs, un sujet considéré comme "avancé" et "difficile".

Il y a souvent une bonne raison à cela : de trop nombreux cours et tutoriels en ligne (je vais éviter de donner des noms pour me fâcher avec personne) prenne ça par le mauvais bout. C'est très mal expliqué, au mauvais moment, et en mélangeant différents aspects :

- ce que c'est,
- les opérations qu'on peut faire dessus en C,
- l'usage qu'on en a pour constituer des structures de données (chaînages en particulier),
- les difficultés d'ordre algorithmique qui s'en suivent.

En réalité, les pointeurs ne sont pas intrinsèquement "difficiles". La définition, c'est simplement

> un pointeur est une **donnée qui contient l'adresse d'une autre donnée**.

Le problème, c'est qu'on se sert des pointeurs pour faire beaucoup de choses. Presque tout, en fait, quand on programme en C. C'est un sujet *riche*.

Orientation : Je n'ai pas voulu faire un cours de C pour débutants complets. Je suppose que le lecteur a commencé à apprendre C (variables, tableaux, fonctions, boucles, structures…), et qu'il veut des explications sur les pointeurs : le sujet est assez vaste comme ça, sans avoir besoin de reprendre tout à zéro.

Contact : Vous pouvez me contacter (michel.billaud@laposte.net) pour toutes remarques, critiques, questions, propositions (honnêtes), contributions, etc.

Copyright : il faut que je me mette à jour sur les différents copyrights. Disons que si vous utilisez une partie significative d'un document, ça parait normal que vous le citiez en référence l'auteur, le titre et le lien où vous l'avez trouvé https://www.labri.fr/perso/billaud/travaux/Pointeurs/pointeurs.html, https://www.mbillaud.fr/dernieres-versions/Pointeurs,…

Je n'ai pas l'intention de devenir riche en vendant mes oeuvres, c'est juste que si vous trouvez qu'un bout est suffisamment intéressant pour que vous vous en inspiriez largement, votre lecteur aura peut-être envie, lui aussi, d'aller voir le document en entier.

Amusez-vous bien !

Versions

- initiale : 25 septembre 2017
- corrections typos : 3-4 février 2019
- corrections : 29 décembre 2020
- corrections : 24 juin 2021 (remerciements à Valentin Morel).
- typos et améliorations légères : 6 mai 2022.

1 Les bases : adresses, pointeurs et indirection

1.1 Notion d'adresse

Vous comprenez certainement le programme suivant :

```c
#include <stdio.h>

int main()
{
    printf("quel est votre âge ?\n");
    int age;
    scanf("%d", & age);

    int annee = 2022 - age;
    printf("vous etes né en %d\n", annee);

    return 0;
}
```

dans le cas contraire, désolé, ce document n'est pas fait pour vous. Lisez d'abord un cours d'initiation à la programmation en C. Et si vous êtes un débutant complet, je vous conseille honnêtement de commencer la programmation avec un langage moins pénible.

Donc, si vous êtes encore là, vous savez que `scanf` sert à lire des données, et `printf` à afficher leur valeur. Et on vous a dit qu'il fallait mettre `&` pour les variables lues par `scanf` (sauf si ce sont des chaînes), et ne pas en mettre dans la liste d'arguments de `printf`. C'est tout-à-fait vrai, on ne vous a pas menti.

Pour être plus précis, `printf` prend comme arguments

- une chaîne de caractères (le *format*),
- et des *valeurs* à afficher.

Du reste on aurait pu se passer de la variable, et écrire directement :

```
printf("vous êtes né en %d\n", 2022 - age);
```

parce qu'on fournirait directement la valeur calculée.

Pour ce qui est de `scanf`, on ne lui donne évidemment pas en argument la *valeur* à lire, puisque cette valeur est fournie par l'utilisateur qui tape la réponse. Ce qu'on indique, c'est l'*endroit* où il faut la mettre. Pour cela, on transmet l'**adresse** de cet endroit.

Et la notation `& age` désigne exactement ceci : l'adresse de la partie de la mémoire qui correspond à la variable `age`.

Donc résumons :

1. Vous avez déjà vu, et utilisé dans vos programmes, des adresses.
2. Une variable correspond à un emplacement dans la mémoire de l'ordinateur. Par exemple pour un `int`, on aura une zone de 4 ou 8 octets, sur les machines actuelles. Un `char` occupera un octet.
3. L'opérateur `&` (on peut l'appeler *address-of*) retourne l'adresse d'une variable.

1.2 Les adresses comme valeurs

Quand vous avez commencé à utiliser `scanf` et `printf`, vous êtes peut-être arrivé à la conclusion qu'il fallait toujours mettre `&` dans un `scanf`, et jamais dans un `printf`.

C'était vrai pour ce que vous en faisiez, mais en fait, vous pouvez parfaitement écrire :

```
printf("adresse de age =  %p", & age);   // voir note ci-dessous
```

ce qui fait afficher l'**adresse** (pas la valeur) de la variable `age`.

```
quel est votre âge ?
12
vous etes né en 2010
adresse de age =  0x7ffdef4accf8
```

L'expression `& age`, si vous préférez, c'est la **valeur de l'adresse**. Une adresse, ça indique une position en mémoire, c'est un numéro d'octet (la mémoire est composée d'octets repérés par leur numéro).

Ici, comme on a utilisé la spécification de format `"%p"` (avec un `p` comme pointeur), l'adresse (qui est un nombre) est affichée en hexadécimal, sous la forme "`0x.....`".

On aurait pu l'afficher en décimal `"%ld"`, mais en général, quand on s'intéresse à ce genre de choses, l'hexadécimal est plus pratique à utiliser.

Note : pour se conformer strictement à la norme du langage C, il faudrait écrire

```
printf("adresse de age =  %p", (void *) & age);
//                             --------
```

pour *transtyper* explicitement l'adresse en "pointeur sur `void`".

Explication : c'est nécessaire parce que la conversion de type pointeur ne se fait pas automatiquement avec la fonction `printf`, qui est *variadique*.

Conseil : ne vous inquiétez pas si vous ne comprenez rien au paragraphe précédent. C'est une formule magique qui me permet d'échapper aux malédictions des puristes, qui m'accuseraient de vous fourvoyer dans des "*undefined behaviours*". Point n'est mon intention. Vous verrez des transtypages (dont l'utilité vous apparaîtra mieux) plus loin dans ce document.

1.3 Et les pointeurs ?

Définition : Un pointeur, c'est une variable (ou une donnée) qui contient l'adresse d'une donnée. Ou plutôt, *peut* en contenir, parce que si le pointeur n'est pas initialisé, il contient a priori n'importe quoi, qui n'est pas forcément une adresse *valide*.

On déclare un pointeur en précédant son nom par le symbole `*`. Exemple

```
int *adresse_age;
```

déclare que la variable `adresse_age` est susceptible de **contenir l'adresse** d'un emplacement mémoire contenant lui-même un `int`.

On peut modifier le programme précédent pour employer cette variable :

```
int *adresse_age  = & age;
...
scanf("%d", adresse_age);
...
printf("adresse de age =  %p", adresse_age);
```

Ci-dessous une représentation des deux variables :

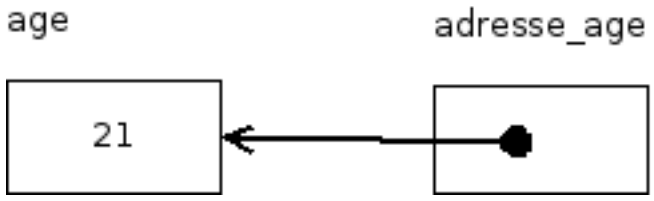

FIG. 1 : Deux variables

Les rectangles figurent les emplacements occupés par des variables, et leur valeur est marquée à l'intérieur. Pour le pointeur, au lieu d'y mettre le nombre `0x7ffdef4accf` qui n'aura pas de grande signification pour nous, on dessine une flèche vers l'emplacement qu'il désigne (l'emplacement numéro `0x7ffdef4accf`, justement).

On dit alors que la variable `adresse_age` **pointe sur** `age`. Ça veut dire qu'elle contient son adresse.

1.4 À quoi vont me servir les pointeurs ?

Un premier usage, c'est pour écrire des **fonctions qui modifient des données**. Pour cela, on donne à ces fonctions **l'adresse** de la donnée à modifier.

Par exemple, une fonction qui demande à taper un entier compris entre deux valeurs. Exemple d'utilisation :

```
int age;
demander_entier("donnez votre age", & age, 0, 150);
```

C'est cohérent avec la manière d'utiliser `scanf` : on fournit l'adresse de la variable `age`, que la fonction est chargée de remplir avec une valeur demandée à l'utilisateur.

Dans la déclaration de la fonction, le paramètre correspondant est donc un pointeur d'entier :

```
void demander_entier(char message[], int *adr_variable, int min, int max);
//                                   ----^------------
```

Il ne reste plus qu'à écrire cette fonction. Une première version :

```
void demander_entier(char message[], int *adr_variable, int min, int max)
{
   printf ("%s (entre %d et %d)", message, min, max);
   scanf("%d", adr_variable);
//             ------------     pas de "&", c'est une adresse
}
```

1.5 L'étoile *, opérateur d'indirection

Pendant qu'on y est, on peut se demander comment faire afficher la valeur de l'âge *à partir du pointeur* qui contient l'adresse de la variable.

```
   int age;
   int *adresse_age  = & age;
   ...
   printf("l'age est %d\n", *adresse_age);
//                           ^-----------
```

C'est ce qu'on appelle faire une **indirection**. On ne veut pas faire afficher la valeur du pointeur, mais *la valeur de la donnée pointée* (par ce pointeur).

Exercice : modifiez la fonction précédente pour qu'elle

- vérifie que le nombre tapé est bien compris entre les valeurs limites,
- repose la question tant que ce n'est pas le cas.

Indication : on mettra `*adr_variable` là où on aurait mis un entier.

1.6 Relation entre usage et déclaration de variable

Un principe fondamental de C (et C++) est que **la déclaration d'une variable ressemble à la manière de l'employer**.

Quand on compare la variable, on écrit quelque chose comme :

```
if ( *adr_variable < min ) {
//   ^------------
   printf("c'est trop petit ...");
}
```

on y emploie `*adr_variable` qui est un `int`, et qui est donc déclarée ainsi :

```
int *adr_variable`.
```

C'est simple, finalement, quand on connaît le principe.

Autre exemple : si on avait un tableau `t` de 10 adresses d'entiers, `t[2]` contiendrait l'adresse d'un entier, et donc `*t[2]` contiendrait un entier. Donc le tableau serait déclaré

```
int *t[10];
```

À quoi ça sert de savoir ça ? En fait, c'est lié à une erreur dans laquelle les débutants tombent souvent. C'est de considérer que la déclaration

```
int * a;
```

déclare `a` comme un pointeur d'entiers, et que "donc" dans

```
int * a, b;
```

`b` serait aussi un pointeur d'entiers. Hé non, perdu. La lecture correcte, c'est que

- `*a` et `b` sont des `int`,
- et donc `a` est un pointeur d'entier,
- mais pas `b`, qui est un `int`.

Pour éviter les confusions, le plus sage est donc d'écrire en collant l'étoile du côté de la variable, voire de déclarer les deux variables séparément (c'est recommandé par certaines "normes de programmation").

```
int *a;
int b;
```

En réalité, C est un langage en "syntaxe libre", le compilateur ne s'intéresse pas à savoir si c'est collé à gauche ou à droite, ou si il y a des espaces.

Si on met des espaces, c'est pour faciliter la vie de ceux qui vont relire le programme. Attention : la personne qui va probablement bénéficier de votre délicate attention d'écrire proprement les choses, ça risque d'être vous un certain temps, pour le corriger tant qu'il ne marche pas, et aussi plus tard quand il faudra le modifier.

Donc écrire les choses proprement, ça a un gros impact. C'est pas un truc décoratif à garder pour plus tard quand ça marchera, au contraire c'est à faire au plus tôt **pour que ça marche** le plus vite possible.

1.7 La constante `NULL`

Un pointeur sert à contenir l'adresse d'une donnée. Parfois on a besoin de savoir que le pointeur ne contient **pas** une adresse valide.

Pour cela, on lui met une valeur spéciale, la constante `NULL` Une constante spéciale, appelée `NULL` sert qui représente "l'adresse de rien du tout". C'est utile par exemple pour initialiser un pointeur.

```
int *pointeur = NULL;
...

if (pointeur != NULL) {
    // faire quelque chose avec *pointeur
    // ...
}
```

Bien entendu, les choses se passeront mal si on tente de "déréférencer" (faire une indirection avec) un pointeur `NULL`. En général, le programme s'arrête avec une "violation mémoire".

Nous aurons l'usage de cette constante `NULL` lorsque nous réaliserons des structures de données chaînées.

1.8 Petits exercices

1.8.1 Échanger deux nombres (exercice guidé)

Vous connaissez la séquence pour échanger les valeurs contenues dans deux variables `a` et `b` :

```
int tmp = a;
a = b;
b = tmp;
```

Faîtes-en une fonction qui échange deux nombres, et qui aura comme paramètres les adresses de ces nombres :

```
void echanger(int *adr_a, int *adr_b)
{
   int tmp =                                  // complétez



}
```

Idée : dans la séquence montrée plus haut, remplacez simplement `a` par `*adr_a`, et pareil pour `b`.

Pour tester la fonction, faites un `main` qui déclare et affecte deux variables, les échange, et affiche leurs valeurs.

```
int main()
{
   int premier = 111;
   int second  = 222;

   echanger (        ,         );

   print("premier=%d, second=%d\n", premier, second);
   return 0;
}
```

1.8.2 Ordonner deux variables

Le bout de programme suivant doit

- lire deux entiers,
- les **ordonner** pour les faire afficher dans l'ordre

Par exemple si on rentre 11 et 22, ou 22 et 11, ça écrira "`min=11, max=22`".

A compléter

```
void ordonner(     ,      )
{
   if (            ) {



   }
}

int main() {
   int a, b;
   printf("donnez deux entiers : ");
   scanf("%d%d",       ,          );
   ordonner(    ,    );
   printf("min=%d, max=%d\n", a, b);
   return 0;
}
```

Suite : dans la fonction `ordonner`, utilisez - si ce n'est pas fait - la fonction `echanger`.

1.9 Complément : pointeurs vers des structures, la notation "flèche"

Lorsqu'on utilise des pointeurs vers des structures, par exemple

```
struct Fraction {
    int numerateur, denominateur;
};

void normaliser(struct Fraction *adr_fraction);
```

on a souvent besoin d'accéder à un champ de la structure dont l'adresse est dans un pointeur.

Par exemple, pour normaliser une fraction, il faut d'abord ramener le dénominateur à une valeur positive. On peut écrire

```
void normaliser(struct Fraction *pf)    // pf = pointeur de fraction
{
   // rectification éventuelle des signes
   if (pf->denominateur < 0) {
       pf->denominateur = - pf->denominateur;
       pf->numerateur   = - pf->numerateur;
   }
   // diviser aussi les deux champs par leur pgcd
   // ...
}
```

La notation `adresse->champ` est simplement une abréviation commode de `(*adresse).champ`.

2 Pointeurs de fonctions

Vous en avez peut-être entendu parler comme quelque chose d'extraordinaire, relevant quasiment de la magie noire. En fait ça n'a rien de compliqué :

- un pointeur d'entier, c'est un truc qui indique où se trouve la valeur d'un entier
- un pointeur de fonction, ça indique où se trouve le code d'une fonction !

Ça sert à plein de choses :

- en "programmation système"[1], pour gérer les signaux (indiquer la fonction à lancer quand un signal sera reçu), pour la programmation parallèle (la fonction qu'un *thread* devra exécuter…),
- au niveau de la bibliothèque standard, la fonction "générique" `qsort` permet de trier un tableau de données de n'importe quel type, à condition de lui fournir une fonction qui permet de comparer,
- pour écrire du code générique, utiliser des "*callbacks*" etc.

2.1 La logique

En fait, c'est une application assez logique des principes déjà vus. Il n'y a pas grand chose de nouveau.

Voyons sur un exemple : nous allons développer une fonction qui applique une même fonction à tous les éléments d'un tableau.

1. D'habitude, pour afficher les éléments d'un tableau d'entiers, on écrit quelque chose comme ça :

```
int main()
{
    int tableau[] = { 111, 22, 3333};

    for (int i = 0; i < 3; i++) {
        printf("%d\n", tableau[i]);  // 1
    }
    return 0;
}
```

2. Maintenant, remplaçons l'action 1 par un appel de fonction :

```
int main()
{
    ...
    for (int i = 0; i < 3; i++) {
        afficher(tableau[i]);    // 2
    }
    ...
}
```

à la fonction qui est définie ainsi

```
void afficher(int n)
{
    printf("%d\n", n);
}
```

Jusque là, tout va bien ?

3. Imaginons que nous ayons réussi de mettre l'adresse de la fonction `afficher` dans un pointeur de fonction :

```
adr_fonction = & afficher;                  // déclaration plus loin
```

[1] Le terme "programmation système" est mal défini. Historiquement ça désignait le code de que l'on écrit, non pas pour réaliser des applications directement, mais pour aider d'autres programmeurs à réaliser des applications, en faisant appel à des mécanismes un peu techniques liés au système d'exploitation. Ou en modifiant le système d'exploitation. Maintenant on l'emploie un peu à tort et à travers, dès que, dans un programme, on appelle une fonction de la bibliothèque du système (comme `fork`, `pipe`, `signal`, …).

il serait cohérent avec les épisodes précédents de remplacer `afficher` par `*adr_fonction` dans la boucle :

```
for (int i=0; i<3; i++) {
    (*adr_fonction)(tableau[i]);    // 2
}
```

4. Maintenant, la déclaration de la variable. Rappelez-vous leprincipe : la déclaration d'une variable ressemble à son utilisation.

Comme `(*adr_fonction)` remplace `afficher` qui a comme prototype

```
void     afficher  (int n);
```

la variable peut être déclarée

```
void (*adr_fonction)(int n);
```

5. En pratique, dans un prototype de fonction, on indique les types des paramètres, mais on peut se passer des noms.

Et voila le code qui en résulte :

```
void (*adr_fonction)(int);    // adr_fonction : pointeur de fonction

adr_fonction = & afficher;    // adresse du code d' afficher

for (int i = 0; i < 3; i++) {
    (*adr_fonction)(tableau[i]);        // appel
}
```

Note on peut aussi écrire

```
    adr_fonction =  afficher;       // 1. sans le &
    for (int i = 0; i < 3; i++) {
        adr_fonction(tableau[i]);   // 2. sans l'étoile
    }
```

Dans le premier cas, c'est une vieille histoire. Le premier standard du langage C (en 1989) est sorti très tard. Il existait de nombreux compilateurs, qui ne traitaient pas les adresses de fonctions de la même façon. Le but du comité de standardisation n'était pas de se fâcher avec la moitié des gens qui écrivaient (et vendaient) des compilateurs - et faisaient partie du comité - en décrêtant que ce qu'ils faisaient était illégal.

Du coup, la comité a décidé que `afficher` ou `& afficher`, ce sont deux manières légales de désigner l'adresse de la fonction `afficher`, avec ou sans "`&`".

Dans le second cas, c'est que la forme d'un appel de fonction est plus générale que ce à quoi vous êtes habitué.

- Avant de lire ce cours, vous connaissiez *nom_de_fonction(paramètre, paramètre, …)*
- et puis maintenant, **pointeur_de_fonction(paramètre, paramètre, …)*,
- et aussi *pointeur_de_fonction(paramètre, paramètre, …)*.

En fait la forme générale, c'est *expression_qui_retourne_une_fonction(paramètre, paramètre, ….)*

Jusqu'ici, on mettait "`&`" et les "`*`" dans les utilisations de pointeurs de fonctions parce que ça permet de bien en voir la logique.

Maintenant que vous avez compris comment ça marche, on va laisser tomber ces opérateurs superflus.

2.2 Déclarer un type pour les fonctions (`typedef`)

Pour jouer avec les fonctions, on a tout intérêt à déclarer un type. Exemple

```
typedef bool (*PredicatEntier) (int);
```

pour les fonctions qui prennent un paramètre de type entier et répondent vrai ou faux.

Il y a deux raisons :

1. d'abord la syntaxe pour décrire une fonction est un peu longue, avec le type de retour et ceux des paramètres, c'est fastidieux à relire autant qu'à écrire
2. ensuite c'est parfois impossible à formuler directement, le compilateur ne voulant pas d'une déclaration bizarre au milieu d'une autre.

En reprenant l'exemple précédent, on peut écrire une fonction qui applique la même action à tous les éléments d'un tableau d'entiers. La ligne

```
typedef  void (* ActionEntier)(int);
```

définit le type `ActionEntier` : les fonctions qui

- prennent comme paramètre un `int`
- ne retournent rien.

Et on utilise ce type pour un des paramètres de `appliquer_action` :

```
void appliquer_action(int tableau[], int taille, ActionEntier action)
//                                                ------------
{
   for (int i = 0; i < taille; i++) {
       action(tableau[i]);
//     ------
   }
}
```

Le `main` devient

```
int main()
{
    int tableau[] = { 111, 22, 3333};
    appliquer_action(tableau, 3, afficher);
    return 0;
}
```

2.3 Fonctions qui retournent des fonctions

On peut même jouer avec des fonctions qui retournent des fonctions.

```
#include <stdio.h>

void manger() { printf("miam miam\n"); }
void dormir() { printf("rrrr zzz\n"); }

// définition d'un type Action = "fonction void sans paramètre"
typedef void (*Action)();

Action que_faire(int heure)
{
    if (heure == 12 || heure == 20) {
        return manger;
    } else {
        return dormir;
    }
}

int main()
```

```
{
    printf("quelle heure est-il (0..24) ?");
    int heure = 12;
    scanf("%d", & heure);
    que_faire(heure) ();  // lance la fonction retournée par que_faire(heure)
    return 0;
}
```

mais bon, il faut en avoir l'usage.

Ce qu'on en retiendra, c'est que l'appel de fonction a une forme générale

expr_1 (*arg_1*, *arg_2*, …)

dans laquelle l'**expr_1** peut être n'importe quelle expression qui retourne un pointeur de fonction. Dans le cas le plus courant, c'est le nom d'une fonction, mais ça peut être autre chose.

2.4 Une application : table d'actions

Ici on utilise une table d'actions, contenant des adresses de fonctions.

```
void aller_au_restaurant()
{
    printf("Miam\n");
}

void retourner_dormir()
{
    printf("Zzzz\n");
}

void aller_au_boulot()
{
    printf("Bof\n");
}

typedef void (*Action)();

Action[3] = {                  // tableau de 3 fonctions
    aller_au_restaurant,
    aller_au_boulot,
    retourner_dormir
};

int main()
{
    for (;;) {
       printf("Action entre 0 et 2 : ");
       int n;
       scanf("%d", &n);              // choix de l'action

       actions[n] ();                // exécution de la fonction correspondante
    }
    return 0;
}
```

3 Les pointeurs et les tableaux

3.1 Qu'est-ce qu'un tableau ?

En C, vous avez peut être remarqué que les tableaux n'étaient pas des variables comme les autres. Par exemple vous pouvez déclarer deux tableaux de même type et de même taille :

```
int a[10], b[10];
```

mais vous ne pouvez pas copier l'un dans l'autre par `a = b;`

```
test.c:12:4: error: assignment to expression with array type
```

La raison, c'est que `a`, ce n'est pas vraiment une variable *contenant* 10 entiers. C'est un emplacement où il y a 10 entiers.

Quand vous écrivez `a = b;`, vous demandez de mettre une adresse dans une autre. Ce n'est pas vraiment ce que vous voulez faire. C'est pas l'adresse que vous voulez copier, c'est le contenu.

3.2 Compatibilité entre tableaux et pointeurs

Bref, il vous faut voir le tableau comme **l'adresse** du premier entier, d'une zone qui en contient 10. Et logiquement vous avez le droit d'écrire :

```
int tableau[10];
int *pointeur;

pointeur = tableau;
```

ce qui met dans `pointeur` l'adresse de `tableau[0]`.

Inversement, vous pouvez aussi utiliser avec les pointeurs la *notation indicée* que vous connaissez sur les tableaux.

```
printf("%d\n", pointeur[3]);
```

Bref, un tableau en C, c'est *comme* un pointeur (non modifiable) vers un espace qui a été réservé pendant la déclaration.

Exercice : que fait afficher la séquence suivante ?

```
pointeur = & tableau[2];
printf("%d", pointeur[3]);
```

Complément : Cette interchangeabilité explique pourquoi vous pouvez utiliser indifféremment les deux versions (pointeurs / tableaux) pour déclarer les fonctions qui ont des tableaux en paramètres

```
void afficher_tableau(int  tableau[], int nombre);
void afficher_tableau(int *tableau  ,  int nombre);
```

3.3 La notation *pointeur+entier*

Si on a un tableau `t` d'entiers (par exemple), la notation `t+k` désigne la position du k-ième entier après celui qui est désigné par `t`

Illustration : le dessin ci-dessous représente la mémoire occupée par un **tableau de trois entiers** déclaré par

```
int t[3];
```

en supposant que chaque `int` est représenté sur 4 octets (les petites cases), ce qui est le cas des machines à architecture 32 bits.

En haut, on voit les positions désignées par `t, t+1`, … (à partir de `t+3`, elles sont en dehors du tableau !), et en bas, les contenus correspondant à `t[0]`, `t[1]`, …

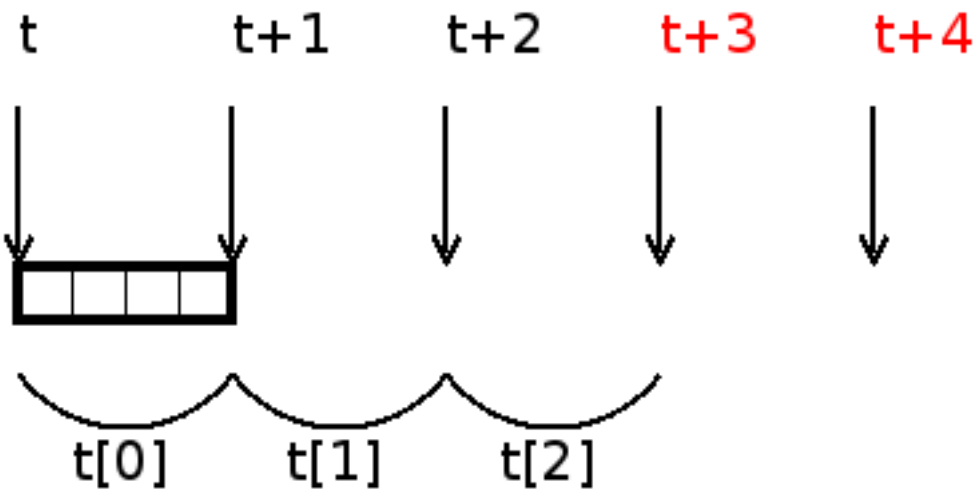

FIG. 2 : Tableau de 3 entiers

Bref, la notation `tableau+entier` ou `pointeur+entier` a un sens en C, c'est la base de ce qu'on appelle **l'arithmétique des pointeurs**.

3.4 L'arithmétique des pointeurs

Exemple, les deux expressions

```
pointeur++;
pointeur = pointeur + 1;
```

signifient toutes deux "mettre dans `pointeur` l'adresse qui est un `int` plus loin que ce que désignait `pointeur` juste avant".

Autrement dit, si `pointeur` désignait une case d'un tableau, maintenant il désigne la case suivante.

Il faut comprendre un peu ce qui se passe en dessous : si un pointeur `p` contient l'adresse d'un objet de type `T`, et que `k` est un entier, `p+k` contient l'adresse de `p`, augmentée de `k` fois `sizeof(T)`

Une démonstration ? Voici un code qui fait afficher la valeur d'un pointeur (de `double`), à qui on ajoute des entiers :

```
int main()
{
    double un_double = 1234.56;

    printf("adresse de un_double = %p\n\n", & un_double);

    double * p = & un_double;

    for (int k = 0; k < 4; k++) {
        printf("       valeur de p+%d = %p\n", k, p+k);
    }
    return 0;
}
```

Ce qu'on voit à l'exécution :

```
adresse de un_double = 0x7ffce909aae8

       valeur de p+0 = 0x7ffce909aae8
       valeur de p+1 = 0x7ffce909aaf0
       valeur de p+2 = 0x7ffce909aaf8
```

```
valeur de p+3 = 0x7ffce909ab00
```

Commentaire : même si les adresses sont affichées en hexadécimal, il n'est pas trop difficile de voir qu'elles progressent de 8 en 8. La raison, c'est que ce sont des adresses de `doubles`, qui occupent chacun 8 octets sur cette machine.

Exercice : que pensez-vous de ce qui suit ?

```
printf("%d\n",  *(pointeur + 4));
```

Exercice : c'est quoi le problème avec ce code (qui plante) ?

```
int nombre    = 0;
int *pointeur = nombre;
*pointeur     = 1234;
printf("%d\n", *pointeur);
```

(Conseil : activer les avertissements du compilateur).

3.5 Retour sur les expressions indicées

Avant de lire ce document, vous connaissiez évidemment la notation indicée sur les tableaux !

```
int tableau[12];
tableau[4] = 23;
```

et vous venez d'apprendre que ça marchait aussi avec les pointeurs

```
int *pointeur = &(tableau[3]);
pointeur[5] = 9;
```

ce qui vous conduit à vous demander si il y a d'autres formes possibles. En fait oui et non, il y en a une seule pour les expressions indicées, qui est plus générale

expression_1 `[` *expression_2* `]`

et qui permet de faire beaucoup de choses. Elle marche quand

- l'expression_1 fournit une adresse
- l'expression_2 fournit un entier.

Un exemple amusant (?) est celui de la conversion d'un nombre (de 0 à 15) en chiffre en hexadécimal

```
int nombre = 11;
char chiffre = "0123456789ABCDEF"[nombre];
```

qui mettra un 'B' dans `chiffre` (je vous laisse y réfléchir).

Un autre exemple

```
char * messages_fr[] = { "bonjour", "au revoir" };
char * messages_en[] = { "hello",   "good bye"};

char ** messages(int code_langue) {
    if (code_langue == 1) {
       return messages_fr;
    } else {
       return messages_en;
    }
}

int main() {
   printf("%s",  messages(1)[0]);
   return 0;
}
```

ici l'*expression_1* est un appel de fonction, qui retourne un pointeur (en fait un tableau de chaînes).

3.6 Les chaînes de caractères

En C, il n'existe pas de type spécifique pour les chaînes de caractères. En fait une chaîne, c'est une suite d'octets en mémoire, terminée par un caractère nul (`'\0'`).

Et pour désigner une chaîne de caractères, on donne simplement l'adresse de son premier octet.

Le compilateur offre simplement quelques raccourcis. Une chaîne notée entre guillemets représente un tableau de caractères.

Les deux notations sont équivalentes

```
char chaine[] = "hello";
char chaine[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

elles réservent 6 octets pour le contenu de la chaîne et le caractère nul qui sert de terminateur.

Mais pas tout-à-fait équivalentes à

```
char *pointeur = "world";
```

qui réserve de la place

- dans un espace mémoire pour `"world"`
- pour un pointeur contenant initialement l'adresse de cette chaîne

Notez qu'on pourra faire `chaine[0] = 'H';` mais pas `pointeur[0] = 'W'` : la place pour `"world"` a été réservée dans un espace protégé en écriture. C'est une constante.

Bref, une chaîne de caractère est décrite par l'adresse de son premier caractère, son contenu va de cette adresse jusqu'au premier caractère nul qui suit.

Traditionnellement, l'écriture de fonctions sur les chaînes de caractères est souvent faite par le biais de pointeurs plutôt que d'indices.

Pour reprendre un exemple éculé, le calcul de la taille d'une chaîne peut s'écrire ainsi (avec des indices) :

```
int longueur(char chaine[])
{
    int i = 0;
    while (chaine[i] != '\0') {
        i++;
    }
    return i;
}
```

ou, avec un pointeur

```
int longueur(char chaine[])
{
    char *p = chaine;
    while (*p != '\0') {
        p++;
    }
    return p - chaine;
}
```

mais plutôt sous la forme idiomatique (boucle `for`) du parcours d'une chaîne

```
int longueur(char chaine[])
{
    for (char *p = chaine; *p != '\0; p++) {
    };
```

```
    return p - chaine;
}
```

Dans tous les cas, l'opération "`p++`" fait glisser le pointeur vers le caractère suivant. L'expression `p - chaine` indique la "distance" (nombre d'octets) entre le début de la chaîne et le caractère nul qui la termine .

4 Un exemple avec tout ça

4.1 Objectif : gestion de stocks

Ci-dessous vous trouverez un début de programme dont le prétexte est la gestion d'un stock d'articles.

Chaque article est enregistré dans une structure qui regroupe une référence (un numéro), une description, et un nombre d'articles.

Le stock est enregistré dans un tableau d'articles, plus précisément dans une structure qui contient à la fois un tableau, et le nombre d'articles (les articles enregistrés seront en début de tableau).

4.2 Les fonctions qui agissent sur un Stock

Pour agir sur le stock, on a écrit un certain nombre de fonctions qui ont donc un "`struct Stock *`" en paramètre. On passe un **pointeur** pour plusieurs raisons :

- **parce qu'on ne peut pas faire autrement** pour les opérations qui ont pour but de modifier le stock. Avec un passage par valeur, la fonction modifierait une copie et pas le stock qu'on lui indique.
- **efficacité** parce si on passe une `struct Stock` par valeur, on en ferait une copie. Faites afficher `sizeof(struct Sock)` pour voir combien d'octets seraient copiés à chaque appel. Attention, `sizeof` retourne un entier non signé (type `size_t`) pour lequel il faut utiliser la spécification de format `"%zu"` (merci à Marc Mongenet de me l'avoir rappelé).
- **par homogénéité** pour que tous les appels se ressemblent, et que le programmeur d'application ne doive pas faire d'effort de mémoire pour retrouver les noms.

C'est la même situation que quand vous utilisez des `FILE *` comme paramètres de diverses fonctions : `fprintf`, `fscanf`, `fclose`, `feof`, `fread`, `fwrite`….

4.3 Le source

```
#include <stdio.h>

#include <string.h>
#include <stdbool.h>

#define TAILLE_MAX_STOCK         100
#define LONGUEUR_MAX_DESCRIPTION 50

struct Article {
    int reference;
    char description[LONGUEUR_MAX_DESCRIPTION];
    int quantite;           // alternative : unsigned à la place de int
};

struct Stock {
    int nombre_articles;                        // ici aussi
    struct Article articles[TAILLE_MAX_STOCK];
};
```

```c
void initialiser_stock(struct Stock *adr_stock)
{
    adr_stock->nombre_articles = 0;
}

bool stock_est_plein(struct Stock *adr_stock)
{
    return adr_stock->nombre_articles == TAILLE_MAX_STOCK;
}

/**
 * cherche dans un stock un article à partir de sa référence.
 * retourne un pointeur sur l'article si présent, NULL sinon.
 */
struct Article *chercher_par_reference(struct Stock *adr_stock,
                                       int reference)
{
    for (int i = 0; i < adr_stock->nombre_articles; i++) {
        if (adr_stock->articles[i].reference == reference) {
            return adr_stock->articles + i;              // trouvé
        }
    }
    return NULL;                                         // pas trouvé
}

/**
 * ajoute un article dans le stock.
 * on transmet l'adresse pour éviter de faire une copie
 *
 * PRECAUTIONS, il est de la responsabilité de l'utilisateur de la
 * fonction de vérifier
 * - que le stock n'est pas plein.
 * - que la référence n'est pas déjà présente.
 * - que la description n'est pas trop longue
 */

void ajouter_article(struct Stock *adr_stock,
                     struct Article *adr_article)
{
    adr_stock->articles[adr_stock->nombre_articles++] = *adr_article;
}

/**
* définition du type "fonction qui agit sur un article"
* la fonction reçoit l'adresse de l'article à traiter
*/

typedef void (*ActionSurArticle)(struct Article *);

/**
* pour_tout_article
* applique une même action à tous les articles d'un stock
*/

void pour_tout_article(struct Stock *stock, ActionSurArticle action)
{
```

```c
    for (int i = 0; i < stock->nombre_articles; i++) {
        printf("%3d ", i);
        action(stock->articles + i);               // c'est une adresse
    }
}

/**
* affichage d'un article (passé par pointeur)
*/

void action_afficher_article(struct Article *adr_article)
{
    printf("%06d %6d %s\n", adr_article->reference,
                            adr_article->quantite,
                            adr_article->description);
}

// --- les tests ----------------------------------------

void test_recherche(struct Stock *adr_stock, int reference)
{
    printf("- Recherche de la réference %d\n", reference);

    struct Article *adr_article
                    = chercher_par_reference(adr_stock, reference);
    if (adr_article == NULL) {
        printf("absente\n");
    } else {
        action_afficher_article(adr_article);
    }
}

void test_1 ()
{
    struct Stock stock;
    struct Article article_patates = {
        .reference = 123,
        .quantite = 20,
        .description = "Sac 3kg de patates"
    };
    // initialisation de structure style C99, voir
    // http://en.cppreference.com/w/c/language/struct_initialization

    struct Article article_chaussettes = {
        .reference = 234,
        .description = "Paire de chaussettes vertes",
        .quantite = 12
    };

    struct Article article_logiciel = {
        .reference = 89,
        .description = "Compilateur C ANSI en solde",
        .quantite  = 5
    };

    printf("* Initialisation et ajouts\n");
```

```
    initialiser_stock(& stock);
    ajouter_article(& stock, &article_patates);
    ajouter_article(& stock, &article_chaussettes);
    ajouter_article(& stock, &article_logiciel);

    printf("* Parcours\n");

    pour_tout_article(& stock, action_afficher_article);

    printf("* Tests de recherche\n");

    test_recherche(& stock, 234);
    test_recherche(& stock, 999);
    }

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

int main()
{
    test_1();
    return 0;
}
```

4.4 Travail proposé

- Essayer de comprendre.
- Compléter avec des fonctions qui permettent de consulter la quantité disponible pour une référence donnée, la modifier, etc.
- Écrire un programme qui dialoguera avec l'utilisateur pour gérer un stock.
- Faire une bibliothèque, compilée séparément.

4.5 Indications pour la compilation séparée

Si on part sur l'idée d'une compilation séparée, le fichier source contiendra un "`include`" de la bibliothèque, et les fonctions qui sont particulières au test :

```
#include <stdio.h>
#include "stock.h"

void test_recherche(struct Stock *adr_stock, int reference)
{
    printf("- Recherche de la référence %d\n", reference);
    ...
}

void test_1()
{
    ....
}

void main()
{
    test_1();
    ...
}
```

Le fichier "`stock.h`" contient les constantes, les déclarations de type et les prototypes des fonctions

```
#ifndef STOCK_H
#define STOCK_H

#include <stdbool.h>

#define TAILLE_MAX_STOCK         100
...

struct Article {
    ...
};

struct Stock {
    ...
};

void initialiser_stock(struct Stock *adr_stock);
bool stock_est_plein(struct Stock *adr_stock);
...

#endif
```

Quant au fichier "`stock.c`, il fait une inclusion du fichier d'en-tête, et contient le code des fonctions

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

void initialiser_stock(struct Stock *adr_stock)
{
    adr_stock->nombre_articles = 0;
}

bool stock_est_plein(struct Stock *adr_stock)
{
    return adr_stock->nombre_articles == TAILLE_MAX_STOCK;
}
```

5 Pointeurs et typage

5.1 Les pointeurs sont typés

Jusqu'ici nous avons utilisé des pointeurs "sur des entiers", des pointeurs "sur des structures", etc.

Ces **pointeurs sont typés**, c'est-à-dire que le compilateur connaît le type des objets pointés. Il y a deux raisons pour cela

1. D'une part le compilateur vérifie qu'on ne se mélange pas les pinceaux en écrivant

```
int   *p1, *p2;
float *p3;

p1 = p2;    // oui
p2 = p3;    // erreur, types incompatibles
```

2. D'autre part le compilateur a besoin de connaitre la longueur du type pointé pour pouvoir traduire les expressions avec indices et l'arithmétique des pointeurs.

En effet, que fait `p = p + 1`, si p est un pointeur d'entiers (`int`) ?

Supposons que `p` contienne l'adresse `0x7ffce90901234`. C'est l'adresse (sur 64 bits, les zéros non significatifs ne sont pas indiqués) d'un `int` qui occupe 32 bits, soit 4 octets. (Les chiffres sont donnés pour mon PC à architecture 64 bits, ils peuvent varier selon les machines, les systèmes et les compilateurs).

`p+1` pointe "un `int` plus loin" en mémoire, donc correspond à la valeur `0x7ffce90901234 + 4 = 0x7ffce90901238`. L'opération `p++` consiste donc à ajouter 4 (le nombre d'octets d'un `int`) à l'adresse contenue dans `p`.

En résumé avec la déclaration

T `*ptr;`

l'instruction `ptr++` ajoute en réalité `sizeof(T)` à l'adresse contenue dans `ptr`.

5.2 Pointeurs génériques

Quand on programme "à bas niveau" (ce qui est le créneau spécifique du langage C), on a parfois besoin d'écrire des fonctions qui agissent sur des paramètres de différents types.

Par exemple, je voudrais voir comment sont codées les données, et pour cela faire afficher leur contenu en hexadécimal.

5.2.1 Exemple d'utilisation

Pour cela, je vais définir une fonction `afficher_en_hexa` que j'appellerai ainsi

```
int   un_entier   = 1789;
float un_flottant = 3.14;
afficher_en_hexa(& un_entier,   sizeof(un_entier));
afficher_en_hexa(& un_flottant, sizeof(un_flottant));
```

Le premier paramètre sera l'adresse du premier octet de la donner à afficher, le second sa longueur.

Remarquez que dans le premier cas l'argument donné est un `int*`, dans le second un `float*`. Il va y avoir un souci de compatibilité de types.

5.2.2 Comment afficher en hexadécimal le contenu d'une zone mémoire ?

Ce n'est pas difficile si la zone est considérée comme un tableau d'octets (en C, octet = `char`)

```
char *tableau = ....
for (int i = 0; i < longueur; i++) {
   printf("%x02", tableau[i]);
}
```

La spécification de format `%x02` s'analyse comme suit

- `x` : afficher en hexadécimal
- `0` : en mettant éventuellement des 0 non significatifs
- `2` : sur une largeur de 2 caractères

5.2.3 Pointeurs génériques : le type `void*`

Pour déclarer la fonction, nous allons utiliser le type spécial `void*`, qui signifie "pointeur sur un type inconnu".

```
void afficher_en_hexa(void *adresse, int longueur);
```

Ce type est compatible, pour l'affectation, avec les autres pointeurs. C'est ce qu'on appelle un **pointeur générique**.

1. d'une part ça permet l'appel de la fonction avec n'importe quel type d'adresse : `afficher_en_hexa(& un_truc, ....)`

2. d'autre part ça autorise l'affectation `tableau = adresse` dans le corps de la fonction.

```
void afficher_en_hexa(void *adresse, int longueur)
{
    char *tableau = adresse;
    for (int i = 0; i < longueur; i++) {
        printf("%x02", tableau[i]);
    }
    printf("\n");
}
```

3. Par contre un pointeur générique ne peut évidemment pas être dé-référencé, et on ne peut pas faire d'arithmétique des pointeurs avec, puis qu'on ne connaît ni le type, ni la longueur de ce qu'il pointe.

5.3 Conversions explicites de pointeurs (transtypage, *typecast*)

Une autre façon de faire "à l'ancienne" aurait été de définir la fonction avec un paramètre "pointeur d'octet" :

```
void afficher_hexa(char *adresse, int longueur)
{
  ...
}
```

ce qui interdit l'appel

```
afficher_hexa(& un_entier,        // Erreur: types incompatibles
              sizeof(un_entier));
```

parce que les types `int*` et `char*` sont incompatibles.

Dans ce cas, la solution est de forcer la conversion de type ("*typecast*")

```
afficher_hexa((char *) &un_entier,
              sizeof(un_entier));
```

en précédant l'adresse par "`(char *)`" qui signifie "considéré comme pointeur de `char`".

Vous verrez ça dans de nombreux exemples que vous trouverez sur Internet. Il faut savoir que nombreuses bibliothèques (allocation dynamique, réseau, ...) ont été écrites à une époque où le type `void*` n'avait pas encore été introduit en C (C89). La pratique normale était alors d'utiliser des `char*`, ce qui imposait d'avoir à faire un "transtypage" explicite.

Nous sommes au XXIe siècle, Les bibliothèques ont été modifiées depuis, mais les exemples préhistoriques sont restés.

Résumé : si **expr** est une expression de type "pointeur sur type T", alors

(T) *expr*

est de type "pointeur sur T".

6 L'allocation dynamique

6.1 Préliminaires, durée de vie des variables

Considérons un programme simple, avec une variable globale

```
#include <stdio.h>

int nombre_appels_carre = 0;

double carre(double nombre)
```

```
{
    nombre_appels_carre++;
    double tmp = nombre * nombre;
    return tmp;
}

void test(double x)
{
    float c = carre(x);
    printf("le carré de %f est %f\n", x, c);
}

int main()
{
    test(12.0);
    test(3.14);
    printf("fonction carre appelée %d fois\n", nombre_appels_carre);
    return 0;
}
```

On y voit plusieurs catégories de variables

- la variable `nombre_appels_carre` qui est **globale**,
- les paramètres `nombre` et `xxx`, qui sont des noms pour les valeurs reçues lors d'un appel
- les variables **locales** `tmp` et `c`

et vous savez qu'elles n'ont pas la même visibilité :

- la variable globale est accessible depuis toutes les fonctions
- alors que les variables locales (et les paramètres) n'existent que dans leur fonction (le même nom peut être utilisé dans plusieurs fonctions) ;

et surtout, pas la même durée de vie :

- la variable globale existe pendant toute la durée d'exécution du programme : sa place a été réservée au début de l'exécution. C'est ce qu'on appelle une variable **statique**.
- une variable locale, n'existe que pendant l'exécution de la fonction : de l'espace est réservé *automatiquement* (sur la pile des appels) pour loger la variable quand la fonction est appelé, et rendu *automatiquement* au retour de la fonction (exécution de `return`, ou de la dernière instruction). C'est ce qu'on appelle une variable **automatique**.

Le dessin ci-dessous (**pile des appels**) représente l'évolution (le temps va de gauche à droite) du contenu de la mémoire au cours de appels et des retours. En haut figure la pile d'exécution, avec dans chaque **contexte** les variables et leur valeur. En bas, la variable globale.

Au delà de ces deux espaces mémoire, la zone statique et la pile, il en existe un autre qu'on appelle "le tas" (*heap*), qui est géré différemment.

- dans la zone statique, les données existent pendant toute la durée du programme ;
- dans la pile, les données apparaissent et disparaissent automatiquement au fil des appels et retours de fonctions,
- pour le tas, c'est le programmeur qui demande au système, quand il le souhaite, - de lui prêter une zone d'une certaine taille (**allocation**), - de la récupérer (**libération**).

6.2 Allocation et libération en C

Les deux fonctions intéressantes sont

```
void *malloc(size_t size);   // allocation
void free(void *pointer);    // libération
```

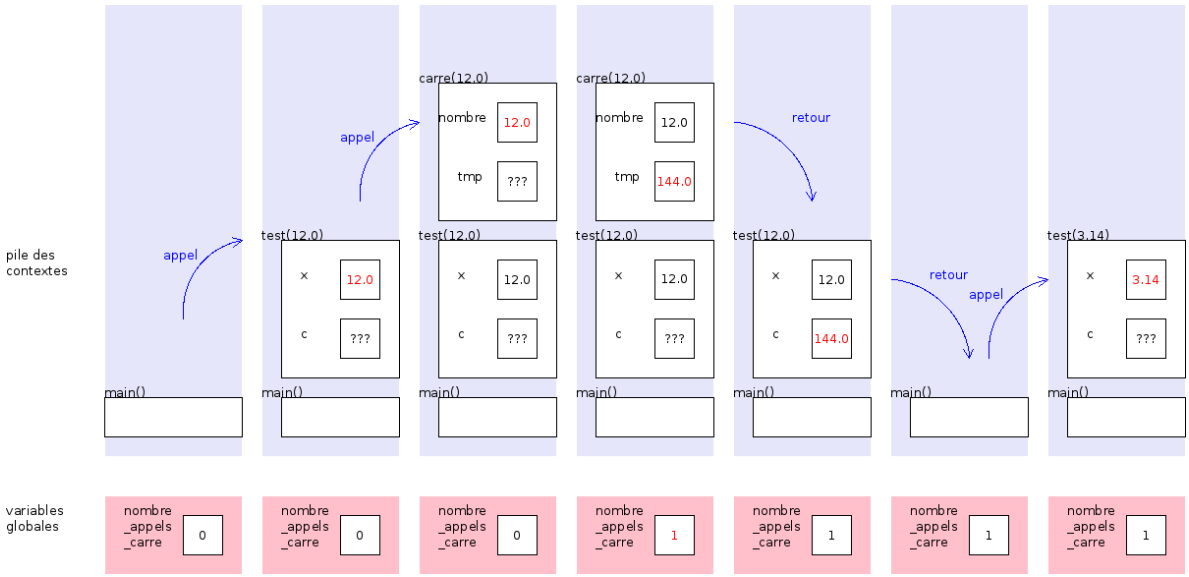

FIG. 3 : Pile des appels

- La première `malloc` (*memory allocation*) reçoit en argument la taille voulue (en nombre d'octets), elle retourne un pointeur contenant l'adresse du premier octet obtenu ;
- la seconde prend comme argument l'adresse d'une zone ainsi allouée, et demande au système de la récupérer.

Pour utiliser ces fonctions, il faut

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

Attention

- `malloc` peut retourner `NULL`, si la taille`size` est 0, ou si il n'y a plus de place disponible pour l'allocation
- `free` doit recevoir comme paramètre l'adresse d'une zone
  - qui a été allouée précédemment par `malloc`
  - qui n'a pas déjà été libérée par `free`
- la valeur `NULL` est aussi acceptée par `free`.

Le programmeur doit donc être particulièrement attentif

- à vérifier que le `malloc` qu'il demande a réussi
- à libérer les zones allouées quand il n'en a plus besoin (sinon il risque d'y avoir une *fuite mémoire*)
- à ne pas tenter de libérer une seconde fois une zone qui a déjà été libérée (problème de *double libération*)

6.3 Un exemple : fabrication d'une table

Un exemple simple : on veut une fonction qui fabrique une table des carrés des nombres entiers de 1 à n (exemple idiot)

D'abord, voici comment on va l'utiliser

```
int main()
{
    int *table = table_carres(10);  // création de la table

    for (int i = 0; i < 10; i++) {  // affichage
        printf("%d\n", table[i]);
    }
```

```
    free(table);                    // libération
    return 0;
}
```

La fonction aura deux étapes : réserver un espace assez grand, et y mettre les carrés. et bien sûr retourner l'adresse de la table :

```
int *table_carres(int n)
{
    int *table = malloc(n * sizeof(int));   // allocation

    for (int i = 0; i < n; i++) {           // remplissage
        table[i] = (i+1)*(i+1);
    }
    return table;
}
```

Remarques :

- puisqu'on veut une table de `n` entiers, le paramètre de `malloc` est logiquement

  `n * sizeof(int)`

- la situation est asymétrique : l'espace est réservé pendant l'exécution de la fonction `table_carres`, il lui survit quand la fonction est terminée. La libération se fait ailleurs (dans `main`).
- Vous avez remarqué qu'on ne teste pas si `malloc` a échoué. Vous avez raison, on devrait. C'est vrai qu'il n'y a aucun risque avec **ce** programme sur un *PC* ou un Mac, ou… Par contre, pour un programme qui utiliserait des données beaucoup plus grosses (une séquence vidéo, par exemple, peut occuper quelques giga-octets), ou sur un processeur beaucoup plus petit (micro-contrôleurs en informatique embarquée, avec quelques kilo-octets de mémoire), c'est une autre histoire.

6.4 Détection d'erreur, stratégies de récupération.

Donc il va falloir tester le résultat de `malloc`. Mais pour faire quoi ?

Une première stratégie possible est de faire mourir le programme de suite, après avoir émis un message d'adieu déchirant sur la sortie d'erreur :

```
int *table_carres(int n)
{
    int *table = malloc(n * sizeof(int));   // allocation

    if (table == NULL) {
        fprintf(stderr, "ERREUR FATALE : table_carres(%d), échec allocation", n);
        exit(EXIT_FAILURE);
    }
    ...
}
```

Une seconde stratégie est de "faire remonter" l'erreur, en retournant `NULL` à l'appelant

```
int *table_carres(int n)
{
    int *table = malloc(n * sizeof(int));   // allocation

    if (table == NULL) {
        return NULL;
    }
```

```
    ...
}
```

qui se voit refiler le mistigri : à lui de vérifier le résultat de `table_carre`, et d'agir en conséquence

```
int main()
{
    int *table = table_carres(10);  // création de la table

    if (table == NULL) {
        printf("Il semblerait qu'on manque de mémoire\n");
        printf("alors on arrête tout\n");
        return 1;
    }

    ...
```

éventuellement, "agir en conséquence", ça veut dire fermer proprement des fichiers, libérer d'autres variables, terminer des connexions réseau, etc. Ou se contenter d'une table plus petite. Ça dépend du problème.

7 Une application de l'allocation dynamique : un tableau extensible

Un bon exercice est la constitution d'un type de données pour les tableaux extensibles.

7.1 Qu'est-ce que c'est ?

Un tableau extensible, c'est un "conteneur" qui, comme un tableau, regroupe des cases repérées par un "indice". Mais à la différence des tableaux ordinaires, il peut être re-dimensionné (agrandi ou rétréci).

Les opérations de base sur un tableau extensible :

- l'initialiser avec une certaine taille,
- consulter la valeur de l'élément d'indice `i`,
- changer la valeur de l'élément d'indice `i`,
- changer la taille du tableau,
- consulter sa taille,
- et enfin, restituer l'espace qu'il occupe.

On va ici définir un type "`Array`" pour les tableaux de nombres flottants. Ça nous changera des `int`, mais bien sûr c'est adaptable pour n'importe quel type de données.

7.2 Définition du type, fonctions de base

Comment on fait ? c'est assez simple : un `Array`, c'est une structure avec

- un tableau de `float`
- un entier qui mémorise le nombre d'éléments du tableau

Le tableau sera alloué dynamiquement, parce qu'on ne connait pas *a priori* sa taille, et qu'elle pourra changer, donc il sera repéré par un pointeur :

```
struct Array {
     int    size;
     float *data;   // alloué dynamiquement
};
```

et on va définir une série de fonctions, dont le nom commencera par `array_`, et qui prendront comme premier paramètre un pointeur sur une telle structure.

Pour l'initialiser, on définit une fonction `array_init`

```
void array_init (struct Array *array_ptr, int size)
{
    array_ptr->size = size;
    array_ptr->data = malloc(size * sizeof(float));
}
```

et pour le libérer :

```
voir array_free(struct Array *array_ptr)
{
    free(array_ptr->data);
    array_ptr->data = NULL;    // précaution contre double array_free
    array_ptr->size = 0;
}
```

pour changer une valeur dans le tableau

```
void array_set(struct Array *array_ptr, int index, float new_value)
{
    array_ptr->data[index] = new_value;
}
```

pour connaitre la taille du tableau, je vous laisse faire

```
int array_size(                              )
{
    return
}
```

pour lire une valeur dans le tableau, vous y arriverez très bien

```
float array_get(                  ,            )
{
    return
}
```

Avec tout ça, vous pouvez constituer un programme de test

```
void test()
{
    struct Array a;
    printf("quelle taille ? ");
    int taille;
    scanf("%d", &taille);

    array_init(&a, taille);

    // test nombre éléments
    printf("le tableau contient %d flottants\n", array_size(&a));

    // écriture (de 100 en 100);
    for (int i = 0; i < array_size(&a); i++) {
        array_set(&a, i,  100*i);
    }

    // consultation
    for (int i = 0; i < array_size(&a); i++) {
        printf("%d -> %f\n", i, array_get(&a, i));
    }
```

```
    // libération
    array_free(&a);
}
```

auquel il ne manque plus que le re-dimensionnement. Voir plus loin.

Ci-dessous, une illustration de ce qui se passe en mémoire (pile des appels et tas) au début de l'exécution du `main`, pendant l'appel de `array_init`. La zone réservée pour l'allocation dynamique est en bas. On voit que l'objet réservé par `malloc` survit à la sortie de la fonction `array_init` qui demandé son allocation.

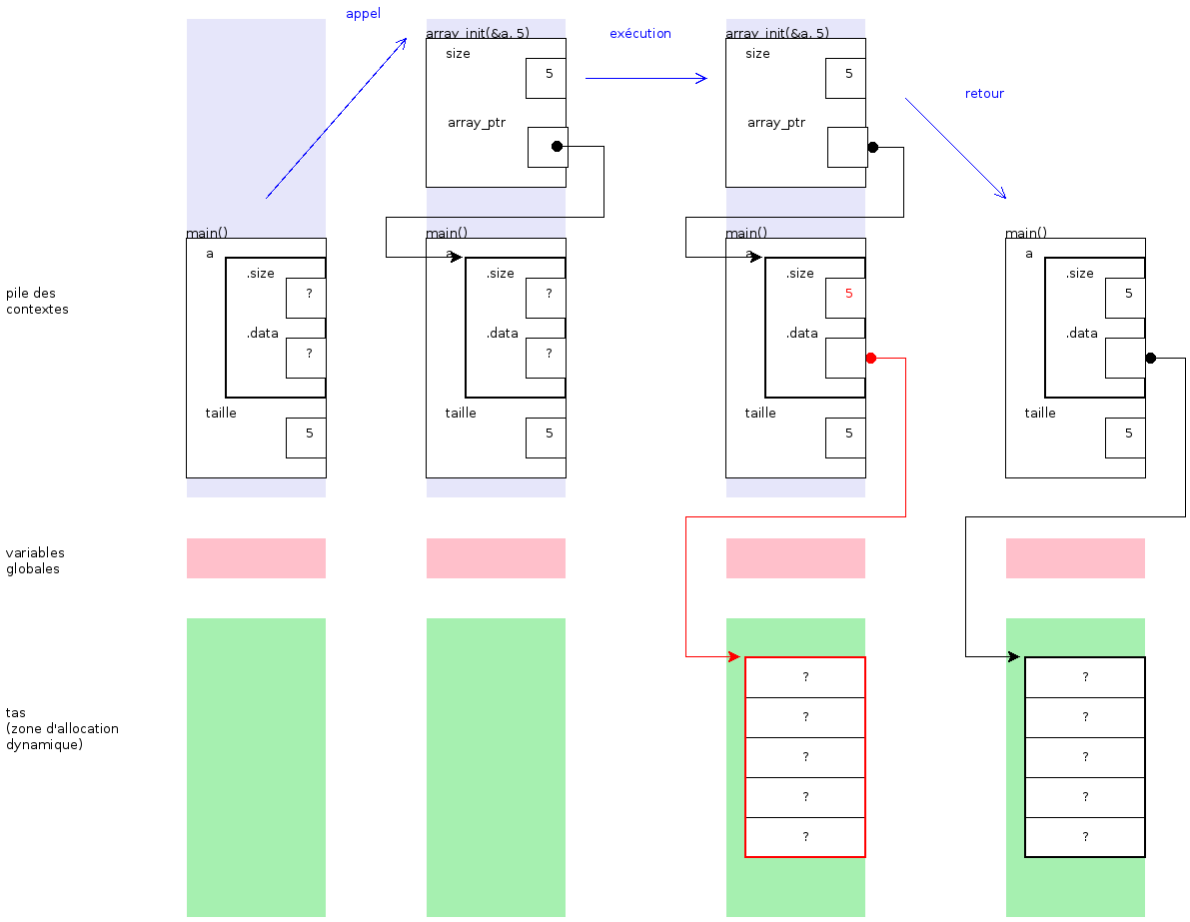

FIG. 4 : Déroulement de l'appel de `array_init`

7.3 Re-dimensionnement

Pour re-dimensionner, on va utiliser la fonction `realloc` qui fait le gros du travail :

```
void array_resize(struct Array *array_ptr, int new_size)
{
    array_ptr->data = realloc(array_ptr->data,
                              new_size * sizeof(float));
    array_ptr->size = new_size;
}
```

La fonction `realloc` prend comme paramètres

- un pointeur sur une zone déjà allouée,
- une nouvelle taille ;

elle s'arrange pour réallouer la zone indiquée, avec la taille voulue, au besoin en la déménageant ailleurs. Le contenu a été copié au passage.

7.4 Exercice : écrire un test

Écrire un test où le tableau est

- créé et initialisé avec 10 éléments,
- affiché,
- redimensionné à 20, on ajoutera des valeurs dans les cases supplémentaires,
- affiché,
- réduit à 5,
- affiché.

7.5 Exercice : écrire la fonction `array_resize` sans `realloc`.

Pour ça, il faut

- réserver une zone à la nouvelle dimensionn
- recopier l'ancienne zone "data" dans la nouvelle (attention aux deux cas : agrandir et rétrécir),
- libérer l'ancienne zone,
- remplacer l'ancienne par la nouvelle,
- et ne pas oublier de modifier le champ `size`…

8 Chaînages

On parle de **chaînage** quand on a des éléments dont un champ sert à indiquer où se trouve un autre élément (le suivant). Bien entendu, quand les éléments sont des données en mémoire, les pointeurs sont un moyen privilégié de matérialiser ce chaînage, grâce à un champ qui indique l'adresse du suivant.

Voici par exemple des éléments, qui ont pour vocation

- de contenir une valeur entière ;
- d'être chaîné avec un autre élément.

```
struct Element {
    int value;
    struct Element *next_ptr;
};
```

Note : on garde ici la convention de mettre le suffixe "_ptr" pour les données qui contiennent des adresses. C'est un peu plus long mais si on met `next` tout court (ce que font la plupart des tutoriels sur les pointeurs), on risque des confusions entre "l'élément suivant" qui est une structure avec "le pointeur vers l'élément suivant" qui est une adresse.

Le plus souvent, ce chaînage sert à représenter des *séquences* (suite linéaire d'éléments, le dernier n'a pas de suivant). Mais plus généralement, il pourrait aussi y avoir des chaînages circulaires (on dit aussi *listes circulaires*), ou des chaînes terminées par un cycle.

Le pointeur `NULL`, qui marque l'absence d'élément, est dessiné sous forme d'une croix.

8.1 Notions de base : les listes simples

Pour représenter une liste, il nous faut un pointeur sur le premier élément.

```
struct Element * first_ptr;
```

Ce pointeur qui sera nul si la liste est vide.

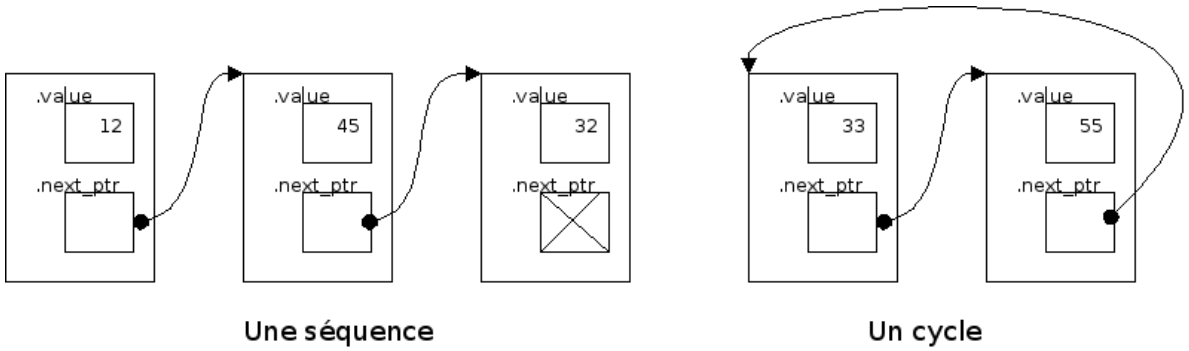

FIG. 5 : Chainages

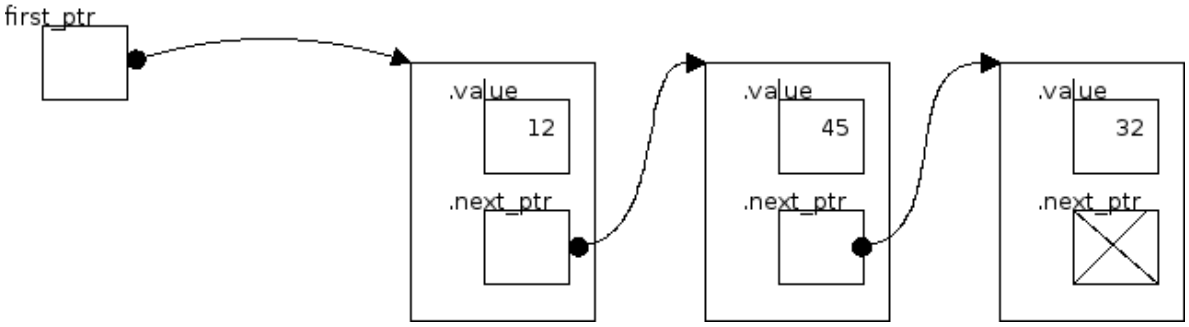

FIG. 6 : Une liste

8.2 Pour parcourir une liste

nous aurons un pointeur qui contiendra d'abord l'adresse du premier élément (soit `first_ptr`), puis du second, etc.

```
struct Element *ptr = first_ptr;  // adresse du premier (si il existe)
while (ptr != NULL) {
    printf("%d\n", ptr->value);
    ptr = ptr->next_ptr;          // passage au suivant
}
```

Avec l'habitude, on préférera la boucle `for` idiomatique

```
for (struct Element *ptr = first_ptr; ptr != NULL; ptr = ptr->next_ptr) {
    printf("%d\n", ptr->value);
}
```

8.3 Pour construire une chaîne vide

Au départ, la chaîne est vide, le pointeur de début est donc initialisé à `NULL`.

```
first_ptr = NULL;
```

8.4 Pour ajouter un élément en tête

C'est l'opération d'ajout la plus facile. A priori il y a deux cas à considérer : soit la liste est vide, soit elle ne l'est pas (!)

1. Si elle est vide, il faut mettre dans `first_ptr` l'adresse d'un nouvel `Element`, créé par `malloc`, contenant la valeur souhaitée et qui n'a pas de suivant. Et dire que cet élément est maintenant le premier de la liste. Soit un code de la forme :

```
if (first_ptr == NULL) {
    struct Element new_ptr = malloc(sizeof (struct Element));
    new_ptr->value         = 1234;
    new ptr->next_ptr      = NULL;
    first_ptr              = new_ptr;
}
```

2. Si la liste n'est pas vide, c'est presque pareil, sauf que le suivant du nouvel `Element` est l'ancien premier (qui devient donc le second)

```
if (first_ptr != NULL) {
    struct Element new_ptr = malloc(sizeof (struct Element));
    new_ptr->value         = 1234;
    new_ptr->next_ptr      = first_ptr;
    first_ptr              = new_ptr;
}
```

3. Mais en fait le second cas recouvre le premier : si `first_ptr` est `NULL`, affecter la valeur `NULL` ou celle de `first_ptr`, c'est la même chose. Donc on n'a pas besoin de tester `first_ptr` pour séparer les cas, et on se contente de :

```
// allocation d'un nouvel élément
struct Element new_ptr = malloc(sizeof (struct Element));

// remplissage de l'élément
new_ptr->value         = 1234;       //
new_ptr->next_ptr      = first_ptr;  // null si premier ajout

// mise à jour du pointeur de début de liste
first_ptr              = new_ptr;
```

Ci-dessous une illustration

8.5 Pour retirer le premier élément

il faudra libérer le premier élément et mettre à jour le pointeur de début de liste. Mais attention à procéder dans le bon ordre

```
struct Element old_ptr = first_ptr;
first_ptr              = old_ptr->next_ptr;
free(old_ptr);
```

Pour bien comprendre, prenez le temps de faire un dessin avec toutes les étapes.

9 Réalisation d'une pile par chaînages

Avec ces opérations qui sont relativement simple on peut réaliser une "pile" (*stack*), une structure de données d'usage courant en programmation

9.1 Qu'est-ce que c'est ?

Pensez à des feuilles de papier posées les unes sur les autres. Vous pouvez

- regarder ce qui est écrit sur celle qui est au *sommet* de la pile
- poser une feuille dessus (*empiler*)
- enlever celle du dessus (*dépiler*)
- et aussi regarder si il reste des feuilles dans la pile (tester si la pile est vide)

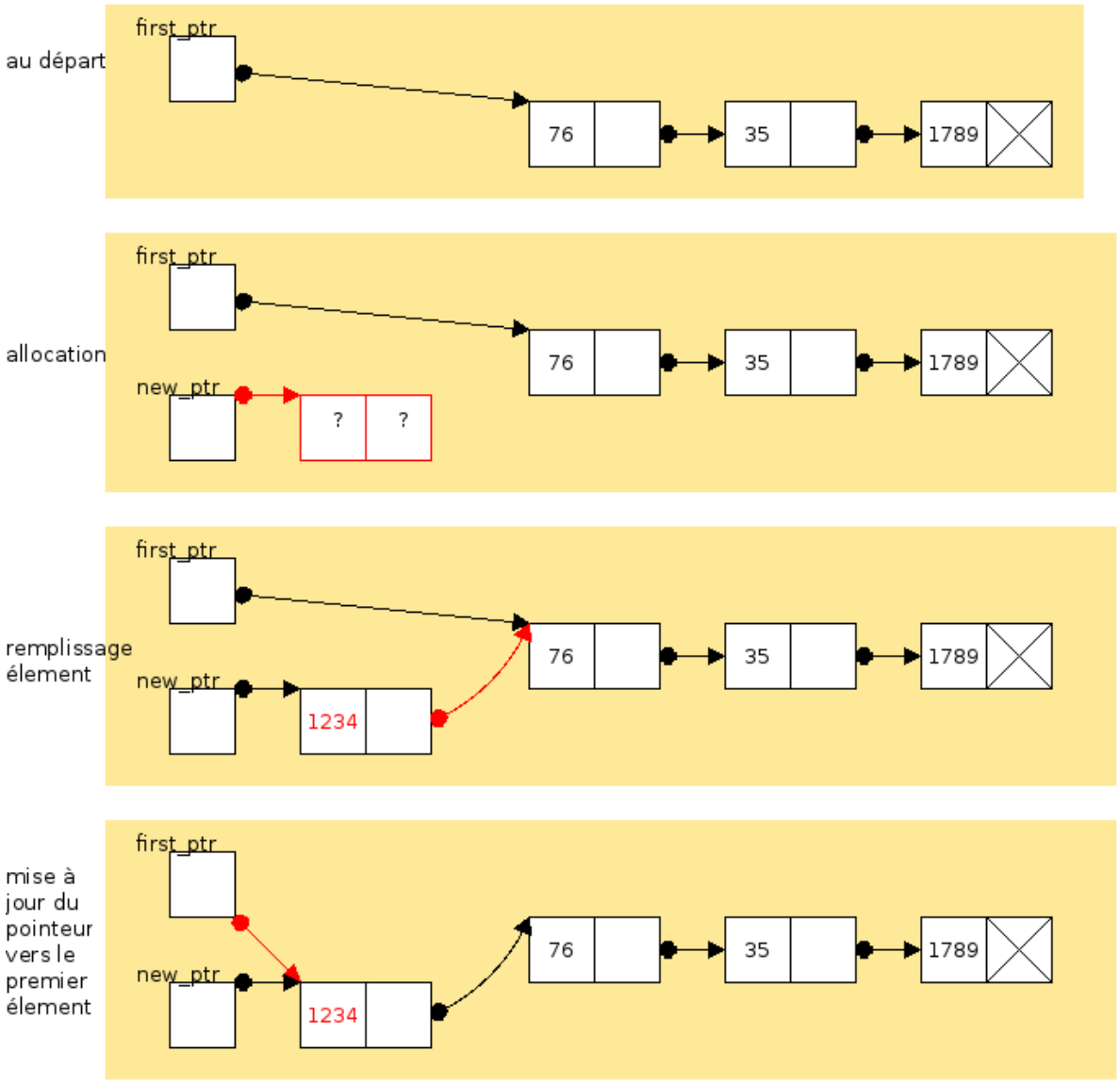

FIG. 7 : Ajout en tête

9.2 Quel rapport avec les chaînages ?

Les opérations qu'on a vues avant permettent de représenter facilement une pile par une **liste chaînée d'éléments** contenant des valeurs

- empiler : c'est ajouter au début,
- dépiler : c'est enlever le premier élément,
- test de pile vide : comparer le pointeur de sommet et `NULL`,
- la valeur du sommet de la pile est accessible par indirection de ce pointeur.

On utilisera les déclarations suivantes

```
struct Element {
    int              value;
    struct Element *next_ptr;
};

struct Stack {
    struct Element *top_ptr;
};
```

Le nom `top_ptr` a été choisi pour mieux refléter l'intention : c'est un pointeur (`ptr`) vers l'élément qui représente le sommet (`top`) de la pile.

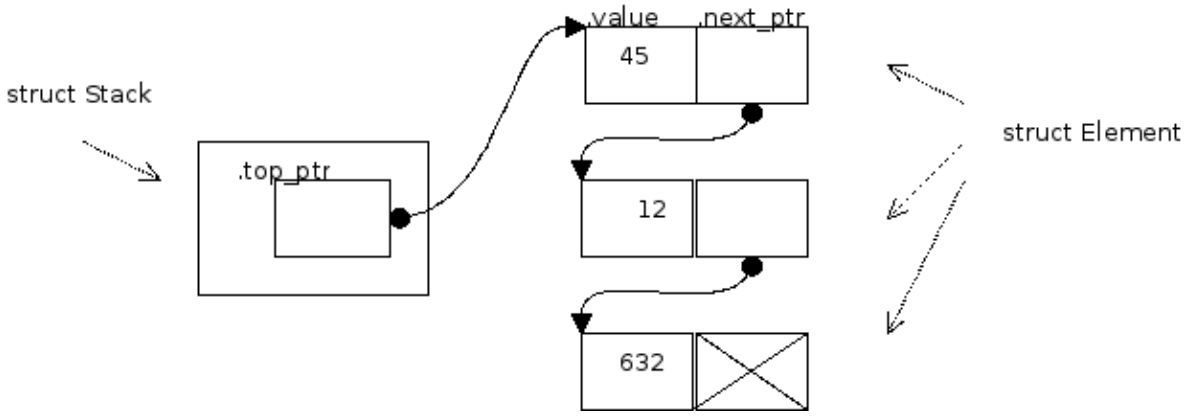

FIG. 8 : Une pile

9.3 Travail à faire

Voici du code de test, qui appelle des fonctions agissant sur des piles

```
void test_pile()
{
    struct Stack s;
    stack_init(& s);

    // empiler 10,20,30,40,50
    for (int i = 1; i <= 5; i++) {
        stack_push (&s, 10 * i);
    }

    printf("On devrait voir : 50 40 30 20 10\n");

    while ( ! stack_is_empty (&s)) {
        printf("%d ", stack_top(&s));
        stack_pop(&s);
```

```
    };
    printf("\n");

    stack_free(&s);
}
```

Vous avez compris ce qu'il vous reste à faire : écrire les fonctions "`stack_quelque_chose`" pour que le code affiche ce qui est attendu.

9.4 Limites pratiques de cette représentation

La représentation d'une pile par une liste chaînée a quelques inconvénients, elle est par exemple **extrêmement inefficace** pour représenter une "pile de caractères" (ou autres données de petite taille).

Il y a plusieurs raisons.

La place perdue

- Un `char` occupe 1 octet. Il faut lui ajouter un pointeur, qui occupe (sur une machine 64 bits) 8 octets.
- Qui plus est, la taille de l'`Element` est arrondie pour des contraintes techniques (alignement de données). Ce ne sont pas 9 octets, mais 16. Faites afficher `sizeof(struct Element)` pour voir.
- Et ce n'est pas fini : pour chaque allocation il y a quelques octets supplémentaires réservés pour la gestion des blocs occupés, la taille effectivement allouée est arrondie (par le haut) à un multiple de 32.

Une simple boucle permet de voir ce phénomène :

```
for (int i=0; i<5; i++) {
    printf("%p\n", malloc(1));  // un seul octet
}
```

qui affiche les adresses successives pour des allocations d'un octet chacune

```
0x168b010
0x168b030
0x168b050
0x168b070
0x168b090
```

Elles sont espacées de 32 octets (0x20). Une liste chaînée de caractères "gaspille" 31 octets pour chaque caractère à stocker !

La dispersion en mémoire

- au gré des allocations et libérations pendant l'exécution d'un programme, il apparaît des "trous" dans la zone mémoire réservée pour l'allocation dynamique,
- deux allocations successives (dans le temps) pourront se retrouver à des endroits distants,
- l'éparpillement des données a une influence désastreuse sur l'utilisation de *caches* de différents niveaux qui sont là pour accélérer l'accès à la mémoire,
- les performances d'un programme peuvent s'en trouver dégradées de façon importante.

Bref, pour stocker des données de petite taille (individuellement) mais nombreuses, il est en général nettement préférable de représenter une pile par un tableau extensible (voir plus haut).

10 Réalisation d'une file (Queue) par chaînage

Une petite adaptation permet de réaliser facilement une "file d'attente" ("*Queue*" en anglais) , qui fonctionne un peu différemment d'une pile. C'est aussi une structure très utile en programmation.

Les opérations :

- y mettre des éléments (enfiler),
- accéder à l'élément **le plus ancien** (pour une pile, c'était le plus récent),
- supprimer le plus ancien,
- et bien sûr tester si il en reste.

10.1 Structures de données

On utilise les dénominations "*back*" (pour l'arrière de la file, là où on met les éléments), et "*front*" (le premier de la file).

La représentation repose sur l'utilisation de deux pointeurs, un sur le premier élément de la liste, l'autre sur le dernier, ce qui facilite les ajouts à la fin.

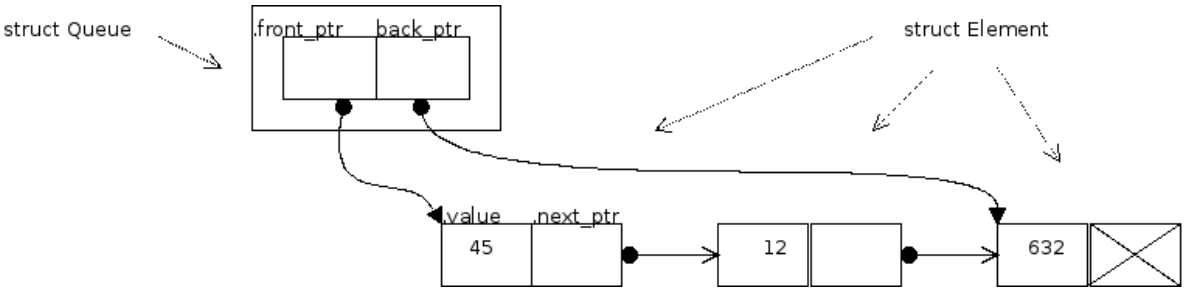

FIG. 9 : Une file

```
struct Element {                    // ça ne change pas
    int              value;
    struct Element *next_ptr;
};

struct Queue {
    struct Element *front_ptr,  *back_ptr;
};
```

10.2 Initialiser une file

Pour initialiser une file (vide) : mettre ses deux pointeurs à `NULL` :

```
void queue_init(struct Queue *queue_ptr)
{
    queue_ptr->front_ptr = NULL;
    queue_ptr->back_ptr  = NULL;
}
```

10.3 Pour connaître la valeur de l'élement de tête

```
int queue_front(struct Queue *queue_ptr)
{
    return queue_ptr->value;
}
```

ça ne marchera évidemment que si la liste n'est pas vide, il faudra avoir vérifié avant.

10.4 Pour tester si la pile est vide

Je vous laisse écrire la fonction `queue_is_empty`, qui retourne un `bool`.

10.5 Pour ajouter/enlever un élément

C'est la fonction `queue_push_back` qui s'en occupera.

Remarquez qu'il y a deux cas : si la liste est vide ou pas. Dans les deux cas, il faudra allouer un nouvel `Element` (qui deviendra le dernier).

Mais :

- si la liste était vide, cet élément devient aussi le premier
- si le liste n'était pas vide, il devient le suivant de l'ancien dernier

Illustration : effet de l'ajout d'une valeur -32 (à comparer au schéma précédent) dans une liste qui n'est pas vide.

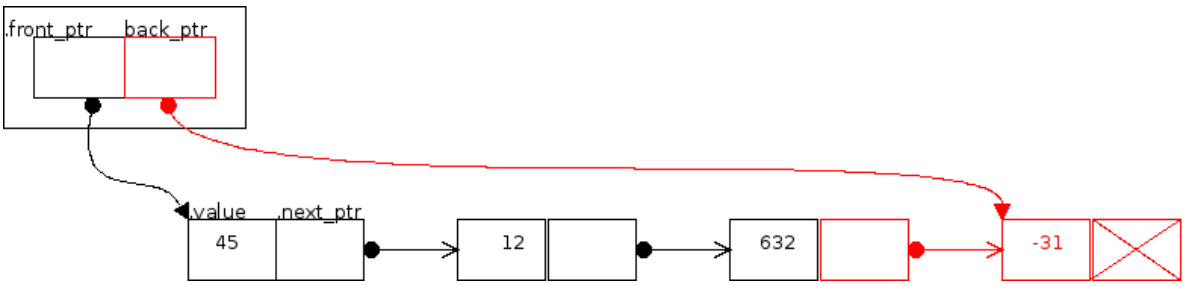

FIG. 10 : Ajout dans une file

À vous d'écrire la fonction, ainsi que `queue_remove_front`. Attention, il faudra aussi prévoir le cas où la liste devient vide.

Idée pour l'ajout : sachant que, dans un cas, il faut modifier le pointeur `front_ptr` de la file, dans l'autre le pointeur `next_ptr` du dernier élément, on peut noter l'adresse du pointeur à modifier dans un pointeur … de pointeur.

```
// qui devra-t-on modifier ?
struct Element  **pointer_ptr;          // adresse d'un pointeur

if (.....front_ptr == NULL) {           // si pointeur de tete nul
    pointer_ptr = &(......front_ptr);      // c'est lui qu'on modifiera
} else {
    pointer_ptr = &(......next_ptr);       // sinon, le suivant du dernier
}

// allocation et remplissage
struct Element new_ptr = malloc(....);
....

// "accrochage"
....next_ptr = new_ptr;
*pointer_ptr = new_ptr;               // modif du pointeur désigné.
```

10.6 Code de test

Écrivez les fonctions nécessaires pour que ce code tourne correctement.

```
void test_queue()
{
    struct Queue q;
    queue_init(& q);

    // ajouter 10,20,30,40,50
```

```
    for (int i = 1; i <= 5; i++) {
        queue_push_back (&s, 10 * i);
    }

    printf("On devrait voir : 10 20 30 40 50\n");

    while ( ! queue_is_empty (&s)) {
        printf("%d ", queue_front(&s));
        queue_remove_front(&s);
    };
    printf("\n");

    queue_free(&s);
}
```

10.7 Liste à double chaînage

Si on veut une structure de données où on ajoute/retire aux deux bouts (ce qu'on appelle une `dequeue` en anglais), on rend la situation symétrique avec, dans chaque élément, deux pointeurs : le précédent et le suivant.

```
struct Element {
    int             value;
    struct Element  *prev_ptr,    // précédent (previous)
                    *next_ptr;    // suivant
};

struct Deueue {
    struct Element *front_ptr,  *back_ptr;
};
```

Bien entendu, le précédent du premier, comme le suivant du dernier, ça sera `NULL`.

Je vous laisse faire un dessin, et écrire les opérations nécessaires

- `dequeue_init`,
- `dequeue_push_front`, `dequeue_push_back`,
- `dequeue_is_empty`,
- `dequeue_front`, `dequeue_back`,
- `dequeue_remove_front`, `dequeue_remove_back`,
- `dequeue_free`,

ainsi qu'un code de test.

11 Liste ordonnée (*PriorityQueue*)

Prenons l'exemple d'une structure "Agenda". Deux opérations principales :

- ajouter des tâches, avec une date de réalisation
- on veut connaître la première à effectuer

et bien sûr, enlever la première, tester si il en reste, etc.

Une façon de faire est de conserver ces tâches dans une liste simple, mais ordonnée par date croissante. Ainsi, il sera facile d'accéder à la première tâche à effectuer : c'est celle qui est en tête.

Note : il y a des structures de données plus efficaces que les listes pour faire ça, mais ça dépasse le cadre de ce document.

La difficulté ici sera d'insérer la nouvelle tâche au bon endroit dans la liste. Jusqu'ici c'était facile, on insérait toujours à un endroit connu : la tête ou la queue de liste. Là ça peut se passer au milieu. Et à

quel endroit exactement ? Ça va dépendre. Juste avant le premier évènement suivant. Ou à la fin.

Comme ça se gâte un peu, on va dire que ça devient *intéressant*. C'est l'occasion de voir comment procéder en douceur.

11.1 Un peu d'ordre dans les pensées

On simplifie le problème, et on regarde comment insérer un nombre (pourquoi pas 33) dans une liste ordonnée de nombres (en ordre croissant)

1. En revenant sur une réflexion précédente (à propos des files/queues), ce qu'on voit, c'est que l'insertion va conduire à modifier
   - soit le pointeur de début de liste
   - soit le pointeur "suivant" d'un des éléments existants.
2. Quand modifie-t-on le pointeur de début de liste ?
   - quand la liste est vide
   - quand elle n'est pas vide, et que son premier élément est plus grand que 33.
3. dans les autres cas, le nombre 33 sera ajouté comme successeur d'un autre élément de la liste. Mais lequel ? Évidemment, il sera mis **après le dernier élément inférieur** à 33.

11.2 A la recherche du dernier élément inférieur

Ça peut paraître compliqué de trouver le dernier élément inférieur, mais on peut procéder par étapes.

1. **Pour commencer** on se donne un bout de programme qui joue avec une liste chaînée de trois éléments (11, 22, 44), et qui fait un **parcours**

```
struct Element {
    int             value;
    struct Element *next_ptr;
};

void afficher_tous(struct Element *first_ptr)
{
    printf("Les éléments sont :\n\t");
    for (struct Element *p = first_ptr; p != NULL; p = p->next_ptr) {
        printf(" %d", p->value);
    }
    printf("\n");
}

int main()
{
    struct Element z = { 44, NULL};
    struct Element y = { 22, &z};
    struct Element x = { 11, &y };

    afficher_tous(& x);
    return 0;
}
```

La construction de la liste est un peu expéditive, elle ne fait pas appel à l'allocation dynamique, mais on a des éléments chaînes, c'est tout ce qui nous intéresse.

Ce qu'il faut voir, c'est la boucle `for`, qui passe en revue tous les éléments, vous la connaissiez déjà.

2. Maintenant, comment faire **afficher tous les éléments plus petits** qu'une certaine valeur ? Voila du code de test

```
    afficher_plus_petits_que(& x, 10);
    afficher_plus_petits_que(& x, 20);
    afficher_plus_petits_que(& x, 33);
    afficher_plus_petits_que(& x, 99);
```

C'est simple : dans la même boucle, on va sortir dès que la valeur limite est atteinte. Ça fera l'affaire parce que la liste est ordonnée : si une valeur dépasse, le reste de la liste aussi.

On utilise sans vergogne un `break`, qui, n'en déplaise aux esprits chagrins, est là pour ça :

```
void afficher_plus_petits_que(struct Element *first_ptr, int value)
{
    printf("Les éléments plus petits que %d sont :\n\t", value);
    for (struct Element *p = first_ptr; p != NULL; p = p->next_ptr) {
        if (p->value >= value) {                        // on a dépassé ?
            break;                                      // alors on arrête
        }
        printf(" %d", p->value);
    }
    printf("\n");
    }
```

remarquez que si il n'y en a pas, et bien ça n'affiche rien. Ça peut arriver, bien sûr.

3. Encore un petit effort, comment trouver **le dernier élément qui soit plus petit** que la valeur ?

L'idée va être de *noter* (dans un pointeur) la position du dernier élément plus petit rencontré jusque là.

Pour cela on définit un pointeur

- initialisé à `NULL` au départ (on n'a pas encore rencontré d'élément)
- mis à jour chaque fois qu'on rencontre un meilleur élément (dans la boucle)
- qu'on utilisera après la boucle.

```
void afficher_dernier_plus_petit(struct Element *first_ptr, int value)
{
    printf("Le dernier élément < à %d est :\n\t", value);

    struct Element *last = NULL;
    for (struct Element *p = first_ptr; p != NULL; p = p->next_ptr) {
        if (p->value >= value) {
            break;
        }
        last = p;
    }

    if (last != NULL) {                 // si on a trouvé un élement
        printf("%d", last->value);      // on affiche sa valeur
    }
    printf("\n");
}
```

11.3 Insertion

On en revient à notre problème : pour insérer un élément dans la liste

1. On alloue un nouvel `Element`.
2. On cherche l'adresse du dernier élément plus petit.
3. On accroche le nouvel élément
    - derrière (`next_ptr`) le dernier plus petit si il y en a un,
    - au début de la liste sinon (`first_ptr`).

12 Conclusion

La notion de **pointeur** est en fait très simple : une **variable contenant l'adresse** de quelque chose.

Mais elle est utilisée pour beaucoup de choses, ce qu'il fait qu'il y a une multitude de domaines connexes à étudier pour prétendre qu'on sait "programmer en C avec des pointeurs".

L'**allocation dynamique** est un mécanisme fondamental pour le développement de la plupart des applications. En elle-même rien de compliqué non plus : c'est juste demander au système d'exploitation de **réserver** un certain nombre d'octets, et évidemment de nous dire où ils se trouvent (en retournant un pointeur). Et les **libérer** quand on n'en n'a plus besoin. La difficulté pratique est de faire la libération au bon moment : pas trop tôt (libération prématurée alors qu'on en aura encore besoin), pas trop tard (parce que ça fera une fuite mémoire), et pas deux fois…

Ce qui est plus compliqué, c'est d'arriver à s'en servir pour **réaliser des structures de données** qui peuvent effectuer un certain nombre d'opérations (pour un conteneur : ajouter des éléments, etc). Et la réalisation de ces opérations nécessite des compétences algorithmiques.

Bien entendu, les structures de données ne se limitent pas aux tableaux extensibles et aux listes chaînées (ou doublement chaînées) que j'ai choisi pour illustrer l'usage des pointeurs, il y en a beaucoup d'autres (arbres, graphes, etc).

En espérant que ce document vous sera utile !