Pointeurs, tableaux, structures

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

1 Pointeurs et tableaux

La mémoire d'un programme est une immense liste de bits, regroupés par octets. En C, lorsque l'on déclare une variable, une zone mémoire lui est réservée pour stocker sa valeur. Cette zone sera composée d'un certain nombre d'octets consécutifs, et le nombre dépendra du type de la variable. Par exemple, pour un `unsigned char`, qui fait 8 bits, un seul octet sera réservé. Cependant pour un `int`, qui fait 32 bits, quatre octets seront réservés. En C, l'opérateur sizeof permet d'obtenir la taille d'un type, en octet :

```
1 int taille_int = sizeof(int); // vaudra 4
2 int taille_char = sizeof(char); // vaudra 1
```

L'adresse d'une variable est l'adresse du premier octet de sa zone mémoire. En C, on parle de ***pointeurs***. Un pointeur est donc une valeur entière, comprise entre 0 et l'adresse maximale de l'ordinateur. Plutôt que d'utiliser un type comme `unsigned int` pour représenter les pointeurs, on utilise des types dédiés.

Si une variable `x` est de type `t`, alors son adresse s'obtient avec `&x`, et est une valeur de type `t*`. On parle de type pointeur. Tous les types pointeurs font la même taille : 64 bits. Les adresses mémoires peuvent donc aller de 0 à $2^{64} - 1$.

Exemple 1. Dessinons l'état de la mémoire lorsque l'on exécute les instructions suivantes. On suppose pour le moment que les adresses mémoires commencent à 1000, et que les variables sont stockées dans l'ordre où elles sont déclarées.

```
1 unsigned int n = 37;
2 char c = 6;
3 unsigned int* p = &n;
```

On peut donc manipuler les pointeurs comme des valeurs habituelles en C. Pour obtenir la valeur stockée à l'adresse d'un pointeur `p`, on utilise `*p` : c'est l'opérateur de ***déréférencement*** :

```
1 int x = 56;
2 int * ptr_x = &x; // ptr_x vaut l'adresse de x
3 int y = *ptr_x;
4 assert(x == y);
```

En pratique, les adresses mémoires des variables d'un programme ne commencent pas à 0, ni même à une valeur privilégiée. D'une exécution à l'autre, si l'on affiche l'adresse des variables d'un programme, on observera des valeurs totalement différentes.

Exemple 2. Compilez le code suivant, puis exécutez-le plusieurs fois de suite : vous verrez des adresses différentes à chaque exécution :

```
1 #include <stdio.h>
2 int main(){
3   int x;
4   int y;
5   printf("Adresse de x: %p\nAdresse de y: %p\n", &x, &y);
6 }
```

En revanche, les adresses de x et y semblent toujours séparées de 4 octets : elles sont donc stockées l'une à côté de l'autre.

Passage par référence On rappelle qu'en C, les fonctions marchent par passage par valeur, et copient les valeurs de leurs arguments, si bien qu'une fonction C ne peut pas modifier ses arguments. Les pointeurs permettent de simuler le passage par référence ! Par exemple, considérons les deux codes suivants :

```c
#include <stdio.h>

void ajouter_un(int* ptr){
  *ptr += 1;
  // instant B
  return;
}

int main(){
  int x = 5;
  // instant A
  ajouter_un(&x);
  // instant C
  printf("%d\n", x);
  return 0;
}
```

```c
#include <stdio.h>

void ajouter_un(int y){
  y += 1;
  // instant B
  return;
}

int main(){
  int x = 5;
  // instant A
  ajouter_un(&x);
  // instant C
  printf("%d\n", x);
  return 0;
}
```

Traçons le schéma de la mémoire au cours de l'exécution de chacun des deux programmes, aux instants A, B et C. On représentera les différentes variables par des boîtes dans lesquelles on écrira leurs valeurs. On regroupera les variables par fonction, et lorsqu'une variable `p` a comme valeur l'adresse d'une autre variable `&x`, on dessinera une flèche de p vers x.
Conseil : Dessinez sur une feuille à part et collez-la ici ou dans vos notes de cours.

Le programme de gauche va bien afficher 6, car la valeur de `x` aura été modifiée. Le programme de droite va afficher 5 car la valeur de `x` aura été dupliquée lors de l'appel de fonction, et c'est la variable locale `y` qui aura été modifiée.

Attention, on **simule** un passage par référence, mais c'est bien un passage par valeur, car l'argument de la fonction n'est pas `x` mais la valeur `&x` ! L'argument de la fonction n'est donc pas modifié, en revanche la zone mémoire pointée par l'argument de la fonction, elle, a changé.

La valeur `NULL` est une valeur particulière, appelée **pointeur nul**, qui pointe toujours vers une case invalide de la mémoire (en pratique c'est toujours la case 0).
Lorsque l'on essaie d'accéder à une adresse mémoire que le programme n'a pas réservé, l'ordinateur empêche l'opération et arrête immédiatement le programme, levant une ***erreur de segmentation***, ou ***segmentation fault***. Cette erreur est une des plus courantes lorsque l'on apprend à coder en C, elle signifie que l'on essaie d'accéder à une zone mémoire non-autorisée.

Pour bien avoir l'habitude des pointeurs, il est utile de s'entraîner à exécuter à la main du code C manipulant des pointeurs, en représentant les différentes cases mémoires du programme (i.e. les différentes variables) et en notant leurs valeurs successives.

Exercice 1.

Question 1. En simulant à la main l'exécution du code suivant, vérifier que la fonction `echange` permet bien d'échanger deux variables :

```c
#include <assert.h>
// échange le contenu des entiers pointés par p et q
void echange(int* p, int* q){
  int temp = *p;
  *p = *q;
  *q = temp;
}

int main(){
  int x = 3, y = 5;
  echange(&x, &y);
  assert(x == 5 && y == 3);
}
```

Question 2. En l'exécutant à la main, qu'affiche le programme suivant ?

```c
#include <stdio.h>
void f(int* p, int** l, int x){
  x++;
  *l = p;
  *p = x;
}

void g(int* p, int* q){
  *p = *q + 1;
  *q = *p + 1;
}

int main(){
  int x = 3, y = 5, z = 8;
  int* px = &x;
  f(&y, &px , z);
  g(&x, px);
  printf("%d %d %d %d\n", x, y, z, *px);
}
```

A Tableaux

Un ***tableau*** de type `T` et de taille n est constitué de n cases mémoires, toutes de la même taille : la taille du type `T`.

On peut déclarer un tableau avec la syntaxe suivante :

```
TYPE NOM_VARIABLE[TAILLE];
```

Par exemple :

```
int mon_tableau[4];
```

Cela aura pour effet de réserver 4 cases mémoires de 4 octets chacune, soit 16 octets contigus en mémoire. On peut ensuite accéder aux cases d'un tableau pour y lire et y écrire avec la syntaxe `tab[i]` avec i allant de 0 à (taille du tableau) - 1. Par exemple :

```
int tab[4];
tab[0] = 5;
tab[3] = tab[0] + 12;
```

Un tableau défini avec la syntaxe ci-dessus se comporte presque comme un pointeur, on peut imaginer qu'un tableau est en fait un pointeur vers sa première case. Par exemple, on peut manipuler les tableaux avec des fonctions en utilisant les types pointeurs :

Exemple 3. La fonction suivante prend en entrée deux tableaux de flottants *src* et *dst*, ainsi qu'un entier n supposé représenter leur taille, et recopie les données de *src* (la source) dans *dst* (la destination) :

```
void recopier(float* src, float* dst, int n){
  for (int i = 0; i < n; i++){
    dst[i] = src[i];
  }
}
```

Ainsi, si l'on dispose juste d'un tableau t, rien ne permet a priori d'obtenir sa longueur. En particulier, il n'y a aucun mécanisme du C qui empêche d'écrire au delà de la mémoire réservée. En pratique cependant, si l'on dépasse trop, on tombe sur une zone mémoire à laquelle on n'a pas le droit d'accéder, ce qui cause une **erreur de segmentation**.

On peut initialiser un tableau directement avec des valeurs :

```
int tab[6] = {12, 63, 268};
```

Attention, ceci ne veut pas dire "tab reçoit la valeur $\{12, 13, 268\}$" ! C'est un raccourci syntaxique pour écrire :

```
int tab[6];
tab[0] = 12;
tab[1] = 63;
tab[2] = 268;
```

On ne peut pas mettre de tableau de la forme `{x_1, ..., x_k}` autre part dans du code que dans le raccourci syntaxique précédent. Par exemple le code suivant est invalide :

```
int tab[5];
tab = {2, 3, 4, 7, 8}; // CA NE COMPILE PAS !
```

Enfin, en C, on ne doit **jamais** déclarer un tableau dont la taille est autre chose qu'une constante numérique, par exemple une variable. C'est accepté par C (et encore, pas pour tous les compilateurs) mais **interdit** par le programme officiel de MP2I, et à raison ! Nous verrons pourquoi dans quelques paragraphes.

Par exemple, on n'écrira jamais :

```c
int n;
scanf("%d", &n);
float tab[n]; // INTERDIT
```

ni même

```c
int n = 5;
float tab[n]; // INTERDIT
```

En revanche, on peut utiliser un tableau de 100000 cases, mais n'en utiliser que les `n` premières (voir TP3) :

```c
float tab[100000]; // Autorisé :)
scanf("%d"n &n);
for (int i = 0; i < n; i++){
  tab[i] = 0;
}
```

2 Mémoire d'un programme

Dans un ordinateur, il y a deux types de mémoire. La mémoire non-volatile correspond au disque dur, c'est là où sont stockés vos fichiers, l'ordre de grandeur est de quelques centaines à quelques milliers de Giga-octets sur les ordinateurs courants. La mémoire volatile correspond à la RAM, c'est là que sont stockées les informations locales des programmes exécutés par l'ordinateur. Son ordre de grandeur est de quelques Giga-octets à quelques dizaines de Giga-octets sur les ordinateurs courants.

Le terme RAM signifie "Random Access Memory", et ici le mot "random" ne veut pas dire aléatoire mais **au choix**. La RAM est donc une mémoire dans laquelle on peut accéder à n'importe quelle donnée au choix rapidement. Au contraire, dans un disque dur, la lecture ou écriture d'une donnée est moins rapide, et la vitesse d'exécution dépend de nombreux facteurs. La RAM est donc utile pour exécuter un programme de manière efficace, mais lorsqu'un programme se termine, la zone mémoire qu'il avait réservé en RAM est libérée : c'est une mémoire **volatile**.

Lorsque l'on lance un programme sur notre ordinateur, le système réserve une zone dans la RAM pour ce programme. La zone mémoire d'un programme est divisée en plusieurs zones, dont les trois principales sont la zone de **texte**, la **pile d'appel** et le **tas**.

La zone de texte contient simplement le code du programme, c'est à dire la suite des instructions à exécuter. Attention, même si elle s'appelle zone de texte, elle contient majoritairement du code machine, illisible par les humains !

La pile et le tas sont deux zones de tailles variables, qui grandissent et rapetissent au fil de l'exécution du programme. En général, le tas est situé au début de la mémoire et grandit vers les adresses croissantes, tandis que la pile est située à la fin de la mémoire et grandit vers les adresses décroissantes.

A Pile d'appel

La pile d'appel, aussi appelée **pile**, sert à stocker les variables locales des fonctions. Au lancement du programme, on ne sait pas de combien d'espace mémoire on aura besoin, car cela dépend de l'exécution. En revanche, lorsque l'on appelle une fonction, on sait **exactement** combien d'octets sont nécessaires : il suffit de faire la somme des tailles de ses différentes variables. On peut même le compter lors de la compilation, simplement en lisant le code sans l'exécuter. Par exemple :

```
1 void f(int x){
2   double t[3] = {0.3, -2.0, 3.65}; // double = type flottant 64 bits
3   int* p = &x;
4 }
```

On a besoin de $4 + (3 \times 8) + 8 = 36$ octets pour stocker respectivement x, t et p. Le compilateur peut donc dire : "Lorsque la fonction f est appelée, il faut réserver 36 octets".

Le système peut donc réserver exactement la bonne taille en mémoire à chaque appel de fonction. Les blocs mémoire alloués aux différents appels sont regroupés dans la pile d'appel,

et sont stockés les uns au dessus des autres.
Au départ, la pile est vide. Lorsque l'on appelle une fonction, on **empile** ses informations au dessus de la pile. Lorsque l'on arrive à une instruction return dans cette fonction, on **dépile** les informations. Une zone ainsi créée pour l'appel d'une fonction s'appelle une **stack frame**, ou "cadre de pile" .
Plus précisément, dans une stack frame, on trouve :

— Les variables locales de la fonction, y compris les paramètres ;

— L'adresse de retour, c'est à dire la ligne d'instruction à exécuter une fois que l'on retourne de la fonction ;

— D'autres informations, auxquelles on ne s'intéressera pas dans ce cours (si vous vous y connaissez un peu en architecture des ordinateurs : on y sauvegarde aussi les valeurs de certains registres du processeur).

Exemple 4. Regardons quelques programmes C, exécutons-les, et dessinons l'état de la pile à divers instants. On séparera les stack frames par des traits épais, et on fera figurer dans chaque stack frame la valeur actuelle de chaque variable. Lorsqu'une variable est un pointeur, plutôt que d'écrire la valeur de l'adresse vers laquelle elle pointe, on pourra dessiner une flèche vers cette adresse.

1.

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

int pgcd(int x, int y){
  int t;
  while (y > 0){
    t = y;
    y = x%y;
    x = t;
  }
  return x;
}

bool premiers_entre_eux(int a, int b){
  bool p = pgcd(a, b);
  return (p == 1);
}

int main(){
  int x = 18, y = 25;
  if (premiers_entre_eux(x, y)){
    printf("Oui !\n");
  } else {
    printf("Non...\n");
  }

  return 0;
}
```

2.

```c
void incrementer(int* p){
  // C
  int a = *p;
  a = a + 1;
  *p = a;
  // D
}

int main(){
  // A
  int x = 3;
  int* px = &x;
  // B
  incrementer(px);
  // E
  return 0;
}
```

3. Attention, il n'y a pas une stack frame par fonction mais une par **appel** de fonction. En particulier dans le cas de fonctions récursives, les différents appels viennent **s'empiler** :

```c
#include <stdbool.h>

int fact(int n){
  // C
  if (n <= 0){
    return 1;
  }
  return n * (fact(n-1));
}

bool fact_depasse_mille(int n){
  // B
  int r = fact(n);
  return (r >= 1000);
}

int main(){
  int x = 3;
  bool b;
  // A
  b = fact_depasse_mille(x);
  // D
  return 0;
}
```

Lorsqu'un appel de fonction se termine, sa stack frame est retirée de la pile, et les données qui y étaient écrites deviennent invalides. On ne peut pas accéder à une stack frame ayant été dépilée : si l'on essaie, le programme affiche une erreur et s'arrête. En particulier, si une fonction renvoie un pointeur vers une de ses variables locales, ce pointeur pointera vers une zone invalide de la mémoire et ne servira donc à rien.

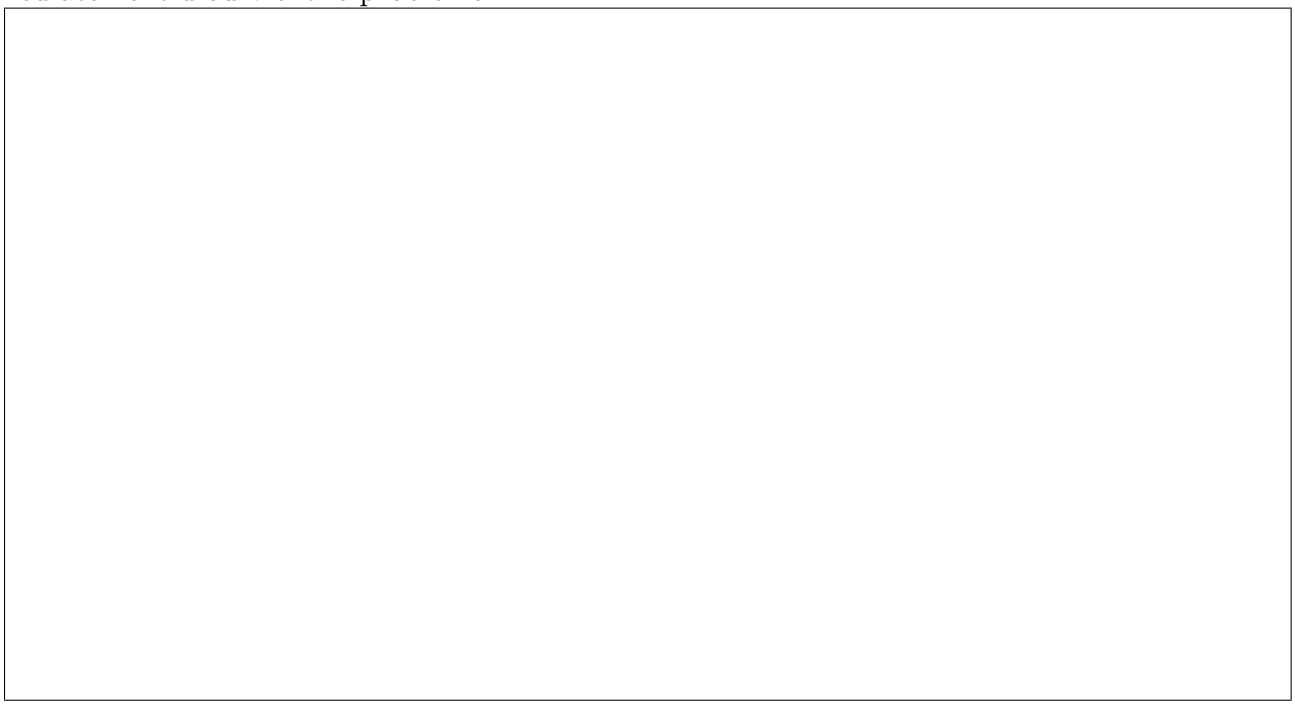
Exemple 5. Si l'on exécute le code suivant :

```
1  int* pointeur_vers_3(){
2    int x = 3;
3    return &x;
4  }
5
6  int main(){
7    int* p = pointeur_vers_3();
8    int a = *p;
9    return 0;
10 }
```

D'une part, à la compilation, on a un avertissement :

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

Ensuite, à l'exécution, le programme rencontre une **erreur de segmentation**, ou **segmentation fault** ! Si l'on dessine la pile d'appel du programme au fil de l'exécution, on voit immédiatement d'où vient le problème :



le pointeur renvoyé par la fonction `pointeur_vers_3` pointe vers une case de la pile qui n'est plus sensée exister. Donc l'instruction `int a = *p;`, qui tente de déréférencer ce pointeur, va aller lire dans une zone invalide de la mémoire.

Pour la même raison, une fonction ne peut pas renvoyer de tableau.

Néanmoins, l'ordinateur ne s'embête pas à effacer physiquement les stack frames dépilées. Après le retour d'un appel de fonction, les données de sa stack frame sont donc toujours présentes dans la mémoire, mais au dessus du sommet de pile. L'ordinateur garde toujours en mémoire une valeur appelée le **stack pointer**, qui est l'adresse mémoire actuelle du sommet de la pile, et peut donc déterminer facilement lorsque l'on essaie d'accéder à une mauvaise zone mémoire.
Notons que l'on peut exploiter le fait que les données des stack frames ne sont pas effacées pour écrire des programmes au comportement étranges !

Exemple 6. Dans le code suivant la fonction `affiche` affiche une variable locale y qui n'est pas définie. Cependant, on peut aller écrire dans la mémoire à l'endroit où y sera stocké et choisir ce qui sera affiché :

```c
#include <stdio.h>
void f(){
  // l'appel de cette fonction écrit 5 dans une case mémoire
  // juste au dessus de la stack frame du main
  int x = 5;
}

void g(){
  // l'appel de cette fonction écrit 162 dans une case mémoire
  // juste au dessus de la stack frame du main
  int bla = 162;
}

void affiche(){
  // la stack frame pour `affiche` est juste au dessus du main,
  // et donc la variable `y` sera stockée au même endroit que `x`
  // et `bla` l'étaient. Si l'on a appelé f() avant affiche(),
  // y vaudra 5 à la création de la stack frame. Si l'on
  // a appelé g(), y vaudra 162.
  int y;
  printf("%d\n", y);
}

int main(){
  affiche();
  f();
  affiche();
  g();
  affiche();
  return 0;
}
```

Simulons l'exécution de ce programme, et dessinons l'état de la pile d'appel au fil de l'exécution. Plutôt que de ne pas dessiner les stack frames ayant été dépilées, nous allons représenter toute la pile, y compris les zones invalides au dessus du stack pointer.

Notons que dans les programme que l'on écrira cette année, on n'utilisera jamais ce mécanisme : lorsque l'on crée une variable sans valeur initiale, on doit considérer que sa valeur est totalement aléatoire et indéterminée.

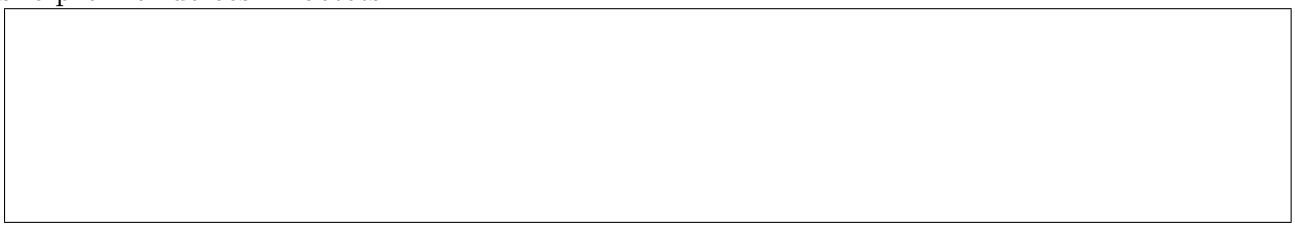
B Tas : allocation dynamique

La pile est donc une zone mémoire éphémère, où les données disparaissent une fois que la fonction qui les ont créées se termine. Le tas, au contraire, est une zone mémoire **permanente**. Cette zone peut donc être utilisée pour stocker des données qui continuent à exister au delà d'un appel de fonction. Cependant, contrairement à la pile, il n'y a aucun mécanisme qui efface automatiquement le tas. La mémoire du tas doit donc être gérée par la personne qui programme (vous), **à la main**.

Pour réserver une zone mémoire dans le tas, on utilise la fonction ***malloc***. Cette fonction prend un entrée un entier X et a pour effet de réserver X octets dans le tas, et de renvoyer un pointeur vers la zone créée. Par exemple, pour réserver 5 cases mémoire de type `int` dans le tas, on écrit :

```
1 int* tab = malloc(5 * sizeof(int));
```

En mémoire, on aura donc $5 \times 4 = 20$ octets réservés dans le tas, et la variable `tab`, qui est stockée dans la pile comme toutes les variables locales, contiendra un pointeur vers le tas, vers le premier de ces X octets.



Écrire `int* tab = malloc(5 * sizeof(int))` a presque le même effet que d'écrire `int tab[5]`, les deux instructions créent un tableau de 5 cases. Seulement, le tableau créé avec malloc est réservé dans le tas, et est donc permanent : il existe jusqu'à ce qu'on le libère à la main. L'autre tableau est stocké dans la pile, et existe temporairement, jusqu'à ce que la stack frame où il a été créé disparaisse.

Exemple 7. Simulons l'exécution du programme suivant, et dessinons l'état de la pile et du tas à plusieurs instants :

```
1  /* Renvoie un tableau de taille n contenant 0, 1, ..., n-1 */
2  int* tableau_croissant(int n){
3    // A
4    int* t = malloc(n * sizeof(int));
5    for (int i = 0; i < n; i++){
6      t[i] = i;
7    }
8    // B
9    return t;
10 }
11
12
13 int main(){
14   int* t = tableau_croissant(5);
15   // C
16   return 0;
17 }
```

Notons qu'avec malloc, on ne fait plus la différence entre les pointeurs et les tableaux : on réserve un certain nombre de cases mémoires, et on dispose d'un pointeur vers la première. En particulier, on peut ne réserver qu'une seule case mémoire.

Exemple 8. Considérons les deux codes suivants, et simulons leurs exécutions en représentant l'état de la pile et du tas à divers instants :

```c
/* Réserve une case mémoire, y
   écrit x, et renvoie l'adresse
   de la nouvelle case.
   (INVALIDE) */
int* ptr_pile(int q){
  int a = x;
  // B
  return &a;
}

int main(){
  int x = 7;
  // A
  int* q = ptr_tas(x);
  // C
  return 0;
}
```

```c
/* Réserve une case mémoire, y
   écrit x, et renvoie l'adresse
   de la nouvelle case.
   (VALIDE) */
int* ptr_tas(int x){
  int* p = malloc(sizeof(int));
  *p = x;
  // B
  return p;
}

int main(){
  int x = 7;
  // A
  int* q = ptr_tas(x);
  // C
  return 0;
}
```

Remarquons qu'avec malloc, on peut réserver des tableaux de taille variable, par exemple :

```c
int n = 3570;
float* tab = malloc(n*sizeof(float));
```

est autorisé mais pas

```c
int n = 3570;
float tab[n];
```

La raison est que dans le deuxième cas, on réserve un tableau dans la pile **sans en connaître la taille au moment de la compilation**, car c'est seulement à l'exécution que l'ordinateur verra qu'il faut réserver 3570 places. A cause de cela, au moment de la création de la stack frame contenant cette déclaration, le système ne saura pas combien de mémoire réserver. En pratique, GCC sait traiter ce genre de déclarations, mais certains autres compilateurs **interdisent** cette pratique.

Exercice 2. Le code suivant lève-t'il une erreur d'assertion ?

```c
int* p = malloc(sizeof(int));
*p = 3;
int x = *p;
assert(&x == p);
```

C Désallocation

Considérons le code suivant :

```
1  /* Affiche "Message: coucou" */
2  void afficher_message(){
3    char* msg = malloc(7*sizeof(char));
4    strcpy(msg, "coucou"); // écrire "coucou" dans msg
5    printf("Message: %s\n", msg);
6  }
7
8  int main(){
9    while (true){
10     afficher_message();
11   }
12 }
```

Si l'on simule ce programme et que l'on dessine l'état du tas après chaque appel de la fonction `afficher_message`, on se rend compte que les différentes zones mémoires réservées avec malloc ne sont jamais libérées, le tas va donc sans cesse grandir, jusqu'à remplir l'intégralité de la RAM, après quoi malloc ne pourra plus trouver d'endroit où réserver de la mémoire, ce qui fera bugger le programme.
On appelle ce type de problème une **fuite mémoire**, elle survient lorsque l'on alloue de la mémoire dans une fonction sans la libérer par la suite.
Le tas est donc une zone mémoire permanente, il faut faire attention à **TOUJOURS** libérer la mémoire que l'on réserve avec malloc. Les fuites mémoires peuvent être la cause de nombreux bugs, et sont difficiles à détecter car elles peuvent ne commencer à causer des problèmes qu'après un certain temps d'utilisation.

Exemple 9. Les jeux Pokémon Violet et Écarlate ont des problèmes de fuite mémoire : plus vous y jouez sans fermer et relancer le jeu, plus la mémoire RAM de votre console est remplie par des données allouées sans être libérées, et plus le jeu devient lent !

Pour libérer une zone mémoire, on utilise la fonction `free`. Cette fonction prend en entrée un pointeur, qui est supposé avoir été obtenu avec malloc, et libère la mémoire qui avait été réservée. Par exemple :

```
1  int main(){
2    int* t = malloc(10*sizeof(int));
3    for (int i = 0; i < 10; i++){
4        t[i] = i;
5    }
6    free(t);
7  }
```

Pas besoin de libérer chaque case de t individuellement !

En pratique, on utilisera presque toujours malloc pour créer des tableaux, et jamais la syntaxe `int t[100];`.

Dans un code, lorsque l'on écrit un malloc, la **PREMIERE** chose à faire est d'écrire le free correspondant : c'est comme un jeu de parenthèses : à chaque parenthèse ouvrante doit correspondre une parenthèse fermante.
Si, dans une fonction, on malloc un pointeur qui doit être renvoyé, alors il ne faut pas le free. En revanche, si une fonction renvoie un pointeur/tableau créé par malloc, c'est l'appel de cette fonction qui devient la "parenthèse ouvrante". Par exemple :

```
1  /* renvoie un tableau de n cases contenant les n premiers carrés parfaits
2     jusqu'à (n-1)^2 */
3  int* carres(int n){
4    int* res = malloc(n * sizeof(int));
5    for (int i = 0; i < n; i++){
6      res[i] = i*i;
7    }
8    return res; // comme on renvoie res, pas besoin de le free avant,
9  }
10
11 int main(){
12   int* tab_carres = carres(100); // allocation
13   ...
14   free(tab_carres);          // désallocation
15 }
```

A retenir

La pile et le tas sont deux zones mémoires servant à stocker les données du programme. Lorsque plusieurs programmes tournent en même temps sur l'ordinateur, chacun a son tas et sa pile.

- La pile contient les données des **appels de fonctions**, notamment leurs variables locales. Les données y sont allouées et libérées **automatiquement** au fil des appels et des retours de fonctions, et sont donc **éphémères**. La pile est constituée de **stack frames**, chaque stack frame correspondant à un appel de fonction. Ces stack frames viennent s'empiler au sommet de la pile lors des appels, et se dépilent lors des retours de fonction. Lors de la compilation, le compilateur compte les variables locales des différentes fonctions, et en déduit les tailles des stack frames.
- Le tas contient des données **permanentes**, on doit y réserver et libérer la mémoire **manuellement** à l'aide des fonctions `malloc` et `free`. Lorsque l'on réserve de la mémoire, il faut systématiquement la libérer après son utilisation. Lorsque l'on perd l'accès à une zone du tas réservée mais que l'on ne l'a jamais libérée, par exemple si l'on sort d'une fonction ayant utilisé malloc, on dit qu'il y a une **fuite mémoire**.

3 Codage du texte, chaines de caractères

Le type `char` sert en réalité à représenter du texte, en associant à chaque symbole un nombre entier entre 0 et 127. Par exemple :

- Les lettres majuscules `'A'` - `'Z'` sont représentées par les entiers 65 - 90 (soit 0x41 - 0x5a)
- Les lettres minuscules `'a'` - `'z'` sont représentées par les entiers 97 - 122 (soit 0x61 - 0x7a)
- Les chiffres `'0'` - `'9'` sont représentés par les entiers 48 - 57 (soit 0x30 - 0x39)

Ainsi, en C, lorsque l'on écrit `'a'`, c'est ***exactement*** comme si l'on écrivait `97` ou `0x41`.

En C, un texte est donc représenté par une ***chaîne de caractères***, c'est à dire un tableau de caractères. Toutes les chaînes de caractère finissent par le caractère 0x00, appelé caractère nul. On peut également l'écrire `'\0'`. Ce caractère sert uniquement à indiquer la fin de chaîne. Par exemple, si l'on écrit :

```
1 char s[10] = "Hello";
```

alors `s` contiendra 10 cases, dont 6 sont réellement nécessaires : les 5 premières stockent les 5 lettres du mot `Hello`, c'est à dire les octets `0x48 0x65 0x6c 0x6c 0x6f`, et la 6ème case, d'indice 5, contiendra le caractère nul, c'est à dire l'octet `0x00`.

Lorsque l'on crée une chaîne de caractère comme ci-dessus, le caractère nul est automatiquement ajouté. En revanche, si l'on crée une chaîne de caractères en y écrivant les caractères individuellement, il faut toujours se rappeler de mettre le caractère nul.

Exemple 10. Écrivons une fonction qui crée et renvoie une chaîne de caractères contenant n fois la séquence *ba*. Il faudra donc allouer $2n + 1$ octets pour stocker les $2n$ lettres ainsi que le caractère nul :

```
1 char* baba(int n){
2   char* res = malloc((2*n +1) * sizeof(char));
3   for (int i = 0; i < n; i++){
4     res[2*i]   = 'b';
5     res[2*i+1] = 'a';
6   }
7   res[2*n] = '\0';
8   return res;
9 }
```

A Librairie `<string.h>`

La librairie `string.h` contient de nombreuses fonctions utiles pour manipuler des strings. Les plus importantes à savoir utiliser sont :

— `int strlen(char* s)` renvoie la longueur d'une chaîne de caractères, sans compter le caractère nul. Par exemple, `strlen("Bonjour")` vaudra 7.

— `void strcpy(char* dst, char* src)` copie le contenu de la chaîne source vers le pointeur destination, y compris le caractère nul. Il faut que le pointeur `dst` pointe vers une zone ayant assez d'espace pour stocker le résultat. Autrement dit, cette fonction ne va pas réserver de mémoire à notre place.

— `void strcat(char* dst, char* src` va concaténer la chaîne source après la chaîne destination. Plus précisément, elle cherche le caractère nul de la chaîne destination, et y recopie la chaîne source à partir de cet octet. Par exemple :

```
1 char s[40] = "bonne nuit les ";
2 char t[7]  = "petits";
3 strcat(s, t);
4 printf("%d\n", s); // affiche "bonne nuit les petits"
```

Comme pour `strcpy`, le caractère nul est aussi recopié, de telle sorte que si l'on appelle `strcat` sur deux chaînes bien formées, la chaîne résultante l'est aussi.

— `int strcmp(char* s1, char* s2)` compare les chaînes s_1 et s_2 selon l'ordre lexicographique, c'est à dire en comparant d'abord la première lettre de chacune, puis la deuxième, etc... C'est l'ordre du dictionnaire. Le résultat renvoyé est 0 si les chaînes contiennent les mêmes caractères, -1 si s_1 précède s_2, et 1 si s_2 précède s_1 :

```
1 assert(strcmp("bonjour", "bonjour") == 0);
2 assert(strcmp("bateau", "babord") > 0);
3 assert(strcmp("babord", "bateau") < 0);
```

En particulier, si on a deux chaînes de caractères `s1` et `s2`, on ne peut pas tester leur égalité avec `==` ! En fait, `s1 == s2` teste l'égalité des adresses mémoires. Par exemple :

```
1 char* s = "bonjour";
2 char* t = "bonjour";
3 char* u = s;
4 assert (u == s);
5 // on ne peut pas a priori faire assert(s == t)
6 // car les deux chaînes ne partagent pas a priori
7 // la même adresse mémoire.
```

4 Tableaux 2D

Il est courant de faire des tableaux à deux dimensions (ou plus). Par exemple pour créer une grille d'entiers avec n lignes et p colonnes, on écrit :

```
1 int** g = malloc(n * sizeof(int*));
2
3 for (int i = 0; i < n; i++){
4   g[i] = malloc(p * sizeof(int));
5 }
```

Chaque case de `g` est un tableau d'entiers, i.e. un `int*`, c'est pour ça que l'on écrit `g[i] = malloc(p * sizeof(int));`. Donc, `g` est un tableau dont chaque case est un `int*`, d'où `int** g = malloc(n * sizeof(int*));`.

Autrement dit, `g` pointe vers la première de n cases mémoires `tab[0]` ... `tab[n-1]`, et chaque `tab[i]` est un pointeur vers une liste de p entiers :

`tab[i][0]`, `tab[i][1]`, ..., `tab[i][p-1]`

Visuellement, on représente les tableaux 2D de la manière suivante :

`tab[0][0]`	`tab[0][1]`	...	`tab[0][1]`	...	`tab[0][p-1]`
`tab[1][0]`	`tab[1][1]`	...	`tab[1][j]`	...	`tab[1][p-1]`
⋮	⋮	⋱	⋮		⋮
`tab[i][0]`	`tab[i][1]`	...	`tab[i][j]`	...	`tab[i][p-1]`
⋮	⋮		⋮	⋱	⋮
`tab[n-1][0]`	`tab[n-1][1]`	...	`tab[n-1][j]`	...	`tab[n-1][p-1]`

Dessinons l'état du tas après avoir exécuté le code au dessus avec $n = 5, p = 3$:

Alternativement, si l'on ne veut pas passer par malloc, on peut utiliser la syntaxe suivante pour réserver des tableaux à plusieurs dimensions dans la pile :

```
1 int tab[N_1][N_2]...[N_k];
```

A nouveau, on utilisera rarement cette syntaxe, on privilégiera l'usage de malloc.

On remarque que la création d'une grille demande plusieurs mallocs. Lorsque l'on écrit une fonction créant un objet avec plusieurs mallocs, on écrira systématiquement une fonction servant à la libération mémoire : on crée la "parenthèse fermante" correspondante. Par exemple, pour créer une matrice nulle de n lignes et m colonnes :

```c
/* Renvoie une grille de taille n par m,
   dont toutes les cases sont nulles */
int** matrice_nulle(int n, int m){
  int** g = malloc(n * sizeof(int*)); // n lignes
  for (int i = 0; i < n; i++){
    g[i] = malloc(m * sizeof(int)); // m cases par ligne
    for (int j = 0; j < m; j++){
      g[i][j] = 0;
    }
  }
  return g;
}
```

```c
/* Libère la mémoire allouée à g grille 2D de n lignes  */
void libere_matrice(int** g, int n){
  for (int i = 0; i < n; i++){
    free(g[i]);
  }
  free(g);
}
```

Quelques remarques :

— On a exactement autant de free que de malloc : un free pour la matrice elle même, et n free au sein d'une boucle pour libérer chaque ligne.

— Il faut libérer g après avoir libéré ses cases. En effet, une fois que l'on a libéré g, impossible d'accéder à $g[i]$. A nouveau, l'image des parenthèses est utile : les free se font toujours dans l'ordre **inverse** des malloc !

— Pour la fonction `libere_matrice`, il est nécessaire de connaître n car on doit libérer chaque $g[i]$, mais on a pas besoin de m.

La règle d'or à toujours retenir : **UN MALLOC = UN FREE**.

5 Structures

On veut écrire un programme qui manipule les données de personnes : nom, prénom, age. Supposons que l'on veut écrire une fonction qui affiche les informations d'une personne, et une fonction qui sauvegarde les informations d'une personne dans une base de données. On aurait alors les en-têtes suivants :

```
1 void affiche_personne(char* nom, char* prenom, int age);
2 void sauvegarde_personne(char* nom, char* prenom, int age);
```

De manière générale, toutes les fonctions qui manipulent les informations d'une personne prendront en paramètre tous les attributs.

Dans le reste du programme, à chaque fois que l'on doit afficher les données d'une personne, on devra appeler la fonction en écrivant `affiche_personne(nom, prenom, age)`, et pareil pour `sauvegarde_personne(nom, prenom, age)`, et pour toute autre fonction que l'on a écrit. Si l'on veut modifier le programme pour pouvoir aussi prendre en compte le métier, le numéro de téléphone, l'email, on devra modifier toutes les définitions et tous les appels de fonction au sein du programme :

```
1 void affiche_personne(char* nom, char* prenom, int age, char* metier,
2                       char* numero, char* email);
```

Les structures vont permettre de **regrouper** les différents attributs en une seule variable. Si l'on veut rajouter des attributs, plutôt que de devoir modifier chaque définition et chaque appel de fonction, on pourra modifier la structure elle-même.
Notons que l'on pourrait utiliser un tableau, et considérer une personne comme un tableau de taille N, avec N le nombre d'attributs :

```
1 void affiche_personne(char** personne);
```

avec `personne[0]` qui contient le nom, `personne[1]` le nom, etc... Cependant, on a un attribut entier (l'age), et les tableaux ne peuvent pas stocker plusieurs types, contrairement aux structures !

Exemple 11. On veut représenter en C les informations basiques d'une personne : nom, prénom, age, taille.

On va **définir un nouveau type** qui permet de stocker ces informations :

```
1 struct humain {
2   char* nom;
3   char* prenom;
4   unsigned int age;
5   float taille;
6 };
```

Ce type s'appelle `struct humain`. Il a quatre **attributs**. On peut ensuite l'utiliser comme suit :

```c
int main(){
  struct humain moi = {
    .nom = "Rousseau",
    .prenom = "Guillaume",
    .age = 22,
    .taille=1.91
  };

  printf("Je m'appelle %s %s\n", moi.prenom, moi.nom);
  printf("J'ai %d ans et je mesure %fm\n", moi.age, moi.taille);
  return 0;
}
```

Dans ce code, `moi` est une variable de type `struct humain`, et a donc quatre attributs. Pour accéder à l'attribut `nom`, on écrit `moi.nom`. C'est comme si on avait quatre variables qui s'appellent `moi.nom, moi.prenom, moi.age, moi.taille`.

Attention : on utilise des points-virgules dans la définition du type, mais des virgules (comme pour un tableau) lorsque l'on crée une variable en initialisant les attributs !

A Syntaxe des structs

Le mot-clé ***struct*** permet de créer un nouveau type, ayant plusieurs attributs, selon la syntaxe suivante :

```c
struct nom_structure{
  type_1 attribut_1;
  type_2 attribut_2;
  ...
  type_n attribut_n;
};
```

On peut ensuite déclarer une variable de type `struct nom_structure`, et l'initialiser avec la syntaxe suivante :

```c
struct nom_structure ma_variable = {
  .attribut_1 = valeur_1,
  .attribut_2 = valeur_2,
  ...
  .attribut_n = valeur_n,
}
```

Enfin, pour accéder à un attribut, on utilise la syntaxe :

```c
ma_variable.attribut_i
```

On peut utiliser cette syntaxe pour modifier ou accéder à la valeur d'un attribut. De plus, lorsque l'on définit une variable d'un type struct, on n'est pas obligé de donner une valeur à tous les attributs d'un coup. Par exemple :

```c
struct humain moi = {
  .nom = "Rousseau",
  .prenom = "Guillaume",
};
moi.age = 39;
moi.age = moi.age + 1;
moi.taille = 1.91
```

Quelle est la taille mémoire d'un type struct ?

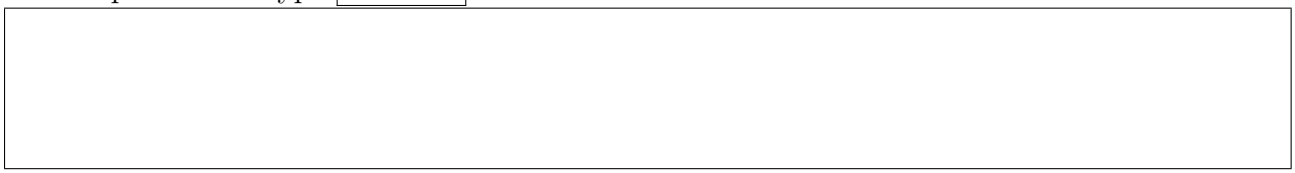
```c
#include <stdio.h>

struct humain {
  char nom[20];
  char prenom[20];
  unsigned int age;
  float taille;
};

int main(){
  printf("%ld\n", sizeof(struct humain));
  return 0;
}
```

Ce programme affiche 48, car nom et prénom prennent 20 octets chacun, et age et taille prennent 4 octets chacun. Donc, dans un type struct, les différents attributs sont stockés en mémoire de manière contigüe, tous les uns à la suite des autres.

Cependant, la taille mémoire d'un type struct n'est pas toujours exactement la somme des tailles de ses attributs. En effet, en C, lorsqu'un type a une taille mémoire de X octets, les variables de ce type doivent avoir des adresses multiples de X. Pour les structures, les adresses doivent être multiples de la taille du plus grand attribut. Par exemple :

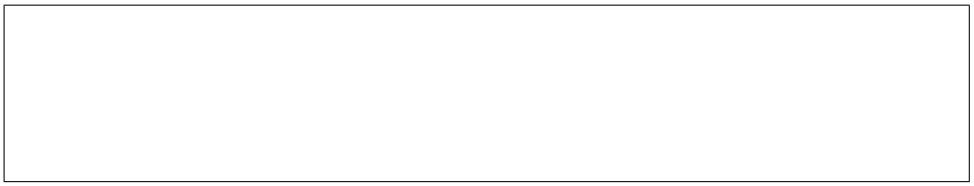
```c
#include <stdio.h>

struct t {
  char c;
  int x;
  char d;
};

int main() {
  printf("%ld\n", sizeof(struct t));
  return 0;
}
```

Si on représente le type `struct t` en mémoire :



Bien que le type `struct t` tient théoriquement sur 6 octets, le programme affiche une taille de 12 octets. Ceci est dû au fait que l'attribut `.x` est un int. D'une part, l'attribut `.x` ne peut pas être stocké juste après l'attribut `.c` qui ne fait qu'un octet, ce qui oblige 3 octets vides à être utilisés entre les deux. D'autre part la structure est forcée de s'aligner sur des adresses multiples de 4, ce qui oblige 3 octets vides à être rajoutés à la fin.
Notons que si l'on déclare les attributs de type char à côté, alors la structure sera stockée de manière plus optimisée, sur 8 octets seulement :

```c
struct t {
  int x;
  char c;
  char d;
};
```



Les types struct sont des types comme les autres. On peut combiner les structs, les pointeurs, les tableaux comme on veut. Par exemple :

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

struct animal{
  char* nom;
  char* espece;
  float poids; // en kilos
  int age;
};

struct humain{
  char* nom;
  int n_animaux; // nombre d'animaux
  struct animal* animaux_compagnie; // tableau d'animaux
};
```

On a donc un type `struct animal` qui représente des animaux, ainsi qu'un type `struct humain` qui représente des humains, et les humains peuvent avoir plusieurs animaux, ce qui est représenté par les deux attributs `n_animaux` et `animaux_compagnie`. Un exemple d'utilisation : on crée une fonction qui permet d'afficher les informations d'un animal, une autre qui permet d'afficher les informations d'un humain, puis on teste tout dans le main :

```c
void print_animal(struct animal a){
  printf("%s est un %s de %d ans qui pèse %f kilos\n",
       a.nom, a.espece, a.age, a.poids);
}

void print_humain(struct humain h){
  printf("%s, a %d animaux de compagnie:\n",
         h.nom, h.age, h.n_animaux);
  // afficher chaque animal
  for (int i = 0; i < h.n_animaux; ++i){
    print_animal(h.animaux_compagnie[i]);
  }
}

int main(){
  struct humain moi = {.nom = "G. Rousseau", .n_animaux = 2};
  moi.animaux_compagnie = malloc(2*sizeof(struct animal))

  moi.animaux_compagnie[0].nom = "Gribouille";
  moi.animaux_compagnie[0].espece = "chat";
  moi.animaux_compagnie[0].poids = 5.21;
  moi.animaux_compagnie[0].age = 1;

  moi.animaux_compagnie[1].nom = "Mao";
  moi.animaux_compagnie[1].espece = "chat";
  moi.animaux_compagnie[1].poids = 12;
  moi.animaux_compagnie[1].age = 19;

  print_humain(moi);

  free(moi.animaux_compagnie);
  return 0;
}
```

B Structure récursive

On veut créer un type struct pour représenter les personnes, et en particulier on souhaite représenter le fait qu'une personne peut avoir un/une meilleur ami/e. Supposons que l'on a définit la structure suivante :

```
1 struct humain{
2   char* nom;
3   unsigned int age;
4   struct humain meilleur_ami;
5 };
```

Autrement dit, on rajoute un attribut du même type que la structure. Le type défini est alors **récursif** car il fait référence à lui-même.
Déterminons la taille T en octets demandée par cette structure. T satisfait $T = 8 + 4 + T$, ce qui n'a clairement pas de solution. En fait, si l'on essaie de compiler ce code, le compilateur produit une erreur.
En revanche, si l'on remplace l'attribut par un **pointeur**, il n'y a plus de problème de taille mémoire, car un pointeur demande 8 octets, peu importe son type.

```
1 struct humain {
2   char* nom;
3   unsigned int age;
4   struct humain* meilleur_ami;
5 };
```

Ce type fait $8 + 4 + 8 = 20$ octets. On peut écrire dans le main :

```
1  int main(){
2    struct humain x = {
3      .nom = "Serge Wifi",
4      .age = 30,
5    };
6
7    struct humain y = {
8      .nom = "Lucie Clavier",
9      .age = 30,
10   };
11   x.meilleur_ami = &y;
12   y.meilleur_ami = NULL; //amitié pas réciproque :(
13
14   return 0;
15 }
```

Une utilité du pointeur NULL, mise en évidence ici, et qu'il peut servir à signaler qu'un attribut est vide.
Remarquons que l'on n'initialise pas toutes les valeurs lorsque l'on déclare x, parce que l'on doit avoir déclaré `y` avant de remplir l'attribut `x.meilleur_ami`.

Pointeurs de structs Écrivons une fonction qui affiche les informations d'une personne et le prénom de son/sa meilleur/e amie :

```
1 /* Affiche les informations de x: nom, prénom, age, taille
2    et prénom du meilleur ami */
3 void print_amities_1(struct humain x){
4   printf("%s: %d ans\n", x.nom, x.age);
5   if (x.meilleur_ami != NULL){
6     printf("Son/sa meilleur/e ami/e est: %s\n\n", (*x.meilleur_ami).nom);
7   }
8 }
```

Comme la fonction prend en entrée un `struct humain`, lorsqu'on l'appelle, toutes les données de l'argument doivent être recopiées dans le paramètre, ce qui est inutilement coûteux. En fait, certains compilateurs **interdisent** de faire des fonctions qui prennent en paramètre ou renvoient des structures directement.
En pratique, on n'écrira (presque) toujours des fonctions qui prennent en paramètre et renvoient des **pointeurs vers des types structs**.
Si l'on dispose d'un pointeur `humain_t* x`, et que l'on veut accéder à son nom, il faut déréférencer le pointeur puis accéder à l'attribut, donc écrire `(*x).nom`. On peut alors réécrire la fonction précédente comme suit :

```
1 void print_amities_2(struct humain* x){
2   assert(x != NULL);
3   printf("%s: %d ans\n", (*x).nom, (*x).age);
4   if ((*x).meilleur_ami != NULL){
5     printf("Son/sa meilleur/e ami/e est: %s\n\n", (*(*x).meilleur_ami).nom);
6   }
7 }
```

On voit que la syntaxe devient très lourde et difficile à lire. En C il y a une syntaxe spéciale pour les pointeurs de structures : `ptr->attr` veut dire `(*ptr).attr`. Autrement dit, `->` est l'équivalent du point, mais pour les pointeurs de structures plutôt que pour les structures directement.
Par exemple la fonction précédente réécrite avec cette syntaxe :

```
1 void print_amities_3(struct humain* x){
2   assert(x != NULL);
3   printf("%s: %d ans\n", x->nom, x->age);
4   if (x->meilleur_ami != NULL){
5     printf("Son/sa meilleur/e ami/e est: %s\n\n", x->meilleur_ami->nom);
6   }
7 }
```

C Alias de Type

Pour alléger le code, on peut renommer les types, en utilisant le mot-clé `typedef` :

```
1  struct humain {
2    char nom[20];
3    char prenom[20];
4    unsigned int age;
5    float taille;
6  };
7
8  typedef struct humain humain_t;
9  // Dans la suite du programme, à chaque fois que l'on
10 // écrit "humain_t", c'est comme si on écrivait "struct humain".
11
12 humain_t x; // exactement comme "struct humain x;"
```

Il y a beaucoup de conventions de nommage de types, en cours on en utilisera une assez standard : donner aux types des noms finissant par `_t`.

On peut aussi combiner définition de struct et aliasing comme suit :

```
1  typedef struct humain {
2    char nom[20];
3    char prenom[20];
4    unsigned int age;
5    float taille;
6  } humain_t;
```

On peut même faire le typedef **avant** la définition du type :

```
1  typedef struct humain humain_t;
2
3  struct humain {
4    char nom[20];
5    char prenom[20];
6    unsigned int age;
7    float taille;
8  };
```

En pratique : il est rare (en MP2I) d'utiliser des variables de type struct directement. On utilisera presque toujours des **pointeurs** de structures, obtenus avec `malloc`, et donc on n'utilisera très rarement la notation `a.x`, où a est une struct, et presque exclusivement la notation `a->x`, où a est un pointeur de struct.

Exercice 3. On propose les types suivants pour représenter une playlist :

```
1  struct chanson {
2    char* nom;
3    char* artiste;
4    int duree; // en secondes
5    struct chanson* suivante;
6  };
7  typedef struct chanson chanson_t;
8
9  typedef struct playlist {
10   char* nom;
11   chanson_t* premiere_chanson;
12 } playlist_t;
```

Chaque struct chanson contient les informations basiques d'une chanson, ainsi qu'un pointeur vers la chanson suivante de la playlist. La structure playlist, elle, contient un nom et un pointeur vers la première chanson. Ce type de structure s'appelle une **liste chaînée**. La dernière chanson c de la playlist vérifiera `c->chanson_suivante == NULL`, marquant la fin de la playlist.

Question 1. Écrire une fonction `int nb_chansons(playlist_t* p)` renvoyant le nombre de chansons de la playlist p.

Question 2. Écrire une fonction `void afficher_playlist(playlist_t* p)` qui affiche les informations d'une playlist, sous le format suivant :

```
Playlist Ma_Super_Playlist:
1) Shutdown - BLACKPINK (3m01s)
2) 010101 (Binary  System) - Hiromi (8m43s)
....
857) Smooth Operators - Sade (4m18s)
Longueur totale: 68h16m34s
```

Question 3. Écrire une fonction `void ajouter_chanson(playlist_t* p, chanson_t* c)` qui ajoute la chanson c **au début** de la playlist p.

Question 4. (Plus dur) Écrire une fonction `void supprimer_titre(playlist_t* p, char* titre)` qui supprime dans la playlist p la première occurrence d'une chanson ayant le titre spécifié.

6 Fichiers

En C, il est possible d'interagir avec des fichiers : lire, écrire, modifier... La librairie `<stdio.h>`, que nous avons déjà utilisé pour les fonctions `printf` et `scanf`, contient également des fonctions permettant d'interagir avec des fichiers.

A Ouverture et fermeture de fichier

La fonction `fopen` sert à ouvrir un fichier. Sa signature est :

```
FILE* fopen(char* file_name, char* mode)
```

Elle prend en argument un nom de fichier `file_name` ainsi qu'un mode d'ouverture `mode`. La valeur de retour est un pointeur vers une valeur de type `FILE`, qui est le type utilisé pour les fichiers. On ne s'intéresse pas à la structure interne du type `FILE`, on n'interagira avec que via les fonctions de la librairie standard.

Lorsque l'on ouvre un fichier dans un programme C, on positionne dans le fichier une tête de lecture, qui détermine où le programme va lire et écrire.

Le mode d'ouverture définit si l'on ouvre le fichier pour y écrire ou y lire, et où l'on place la tête de lecture. Il existe plusieurs modes, dont les principaux sont :

- `"r"` ouvre le fichier en lecture uniquement, avec la tête de lecture au début du fichier.
- `"w"` ouvre le fichier en mode écriture, et crée le fichier s'il n'existe pas déjà. Ce mode positionne la tête de lecture au début de fichier, et **vide le fichier** s'il existe déjà.
- `"a"` ouvre le fichier en mode écriture, créant le fichier s'il n'existe pas, et positionne la tête de lecture à la fin du fichier. Ce mode permet donc d'écrire en rajoutant des caractères à un fichier. Le “a” signifie ”append”.

Si l'ouverture n'est pas possible, par exemple si l'on ouvre en lecture seule un fichier qui n'existe pas, la valeur de retour est NULL, et les lectures et écritures suivantes vont donc causer des erreurs de segmentation. Ainsi, lorsqu'on ouvre un fichier, il est important de vérifier que l'ouverture a marché, sans quoi on affiche une erreur et on arrête le programme. On peut par exemple utiliser assert pour cela.

Inversement, la fonction `fclose` sert à fermer un fichier qui a été ouvert. Sa signature est :

```
int fclose(FILE* file)
```

Elle renvoie un code indiquant une éventuelle erreur : 0 si tout s'est bien passé, autre chose sinon. Il faut **toujours fermer un fichier qui a été ouvert**. En effet, un fichier ouvert sans être fermé est un type de **fuite mémoire**. La manipulation d'un fichier se fait donc ainsi :

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

int main(){
  FILE* file = fopen("mon_fichier.txt", "w"); // ou "r" ou "a"
  assert(file != NULL);

  /* LECTURE OU ECRITURE DANS LE FICHIER */

  fclose(file);
  return 0;
}
```

Comme pour malloc et free, il est de bonne pratique d'écrire l'appel à fclose dès que l'on écrit l'appel à fopen, comme si c'étaient des parenthèses.

B Lecture, écriture

La lecture et l'écriture dans un fichier se font presque exactement comme la lecture et l'écriture dans le terminal. On utilise les fonctions `fscanf` et `fprintf`, qui marchent exactement comme `scanf` et `printf`, en précisant en premier argument un fichier, de type `FILE*` :

```
1 FILE* file = fopen("mon_fichier.txt", "w");
2 fprintf(file, "Bonjour les MP%dI\n", 2);
3 fclose(file);
```

Exécuter ces trois lignes dans un programme C aura pour effet de créer un fichier appelé "mon_fichier.txt" et d'y écrire "Bonjour les MP2I".

```
1 FILE* file = fopen("date.txt", "w");
2 int j, m, a;
3 fscanf(file, "%d %d %d", &j, &m, &a);
4 fclose(file);
```

Ce code aura pour effet d'ouvrir un fichier appelé "date.txt" et d'essayer d'y lire trois entiers. Comme pour scanf, si la fonction lit autre chose que des entiers, il peut se passer n'importe quoi.

C Fichiers spéciaux

Dans la bibliothèque `<stdio.h>`, trois fichiers spéciaux sont déjà définis. Ces fichiers sont ouverts au début de chaque programme, et il ne faut pas les fermer soi-même (vous pouvez essayer...).

Entrée standard `FILE* stdin` est le flux d'entrée standard. Ce fichier correspond à ce que l'on écrit dans le terminal. Ainsi, lire depuis le fichier stdin revient exactement à lire depuis le terminal. Autrement dit, les deux instructions suivantes sont équivalents :

```c
scanf("%d", &n);
```

```c
fscanf(stdin, "%d", &n);
```

Sortie standard `FILE* stdout` est le flux de sortie standard. Ce fichier correspond à ce qui apparaît dans le terminal. Ainsi, écrire dans le fichier stdout revient exactement à écrire dans le terminal. Autrement dit, les deux instructions suivantes sont équivalents :

```c
printf("%d\n", n);
```

```c
fprintf(stdout, "%d\n", n);
```

Flux standard d'erreur `FILE* stderr` est le flux d'erreur standard. C'est dans ce fichier que le programme écrit lorsqu'il rencontre une erreur. De notre point de vue, le flux d'erreur standard s'affiche dans le terminal comme la sortie standard.

D Redirection de flux

Dans un terminal, il est possible de lancer un programme en ***redirigeant*** ses flux standards d'entrée, de sortie, et d'erreur :

— Pour rediriger la sortie standard :

```
./monprogramme > out.txt
```

En lançant cette commande, le programme `./monprogramme` va écrire dans `out.txt` tout ce qu'il aurait écrit dans le terminal normalement (sauf les messages d'erreur). Cette redirection de flux se fait en mode "w", autrement dit en écrasant le contenu précédent du fichier. Tout se passe comme si, au tout début du main, on avait écrit `stdout = fopen("out.txt", "w");`. Si l'on veut faire la redirection en mode "a", c'est à dire en rajoutant les données à la fin du fichier, on utilise :

```
./monprogramme >> out.txt
```

— Pour rediriger l'entrée standard :

```
./monprogramme < in.txt
```

En lançant cette commande, le programme `./monprogramme` va s'exécuter, en lisant dans in.txt au lieu de lire dans le terminal, comme si, au début du main, on avait écrit `stdin = fopen("in.txt", "r");`.

— Pour rediriger le flux d'erreur :

```
./monprogramme 2> err.txt
```

Cette commande va exécuter le programme monprogramme, en écrivant les erreurs dans err.txt. Les erreurs telles que les stack overflow, les segmentation fault ne sont pas écrites dans le flux d'erreur standard car elles ne sont pas écrites par le programme lui-même. En revanche les erreurs d'assertions, et les autres erreurs soulevées par des instructions spécifiquement dédiée aux erreurs, y sont écrites (en C, la fonction `perror` sert à afficher un message d'erreur).

On peut combiner toutes les syntaxes pour rediriger les trois flux standards en même temps :

```
./monprogramme < in.txt > out.txt 2> err.txt
```

Par exemple, on considère le programme suivant :

```c
#include <stdio.h>

int main(){
  int x;
  while (scanf("%d", &x) != EOF){
    printf("%d\n", x*x);
  }
  return 0;
}
```

Ce programme lit un entier dans l'entrée standard, et écrit son carré dans la sortie standard. Il s'arrête lorsque scanf renvoie la valeur `EOF` (**E**nd **O**f **F**ile). En C, cette valeur indique que l'on a atteint la fin d'un fichier. On peut simuler une fin de fichier dans un terminal en appuyant sur CTRL+D.

On peut donc exécuter ce programme directement :

```
fredfrigo@ubuntu:~$ gcc square.c -o square -Wall -Wextra
fredfrigo@ubuntu:~$ ./square
6   (rentré)
36  (affiché par le programme)
-3  (rentré)
9   (affiché par le programme)
12  (rentré)
144 (affiché par le programme)
(On appuie sur CTRL+D pour arrêter l'exécution)
```

Ou bien l'exécuter en redirigeant sa sortie et son entrée :

```
./square < in.txt > out.txt
```

Ce qui aura pour effet de créer un fichier "`out.txt`" contenant le carré de tous les entiers stockés dans "`in.txt`".

Enfin, il est possible de rediriger la sortie d'un programme vers l'entrée d'un autre programme. Ce mécanisme utilise la syntaxe suivante :

```
./prog1 | ./prog2
```

Le symbole "|" s'appelle le tuyau, ou **pipe** en anglais, car il se comporte comme si on avait installé un tuyau entre la sortie de prog1 et l'entrée de prog2.

Cette commande a donc pour effet de lancer prog1 et prog2 en même temps, avec prog2 qui va lire la sortie de prog1 comme si c'était le terminal.