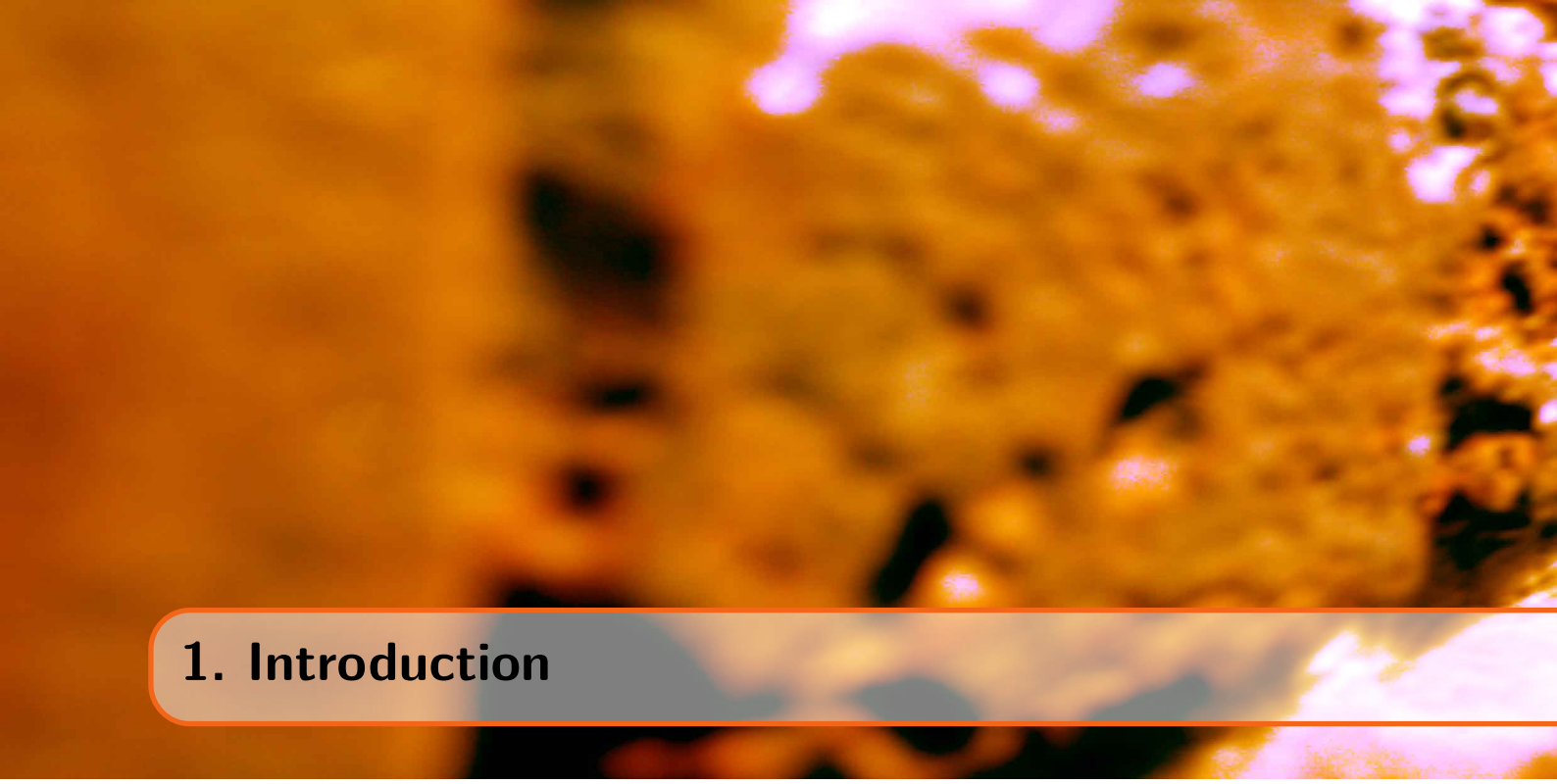
Informatique

CPGE

Marc de Falco

Table des matières

1 Introduction



1. Introduction

Ce site présente des documents personnels autour de l'informatique.

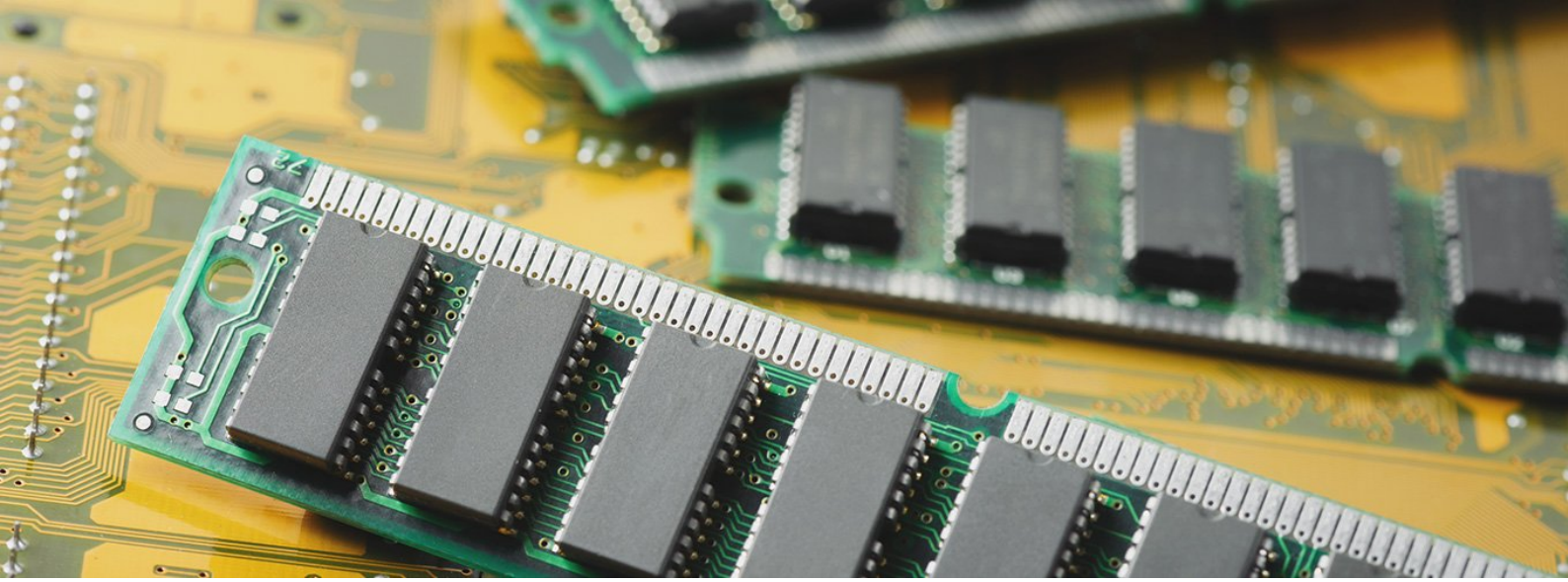
Il s'agit pour le moment d'un premier jet, et il est amené à beaucoup évoluer. Il correspond à mon interprétation personnelle de certaines notions.

Pour le moment, ce sont des développements indépendants. Dans un second temps, ils seront éventuellement articulés autour d'une progression.

Des problèmes sont présentés au cours des documents, ils peuvent donner lieu à des développements en classe, des séances de travaux pratiques ou des problèmes. Plutôt que de donner un découpage figé en questions, ils sont présentés tels quels.

Mon idée principale avec ces ressources est de proposer un contenu assez riche, plutôt à destination des enseignants. Certains points un peu pointus ne sont pas forcément essentiels pour tous les étudiants.

II Systèmes



2. Gestion de la mémoire dans un programme compilé

■ **Remarque 2.1** Ce chapitre se concentre sur la manière dont un programme compilé gère la mémoire. Il est question, en particulier, de la notion de variable. Le modèle dans lequel on se place est celui du langage C où une variable est un emplacement mémoire.

■

I Organisation de la mémoire

■ **Note 2.1** Ici, je fais le choix d'une présentation assez informelle pour ne pas qu'elle soit trop liée à la réalité d'un compilateur en particulier.

■

Un programme compilé gère la mémoire d'un ordinateur de deux manières très différentes

— statiquement : c'est le cas des variables locales ou globales définies dans le programme. Au moment de la compilation, le compilateur dispose de l'information suffisante pour prévoir de la place en mémoire pour stocker ces données.
— dynamiquement : c'est le cas des objets dont la taille n'est connue qu'à l'exécution et peut varier selon l'état du programme. C'est alors au moment de l'exécution que le programme va faire une demande d'allocation pour obtenir une place mémoire.

En terme d'allocation statique, on peut distinguer plusieurs types de mémoire :

- les variables globales initialisées qui seront stockées dans le binaire et placées en mémoire dans une zone spécifique chargée avec le binaire
- les variables globales non initialisées dont seule la déclaration sera dans le binaire et qui seront allouées, placées en mémoire et initialisées à 0 au moment du chargement du binaire
- les variables locales et les paramètres qui sont placés dans une pile afin de les allouer uniquement au moment de l'exécution du bloc ou de la fonction

L'allocation dynamique utilise une zone mémoire appelée tas dont une possible organisation est développée dans la partie sec. V.4.

Structure de la mémoire associée à un programme

Considérons le programme c suivant.

```c
const int a  = 42;
int b[] = { 1, 2, 3 };
int c;

int f(int x, int y)
{
    int z = x;
    z = z * y;
    return z;
}

int main(int argc, char **argv)
{
    const int d = 1664;

    c = f(a, d);

    return 0;
}
```

Il est possible d'observer la manière dont sa mémoire sera répartie en utilisant la commande `objdump` :

```
$ gcc -c memoire.c
$ objdump -x memoire.o

memoire.o:     file format elf64-x86-64
memoire.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000053  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000093  2**0
                  CONTENTS, ALLOC, LOAD, DATA
```

```
  2 .bss          00000004  0000000000000000  0000000000000000  00000094  2**2
                  ALLOC
  3 .rodata       00000014  0000000000000000  0000000000000000  00000098  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000013  0000000000000000  0000000000000000  000000ac  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000bf  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000c0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 orga.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata    0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack    0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     O .rodata    0000000000000004 a
0000000000000000 g     O .data  000000000000000c b
0000000000000000 g     O .bss   0000000000000004 c
0000000000000000 g     F .text  000000000000001f f
000000000000001f g     F .text  0000000000000034 main


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000042 R_X86_64_PLT32    f-0x0000000000000004
0000000000000048 R_X86_64_PC32     c-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x000000000000001f
```

On retrouve dans les sections mémoire :

- `.text` contenant le programme binaire
- `.data` contenant les variables globales initialisées et non constantes
- `.bss` contenant les variables non initialisées
- `.rodata` contenant les variables globales initialisées mais constantes.

Ici, le schéma mémoire est un peu plus compliqué car une zone mémoire distincte est prévue pour les variables constantes initialisées pour des raisons de sécurité.

La pile et le tas sont automatiques et n'ont pas besoin de figurer dans le binaire, c'est pour cela qu'on ne les trouve pas dans la liste.

Dans cette description mémoire, on trouve la table des symboles qui décrit où vont se trouver en mémoire certaines variables.

Nom de variable	Sorte de déclaration	Section mémoire
a	constante globale initialisée	rodata
b	globale initialisée	data
c	globale non init.	bss

La section suivante permettra de compléter le tableau en étudiant comment les paramètres et variables locales sont gérés. On remarque cependant qu'il n y a pas de symboles pour ceux-ci. En effet, le nom des variables locales est a priori perdu après la compilation contrairement aux variables globales.

II Portée d'un identificateur

En ce qui concerne une variable dans un programme, on peut définir deux notions d'apparence assez similaire.

D'une part la portée d'un identificateur qui correspond au texte du programme :

Définition II.1 La **portée** d'un identificateur est définie par la zone du texte d'un programme dans laquelle il est possible d'y faire référence sans erreurs.

■ **Remarque 2.2** Dans le langage C, un identificateur peut être utilisé dès sa déclaration mais tant que la variable n'est pas initialisée, le comportement n'est pas spécifié et il faut considérer cela comme une erreur. Le compilateur produit ainsi un avertissement quand on utilise le paramètre `-Wall`.

Dans le cas d'une définition, il est ainsi possible de faire référence à l'identificateur dans l'expression de son initialisation : `int x = x`. Ce cas est pathologique et le fait qu'on compte la ligne de déclaration dans la portée ne devrait pas inciter à écrire ce genre de code qui produira, de toutes façons, une erreur avec les options `-Wall -Werror`.

■

Dans le programme :

```
1  int a = 1;
2
3  int f (int x)
4  {
5      int y = x + a;
6      return y;
7  }
8
9  int g()
10 {
11     int z = 3;
12     return z + f(z);
13 }
```

La portée des identificateurs est :

Identificateur	Portée
a	1-13
x	3-7
y	5-7
f	4-13

Identificateur	Portée
g	10-13
z	11-13

Pour une fonction, afin de pouvoir écrire des fonctions récursives, l'identificateur est utilisable dans le corps de la fonction.

Comme la portée est une notion associée aux identificateurs, elle est indépendante de la notion de variables. Si on considère le programme suivant :

```
1  void f()
2  {
3      int i = 3;
4
5      return i;
6  }
7
8  void g()
9  {
10     int i = 5;
11
12     return i+1;
13 }
```

L'identificateur `i` a pour portée les lignes 3-6 et 10-13.

Un autre phénomène plus complexe peut se produire quand on redéfinit un identificateur dans sa portée.

Considérons le programme suivant

```
1  void f()
2  {
3      int i = 3;
4
5      for (int j = 0; j < 3; j++)
6      {
7          int i = 4;
8
9          i += j;
10     }
11
12     return i;
13 }
```

Ici, l'identificateur `i` a pour portée les lignes 3-13 mais dans les lignes 7-10 il y a un phénomène dit de masquage où la première définition est cachée par la seconde.

L'identificateur associé à une variable globale a pour portée l'ensemble des lignes suivant sa déclaration.

■ **Remarque 2.3** En C, la portée d'un identificateur est **statique** : elle dépend uniquement du texte du programme au moment de la compilation.

En Python, la portée d'un identificateur est **dynamique** : elle peut dépendre de l'exécution d'un programme. Par exemple si on considère le programme Python

```
1  if condition:
2      x = 3
```

La portée de l'identificateur `x` dépend ici du fait que la `condition` soit réalisée ou non.

■

III Durée de vie d'une variable

Définition III.1 La **durée de vie** d'une variable correspond à la période de son exécution durant laquelle la variable est présente en mémoire.

Sauf indication contraire, la durée de vie d'une variable locale en C est définie par la portée de l'identificateur qui lui est associé : la place est réservée au début de sa portée et libérée à la fin.

La dure de vie d'une variable globale est l'intégralité du programme. En effet, comme on a pu le voir dans la section sec. I.

Il est possible de définir en C des variables locales dont la durée de vie dépasse sa portée. On parle alors de variables *statiques*. Ce point étant assez technique, il sera ignoré ici.

IV Piles d'exécution, variables locales et paramètres

On a vu qu'en raison de leur durée de vie, les variables globales étaient allouées dès le chargement du programme. Pour les variables locales ainsi que la mécanique des appels, on utilise une **pile**.

Cette pile d'exécution est représentée en mémoire par un tableau et un indicateur de fond de pile.

Le remplissage de ce tableau s'effectue souvent des adresses hautes vers les adresses faibles : on empile en faisant diminuer les adresses.

■ **Remarque 2.4** En fait, il existe des architectures où les adresses sont croissantes. Ce qui importe est que le tas et la pile aient des comportements opposées pour qu'ils puissent grandir dans la même zone mémoire.

■

Un compilateur peut faire le choix d'utiliser directement des registres processeurs pour les variables locales ou pour passer des paramètres à une fonction. Ici, pour simplifier, on va supposer que ce n'est pas le cas et que tout passe par la pile d'exécution.

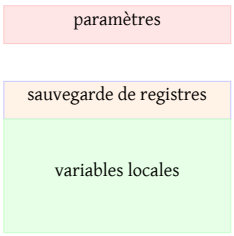
■ **Remarque 2.5** Afin de pouvoir appeler une fonction dans une bibliothèque potentiellement compilée avec un autre compilateur, il est nécessaire d'avoir une convention d'appels de fonctions. Une telle convention est appelée une *interface applicative binaire* (Application Binary Interface).

La convention `System V AMD64 ABI` qui est celle de Linux et macOS sur des architectures 64bits consiste à utiliser des registres entiers pour les six premiers arguments entiers ou pointeurs et des registres flottants pour les huit premiers arguments flottants. Les arguments suivants sont alors passés sur la pile (donc dès le septième entier/pointeur ou neuvième flottant.

La convention `cdecl` qui est assez répandue sur les architectures 32bits consiste à utiliser la pile. Par contre la valeur de retour est présente dans des registres comme pour la convention `System V AMD64 ABI`.

■

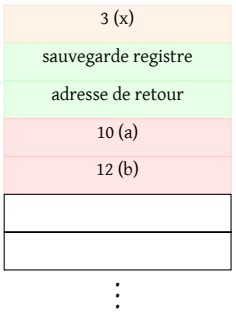
Lors d'un appel une fonction passant par la pile, on commence par empiler les paramètres (souvent de la droite vers la gaucher) puis on empile l'adresse à laquelle doit revenir l'exécution une fois que la fonction aura terminé son exécution.
Au début de l'exécution de cette fonction, on place sur la pile l'adresse du fond de pile et on déplace celui-ci pour réserver de la place pour les variables locales.
L'empreinte sur la pile d'un appel de fonction est appelée une structure pile (**stack frame** en anglais). La pile est alors organisée, depuis l'appel au point d'entrée du programme, par empilement et dépilement de structures piles.



Structure pile associée à un appel de fonction

Voici un exemple possible de l'état de la pile d'exécution lors de l'exécution d'un programme compilé avec la norme `cdecl` (Il suffit d'ajouter l'argument `-m32` pour compiler en 32bits).

```
1
2  int f(int a, int b)
3  {
4      int c = 3;
5      /* pile ici après l'appel en l13 */
6      c = c + b;
7      c = c * a;
8      return c;
9  }
10
11 int main()
12 {
13     int x = f(10,12);
14 }
```



A chaque appel de fonction, on va donc empiler une structure de pile complète, puis la dépiler à la sortie. Ce mécanisme est essentiel pour permettre la récurrence car il permet d'effectuer plusieurs appels d'une même fonction sans risquer que la mémoire utilisée lors d'un des appels interfère avec un autre. On comprend également les limites de la récursivité ici car cet empilement successif de structures de piles peut dépasser la taille maximale de la pile d'exécution : on parle alors de *dépassement de pile* ou **stack overflow** en anglais.

■ **Remarque 2.6** Il est possible de définir des variables locales qui soient situées au même emplacement mémoire pour tous les appels d'une fonction, c'est ce qu'on appelé des variables *statiques* dans le paragraphe précédent.

Ce mécanisme est essentiellement géré comme les variables globales et il ne sera pas développé dans la suite.

■

■ **Note 2.2** Je me demande s'il faudrait parler plus précisement de la manière dont la pile est gérée avec les registres ebp/esp. Mais ça ne me semble pas apporter grand chose ici.

■

V Allocation dynamique

Comme cela a été vu dans la section sec. I, il est possible d'allouer dynamiquement de la mémoire. Pour gérer cette allocation dynamique, on passe par une zone mémoire appelé le *tas* ainsi que par un mécanisme d'allocation et de libération de mémoire au niveau du système.

Pour l'utilisateur, cette gestion interne est transparente et on peut se contenter de considérer qu'il y a deux mécanismes :

— l'**allocation** mémoire où on demande à ce qu'une zone mémoire d'une certaine taille soit allouée
— la **libération** mémoire où on signale que la zone mémoire peut être récupérée par le système.

Naturellement, la mémoire d'un ordinateur étant finie, il est très important de libérer au plus tôt la mémoire non utilisée pour éviter d'épuiser la mémoire. Quand un programme ne libère pas toute la mémoire qu'il alloue, on parle de **fuite mémoire**. L'empreinte mémoire d'un tel programme peut alors croître jusqu'à rendre le programme ou le système inutilisable.

V.1 Allocation

Pour allouer une zone mémoire, on utilise la fonction `malloc` dans `stdlib.h` de signature :

```
void *malloc(size_t size)
```

Ici `size` indique le nombre d'**octets** à allouer et la fonction renvoie un pointeur vers le premier octet alloué. Comme la fonction ne connait pas le type d'objets alloués, on utilise ainsi le type `void *`.

Ce type joue un rôle spécial et on peut changer directement le type de la valeur de retour sans rien avoir à écrire d'autre l'appel à malloc :

```
char *t = malloc(n);
```

■ **Remarque 2.7** Dans le langage C++ qui peut être vu comme un successeur de C, il est obligatoire de préciser ici le nouveau type à l'aide d'un un **transtypage**. Pour convertir la valeur `x` vers le type `char *` on écrit alors `(char *)x`. Ainsi, pour allouer un tableau de `n` caractères, on utilisera :

```
char *t = (char *) malloc(n);
```

Bien que ce ne soit pas nécessaire en C, il est fréquent de rencontrer des programmes présentant de tels transtypages qui sont superflus mais corrects syntaxiquement en C.

■

Pour obtenir la taille à allouer, il peut être utile d'utiliser l'opérateur `sizeof` qui prend en entrée un type ou une valeur et renvoie sa taille. Ainsi si on peut allouer un tableau de n entiers non signés ainsi :

```
unsigned int *t = malloc( sizeof(unsigned int) * n );
```

et cet appel ne dépend de la taille prise par un `unsigned int` sur l'architecture.

Une autre raison de l'utilisation de `sizeof` est l'extensibilité. Par exemple, si on a un `struct point` représentant des points en 2D :

```
struct point {
    float x;
    float y;
};
```

on peut allouer un tableau de n points ainsi :

```
struct point *t = malloc( sizeof(struct point) * n );
```

Si jamais on change la structure pour représenter des points en 3D ainsi :

```
struct point {
    float x;
    float y;
    float z;
};
```

il sera inutile de changer le code d'allocation du tableau car `sizeof(point)` tiendra compte automatiquement du changement.

Si jamais une erreur empêche d'allouer la mémoire - ce qui peut être le cas s'il n'y a plus de mémoire disponible - le pointeur renvoyé par `malloc` a la valeur spéciale `NULL`.

■ **Remarque 2.8** La zone mémoire renvoyée par `malloc` n'est pas initialisée. On ne peut pas supposer qu'elle soit remplie de la valeur 0. Il faut donc manuellement initialiser la mémoire après le retour de `malloc`.

■

V.2 Libération

Pour libérer la mémoire, on utilise la fonction `free` également présente dans `stdlib.h` et dont la signature est :

```
void free(void *ptr);
```

■ **Remarque 2.9** Il est très important d'utiliser uniquement un pointeur obtenu précédemment par un appel à `malloc` et de ne pas l'utiliser plus d'une fois.

Le programme suivant provoque une erreur `free(): invalid pointer` à l'exécution mais est détécté par un avertissement du compilateur : `warning: attempt to free a non-heap object 'a'`.

```
#include <stdlib.h>

int main()
{
    int a;
    free(&a);
    return 0;
}
```

Le programme suivant alloue un tableau de deux `char` et appelle `free` sur l'adresse de la seconde case. En faisant cela, il n'y a pas d'avertissement car on appelle `free` sur un objet qui est effectivement sur le tas. On obtient alors à nouveau une erreur à l'exécution `free(): invalid pointer`.

```
#include <stdlib.h>

int main()
{
    char *a = malloc(2);
    free(&(a[1]));
    return 0;
}
```

Le programme suivant libère deux fois la mémoire et provoque l'erreur `free(): double free detected in tcache 2` à l'exécution.

```c
#include <stdlib.h>

int main()
{
    char *a = malloc(2);
    free(a);
    free(a);
    return 0;
}
```

■

V.3 Protection mémoire

Comme on l'a vu dans la première partie, quand un programme s'exécute il a un environnement mémoire constitué de plusieurs zones, parfois appelées **segments**, avec le droit d'écriture dans certaines d'entre elles.

Le système d'exploitation protège ainsi la mémoire et, au niveau matériel, l'unité de gestion de la mémoire connait les adresses accessibles à un programme. En cas d'accès anormal, le matériel provoque une erreur qui remonte au système d'exploitation qui termine l'exécution du programme avec une erreur souvent intitulée `Segmentation fault`.

Voici quelques exemples commentés produisant des erreurs de type `segmentation fault` à l'exécution.

Lecture à l'adresse 0, ce qui provoque toujours une erreur.
Même problème avec une adresse inaccessible ou invalide.

```c
int main()
{
    int *a = 0;

    return a[0];
}
```

Écriture dans une zone en lecture seule comme le segment du code.

```c
int main()
{
    int *a = (int*) (&main);
    // a pointe sur le corps de la fonction ma

    *a = 0;

    return 0;
}
```

V.4 Réalisation d'un système d'allocation de mémoire

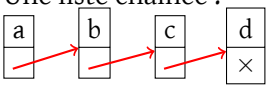
■ **Note 2.3** Prérequis : listes chaînées

■

Afin de comprendre comment fonctionne le tas, et en particulier malloc et free, on va simuler ici ce comportement en allouant une grande plage de mémoire avec malloc et en gérant le découpage et l'allocation de celle-ci.

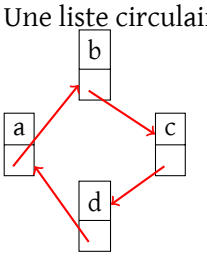
Pour gérer les blocs mémoires libres, on utilise une liste circulaire. Une liste circulaire est une liste chaînée avec un lien supplémentaire entre le premier et le dernier maillon, ce qui fait qu'on peut considérer n'importe quel maillon comme étant la *tête* de la liste.

Une liste chaînée :



Ici le dernier maillon comprend un pointeur qui ne pointe sur rien indiqué par ×, en pratique il a la valeur `NULL`

Une liste circulaire :



Le seul changement est donc de faire pointer le dernier maillon sur le premier. Le fait d'avoir preserver les valeurs dans les maillons permet ici de voir ce qu'est devenu le premier maillon, mais on peut accéder à cette liste par n'importe lequel de ces maillons.

Les noeuds de la liste circulaire de blocs libres auront pour valeur un triplet `(adresse,taille,libre)` qui indique qu'à l'adresse `adresse` il y a un bloc de `taille` octets et le booléen `libre` indique si ce bloc a été alloué ou non.

Pour cela on commence par définir une structure `bloc` et une fonctiond de création d'un bloc :

```
struct bloc {
    void *adresse;
    uint32_t taille;
    bool libre;
    struct bloc *suivant;
};

struct bloc *cree_bloc(void *adresse, uint32_t taille, bool libre)
{
    struct bloc *b = malloc(sizeof(struct bloc));
    b->adresse = adresse;
    b->taille = taille;
    b->libre = libre;
    return b;
}
```

On définit ensuite deux variables globales :

— `bloc_libres` qui va pointer sur un maillon de la liste circulaire des blocs

— `plage_memoire` qui pointe sur l'adresse de la plage mémoire que l'on va gérer et servira à la libérer en sortie de programme.

```
struct bloc *blocs_libres;
void *plage_memoire;
```

La fonction `creation_blocs_libres` permet de créer la liste circulaire avec un premier bloc qui pointe sur lui-même et qui correspond à l'adresse que l'on va placer dans `plage_memoire`.

```
void creation_blocs_libres(uint32_t taille_bloc_initial)
{
    plage_memoire = malloc(taille_bloc_initial);
    blocs_libres = cree_bloc(plage_memoire, taille_bloc_initial, true);
    blocs_libres->suivant = blocs_libres; // boucle initiale
}
```

Pour libérer la liste à la sortie du programme, on définit la fonction `destruction_blocs_libres` qui présente ainsi le parcours usuel d'une liste circulaire : on procède comme pour une liste chaînée classique mais, au lieu d'utiliser un test `bloc_courant->suivant == NULL` pour l'arrêt, il faut se souvenir du premier bloc et tester pour voir si on est revenu au point de départ. On n'oublie pas de libérer l'espace `plage_memoire` à la fin.

```
void destruction_blocs_libres()
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    // on boucle pour libérer chaque maillon
    while (true) {
        struct bloc *bloc_suivant = bloc_courant->suivant;
        free(bloc_courant);
        if (bloc_suivant == premier_bloc) return;
        bloc_courant = bloc_suivant;
    }

    // on libère la plage mémoire initiale
    free(plage_memoire);
}
```

Pour allouer t octets, on parcourt la liste des blocs jusqu'à trouver un bloc b libre de taille $b.t$ telle que $b.t \geq t$. Si un tel bloc n'existe pas, on renvoie le pointeur `NULL` signe d'un échec d'allocation. Sinon, on indique que le bloc est occupé, on va renvoyer l'adresse du bloc obtenu mais, si $b.t > t$ on insère après b un nouveau bloc libre de taille $b.t - t$. Dans tous les cas, on fait pointer la liste des blocs libres vers le bloc qui suit b, qui est peut-être le bloc nouvellement créé et a de grandes chances d'être libre.

Ce mécanisme est implementé dans la fonction `allocation` :

```
void *allocation(uint32_t taille)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (!bloc_courant->libre || bloc_courant->taille < taille)
    {
        bloc_courant = bloc_courant->suivant;
        if (bloc_courant == premier_bloc)
        {
            // Retour au point de départ : échec d'allocation
            return NULL;
        }
    }

    // bloc_courant pointe sur un bloc libre de bonne taille

    void *adresse = bloc_courant->adresse;
    bloc_courant->libre = false;
    if (bloc_courant->taille > taille) {
        // on le sépare en deux pour récupérer la place
        struct bloc *bloc_libre = cree_bloc(adresse+taille,
                bloc_courant->taille-taille, true);
        bloc_libre->suivant = bloc_courant->suivant;
```

```
        bloc_courant->suivant = bloc_libre;
    }

    // On pointe sur le bloc suivant qui est sûrement libre
    blocs_libres = bloc_courant->suivant;

    return adresse;
}
```

Pour libérer un bloc, on parcourt la liste jusqu'à trouver le bloc correspondant à l'adresse à libérer et on indique que le bloc est libre. Ici, il y a deux `assert` permettant de s'assurer que l'adresse est bien celle d'un bloc et que le bloc n'a pas déjà été libéré.

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }

    // pas de double libération
    assert(!bloc_courant->libre);

    // on libère l'adresse
    bloc_courant->libre = true;
}
```

Voici un premier programme de test de ces fonctions qui alloue 10 octets puis effectue plusieurs allocation. L'allocation de `c` échoue car il n'y a plus de place libre.

```
int main()
{
    creation_blocs_libres(10);

    uint8_t *a = allocation(5)
    uint8_t *b = allocation(3);
    uint8_t *c = allocation(3);
    liberation(b);
    uint8_t *d = allocation(4);

    printf("Allocation a:0x%lx b:0x%lx c:0x%lx d:0x%lx\n",
            (uint64_t)a, (uint64_t)b, (uint64_t)c, (uint64_t)d);

    destruction_blocs_libres();

    return 0;
}
```

Ce programme affiche alors

```
Allocation a:0x55d92b84f2a0 b:0x55d92b84f2f0 c:0x0 d:0x55d92b84f2f0
```

■ **Note 2.4** Ajouter des diagrammes d'évolution de la liste circulaire.

■

Cette méthode d'allocation a un défaut majeur : elle fragmente l'espace libre. Dans le programme suivant, il sera impossible d'allouer `b` car la liste circulaire contient deux blocs libres de 5 octets et non une bloc libre de 10 octets.

```
creation_blocs_libres(10);

uint8_t *a = allocation(5)
liberation(a);
uint8_t *b = allocation(10);
```

■ **Note 2.5** Ajouter un diagramme.

■

On peut éviter cela en effectuant une phase de coalescence des blocs libres à la libération. En vertu de la nature de la liste circulaire, il est nécessaire de fusionner un bloc libre avec les blocs suivants. En utilisant une liste circulaire doublement chaînée, on pourrait également fusionner avec les blocs précédents.

Pour cela on change la fonction `liberation` ainsi :

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }

    // pas de double libération
    assert(!bloc_courant->libre);

    // on libère l'adresse
    bloc_courant->libre = true;

    premier_bloc = bloc_courant;
    bloc_courant = bloc_courant->suivant;

    while (bloc_courant != premier_bloc && bloc_courant->libre)
    {
        struct bloc* actuel = bloc_courant;
        premier_bloc->taille += bloc_courant->taille;
        bloc_courant = bloc_courant->suivant;
        free(actuel); // on libere le bloc inutile
    }

    premier_bloc->suivant = bloc_courant;
}
```