

## Programmation avancée et Structures de données

Ingénieurs Sup Galilée – Ingénieur Informatique 1<sup>o</sup> année – Paris 13

2021-2022

- Cahier élève -

Partie 1 – Rappels sur les variables

Partie 2 – Les pointeurs

Partie 3 – Listes chaînées

Partie 4 – Files, Piles, chaînage double

Partie 5 – Arraylist, Table de hachage

**Enseignant**

John Chaussard

Université Paris 13, LAGA, Bureau D402

[chaussard@math.univ-paris13.fr](mailto:chaussard@math.univ-paris13.fr)



# Programmation avancée et structures de données

*Partie 1 – Rappels sur les variables*

Ecole Sup Galilée - Spécialité Informatique - 1<sup>ère</sup> année

2020-2021

**John Chaussard**

LAGA – Université Paris 13  
chaussard@math.univ-paris13.fr



## L'objectif de ce cours



## Deux aspects importants de la programmation

Les éléments d'un programme informatique peuvent se répartir en deux catégories :

- Les données de ce programme, qui sont stockées dans des variables, des tableaux, des fichiers, des ports, ...
- Les instructions qui agissent sur les données : addition, soustraction, multiplication, recopie, ...

Tout programme consiste à **combiner des instructions** qui agissent sur des données afin de réaliser une tâche spécifique à l'aide de l'ordinateur (tester si un nombre est premier, détecter des éléments dans une image, etc...)

3



## Deux aspects importants de la programmation

Vous avez vu en cours d'algorithme **les instructions** à utiliser pour parvenir à réaliser certaines tâches dans des temps assez courts (tri de tableau, recherche d'éléments, ...).

Ex : Rechercher un élément dans un tableau trié

> **Recherche dichotomique**

L'objectif de ce cours sera de voir les différents moyens de stockage des données (**les structures de données**) permettant de réaliser certaines tâches rapidement.

Ce cours pourrait s'appeler « Algorithmique du stockage de données ».

4



# Les variables entières en C



## Les variables entières en C

### Les char

Type	Entier
Taille mémoire	8 bits = 1 octet
Signé ou non signé ?	Ca dépend de l'environnement utilisé... Il faut utiliser les mots clef <code>signed</code> ou <code>unsigned</code> pour le spécifier (voir ci-dessous)
Plage de valeurs	$[0 ; 2^8 - 1] = [0 ; 255]$ si non signé $[-2^{8-1} ; 2^{8-1} - 1] = [-128 ; 127]$ si signé
Identificateur d'affichage (printf)	%c pour afficher le caractère correspondant de la table ASCII %hh <i>i</i> ou %hhd pour afficher la valeur numérique signée %hu pour afficher la valeur numérique non signée

```
unsigned char c; //On définit un char non signé
signed char d; //On définit un char signé
```



## Les short

Type	Entier
Taille mémoire	16 bits = 2 octets
Signé ou non signé ?	Signé par défaut, il faut utiliser le mot clef <code>unsigned</code> pour le rendre non signé.
Plage de valeurs	$[0 ; 2^{16} - 1] = [0 ; 65\ 535]$ si non signé $[-2^{16-1} ; 2^{16-1} - 1] = [-32\ 768 ; 32\ 767]$ si signé
Identificateur d'affichage (printf)	%hi ou %hd pour afficher la valeur signée %hu pour afficher la valeur non signée

```
short e; //On définit un short signé
unsigned short f; //On définit un short non signé
```



## Les int

Type	Entier
Taille mémoire	32 bits = 4 octets
Signé ou non signé ?	Signé par défaut, il faut utiliser le mot clef <code>unsigned</code> pour le rendre non signé.
Plage de valeurs	$[0 ; 2^{32} - 1] = [0 ; 4\ 294\ 967\ 295]$ si non signé $[-2^{32-1} ; 2^{32-1} - 1] = [-2\ 147\ 483\ 648 ; 2\ 147\ 483\ 647]$ si signé
Identificateur d'affichage (printf)	%i ou %d pour afficher la valeur signée %u pour afficher la valeur non signée

```
int g; //On définit un int signé
unsigned int h; //On définit un int non signé
```



## Les long long

Type	Entier
Taille mémoire	64 bits = 8 octets
Signé ou non signé ?	Signé par défaut, il faut utiliser le mot clef <code>unsigned</code> pour le rendre non signé.
Plage de valeurs	$[0 ; 2^{64} - 1] \approx [0 ; 18 \times 10^{18}]$ si non signé $[-2^{64-1} ; 2^{64-1} - 1] \approx [-18 \times 10^{18} ; 18 \times 10^{18}]$ si signé
Identificateur d'affichage (printf)	<code>%lli</code> ou <code>%ld</code> pour afficher la valeur signée <code>%lu</code> pour afficher la valeur non signée

```
long long g; //On définit un int signé
unsigned long long h; //On définit un int non signé
```



## Les booléens

On peut implémenter des booléens en utilisant un autre type de données entier (comme des char) et en considérant que toute valeur différente de 0 est vraie.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    unsigned char b=5;
    if(b)
        printf("b est vraie\n");
    return 0;
}
```

A l'écran, on aura :

```
b est vraie
```



## Les booléens

Mais là, on aura quoi ?

```
#include <stdio.h>

int main(int argc, char **argv)
{
    unsigned char b=256;
    if(b)
        printf("b est vraie\n");
    return 0;
}
```

A l'écran, on n'aura rien car la valeur entrée dans b dépasse sa capacité, et ce dernier vaut 0 (overflow).



## Les booléens

Pour corriger ce problème, il existe un type pour les booléens **où toute valeur non nulle est interprétée comme vraie** (en fait, comme 1).

L'identificateur `_Bool` est utilisé.

Il faut importer la bibliothèque `stdbool.h`



## Les booléens

Ce qui donne :

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

int main(int argc, char **argv)
{
    _Bool b=256;
    if(b)
        printf("b est vraie\n");
    return 0;
}
```

A l'écran, on aura bien

b est vraie

car on a placé dans b une valeur non nulle : il devient donc vrai.



## Les booléens

Et que donne ceci ?

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

int main(int argc, char **argv)
{
    _Bool b=1;
    b=b+1; ←
    if(b)
        printf("b est vraie\n");
    return 0;
}
```

Ici, b+1 vaut 2, qui est non nul...  
On place donc 1 (vrai) dans b.

On peut interpréter, avec les variables booléennes, le + comme le « ou » logique, et le \* comme le « et » logique (chose qui peut ne pas fonctionner avec des char en raison des dépassemens de capacité).

A l'écran, on aura

b est vraie



## Les long

Les longs sont un type entier, de 4 octets sous Windows ou Linux 32 bits, et 8 octets sous OSX... (aie)

Sur certaines architectures exotiques (super calculateurs), tous les types entiers font 8 octets...

En réalité, le standard C ne spécifie pas la taille mémoire que doit avoir un type de données, mais plutôt la taille minimale qu'il doit avoir !

Par exemple, les int sont garantis de faire 16 bits (et non 32) !  
Faire un code portable est un vrai casse-tête !



## La bibliothèque stdint.h

Heureusement, il existe une bibliothèque implémentant des types de variables dont la taille exacte est spécifiée dans le nom : la bibliothèque **stdint.h**

Le nom du type de donnée entière se construit ainsi :

**uint<taille>\_t**

On écrit le « u » si on souhaite un type de données non signé

On écrit ici la taille de la variable en bits (8, 16, 32 ou 64)



## La bibliothèque stdint.h

Heureusement, il existe une bibliothèque implémentant des types de variables dont la taille exacte est spécifiée dans le nom : la bibliothèque `stdint.h`

Le nom du type de donnée entière se construit ainsi :

**uint<taille>\_t**

Par exemple, si on souhaite déclarer un entier non signé de 16 bits :

```
#include <stdint.h>
...
uint16_t i; //i est un entier non signé sur 16 bits
```



## La bibliothèque stdint.h

Heureusement, il existe une bibliothèque implémentant des types de variables dont la taille exacte est spécifiée dans le nom : la bibliothèque `stdint.h`

Le nom du type de donnée entière se construit ainsi :

**uint<taille>\_t**

Qu'écrire si on souhaite un entier signé sur 64 bits ?

```
int64_t e; //e est un entier signé sur 64 bits
```



# Les variables réelles en C



## Les variables réelles en C

### Les float

Type	Réel
Taille mémoire	32 bits = 4 octets
Signé ou non signé ?	Signé
Plage de valeurs	$[-3.4 \times 10^{38} ; 3.4 \times 10^{38}]$
Identificateur d'affichage (printf)	%f

```
float f; //On définit un réel de type flottant
```



## Les double

Type	Réel
Taille mémoire	64 bits = 8 octets
Signé ou non signé ?	Signé
Plage de valeurs	$[-1.7 \times 10^{308} ; 1.7 \times 10^{308}]$
Identificateur d'affichage (printf)	%lf

```
double d; //on définit un réel de type double
```



## Les long double

Type	Réel
Taille mémoire	96 bits = 12 octets (des fois, 128 bits = 16 octets)
Signé ou non signé ?	Signé
Plage de valeurs	$[-beaucoup; +beaucoup]$
Identificateur d'affichage (printf)	%Lf

```
long double ld; //on définit un réel de type long double
```



## Précision des floats

Que donne ce code ?

```
int main(int argc, char **argv)
{
    float y;
    y=0.1;
    printf("%f", y);
    return 0;
}
```

A l'écran, on a :

0.100000

Tout semble aller bien...



## Précision des floats

Affichons avec un peu plus de précision :

```
int main(int argc, char **argv)
{
    float y;
    y=0.1;
    printf("%.20f", y);
    return 0;
}
```

A l'écran, on a :

0.10000000149011611938

Oh oh...



## Précision des floats

Que s'est-il passé ?

En fait, un float est sur 32 bits : comme pour les entiers, on ne pourra coder sur une telle variable seulement  $2^{32}$  ( $\approx 4 \times 10^9$ ) valeurs différentes.

Or, dans l'intervalle  $[-3.4 \times 10^{38} ; 3.4 \times 10^{38}]$ , il y a un peu plus que 4 milliards de valeurs possibles... Comme c'est un intervalle de  $\mathbb{R}$ , il y a une infinité de valeurs !

**Conclusion** : les float, comme les double, **ne peuvent représenter que certains nombres réels** dans l'intervalle  $[-3.4 \times 10^{38} ; 3.4 \times 10^{38}]$ .



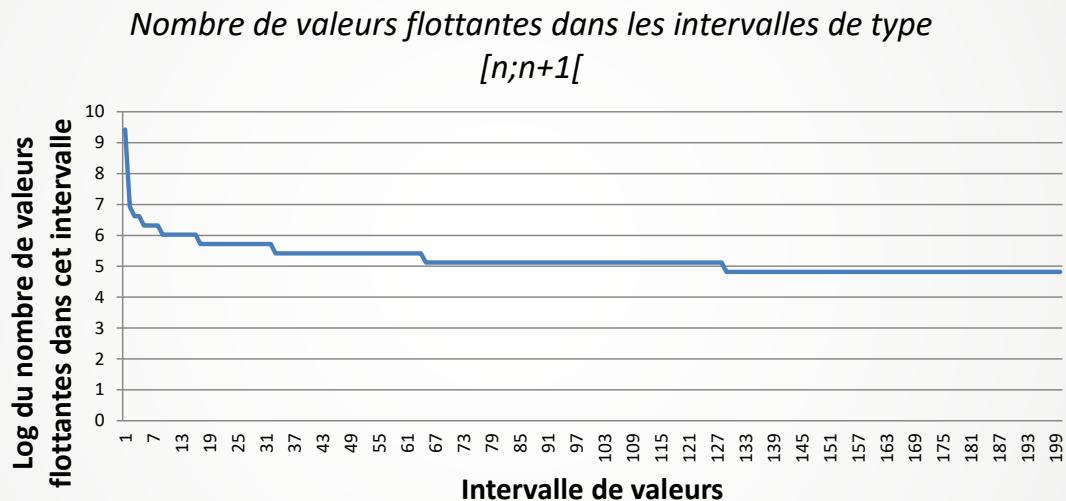
## Précision des floats

Plus la valeur à représenter par un float est grande, et plus l'imprécision sera importante.

Les float, comme les double, se « concentrent » autour des petites valeurs, et sont éparpillés ensuite vers les plus grandes valeurs...



## Précision des floats



Les valeurs flottantes sont éparpillées lorsque les valeurs entières augmentent : seulement 65536 valeurs flottantes différentes entre 199 et 200 !



## Précision des floats

Le problème de précision avec les float peut se propager dans les calculs :

```
int main(int argc, char **argv)
{
    float f = 0;
    int i;

    for(i=0; i<1000; i++)
    {
        f+=0.1;
    }
    printf("%f\n", f);
    return 0;
}
```

Ce qui ne produit pas 100, mais

99.999046



## Précision des floats

L'addition de deux flottants de « différentes échelles » fait que l'un mange en général l'autre :

```
int main(int argc, char **argv)
{
    float f = 0.0000000001;
    float e = 1.0 + f;

    printf("%.15f\n", e);
    return 0;
}
```

Le résultat ici donne :

```
1.0000000000000000
```



## Précision des floats

Ceci peut poser de plus graves problèmes à terme :

```
int main()
{
    double i;

    for(i=0.0; i<1000000; i+=0.0000001)
    {

    }

    return EXIT_SUCCESS;
}
```

La boucle ici ne se termine jamais, car lorsque *i* est assez grand, son incrémentation ne le modifie pas.



## Précision des doubles

Le problème est le même avec les doubles, même si du fait de leur plus grande capacité mémoire, les problèmes surviennent moins rapidement qu'avec les float.

En conclusion : le calcul en entier pose des limites strictes en terme de valeur maximale autorisée, mais ces dernières peuvent être facilement contrôlées.

Le calcul en réel **pose des limites en termes de précision**, qui ne sont pas toujours correctement évaluées par les programmeurs : soyez prudents sur les résultats obtenus après des opérations sur les réelles.



## Un exemple à ne pas suivre...

Par exemple, dans votre code, évitez de tester l'égalité stricte de deux réels :

```
int main(int argc, char **argv)
{
    float f = 1.0;

    while(f!=0)
    {
        f=f-0.00001;
        printf("%f\n", f);
    }

    printf("Fini\n");
    return 0;
}
```

Ce programme boucle à l'infini...



## Un exemple à suivre...

Faites plutôt des tests basés sur des inégalités :

```
int main(int argc, char **argv)
{
    float f = 1.0;

    while(f>0)
    {
        f=f-0.00001;
        printf("%f\n", f);
    }

    printf("Fini\n");
    return 0;
}
```

Ce programme s'arrêtera lorsque *f* sera suffisamment proche de 0.



13 Les tableaux statiques



# Les tableaux statiques

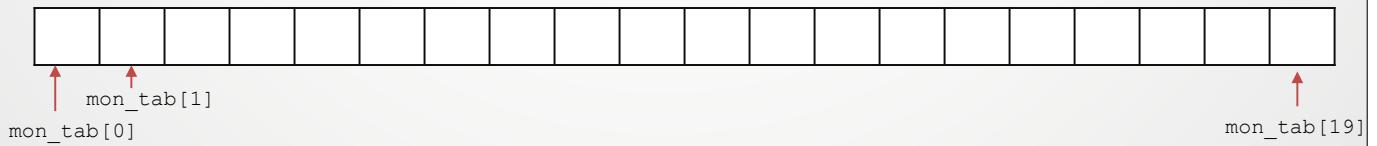
On définit, en C, un tableau ainsi :

```
<type> nom[nb de cases];
```

Par exemple, pour définir un tableau d'entier non signé sur 32 bits, de 20 cases, on fera :

```
uint32_t mon_tab[20];
```

Ceci permet de construire, dans la mémoire, 20 variables consécutives en mémoire, la première s'appelant `mon_tab[0]`, la seconde `mon_tab[1]`, et la dernière étant `mon_tab[19]`.



# Les tableaux statiques

Quel avantage d'utiliser des tableaux par rapport à déclarer n variables ?

- Evidemment, le premier avantage est la rapidité à coder : au lieu de déclarer n variables dans le code, on en déclare une seule qui contiendra les n variables que l'on souhaite.
- Avec un tableau, les n variables sont créées conséutivement dans la mémoire, ce qui permet d'utiliser des algorithmes efficaces pour y trouver certaines valeurs (dichotomie) ou pour les trier (tri fusion). Ce n'est pas le cas avec n variables créées séparément.
- L'utilisation des tableaux permet une nomenclature très efficace des variables, permettant de parcourir toutes les variables avec une simple boucle.



## Petit rappel

On ne doit pas, en C (norme C90), déclarer un tableau statique dont la taille dépend d'une variable. La principale raison est **l'incapacité à vérifier si le tableau a été créé correctement** :

```
int main(int argc, char **argv)
{
    uint32_t i, n = 10000000;
    uint32_t mon_tab[n];

    for(i=0; i<n; i++)
    {
        mon_tab[i]=0;
    }
    return 0;
}
```

Ici, le programme a de grandes chances d'échouer avec une segmentation fault, alors que le tableau demandé ne faisait que 40Mo.



## Les tableaux statiques : contraintes

- Une fonction **ne peut pas renvoyer un tableau statique** (c'est un choix des responsables du langage C qui pourrait être changé, mais qui demeure aujourd'hui).
- Une fonction **peut prendre en paramètre un tableau statique** : dans ce cas, il est **passé par référence**, c'est-à-dire que tout modification de valeur du tableau dans la fonction sera répercutée dans la fonction appelante :

```
void init(int32_t a[], int32_t taille)
{
    int32_t i;
    for (i=0; i<taille; i++)
    {
        a[i]=4;
    }
}
```

```
int32_t main()
{
    int32_t i;
    int32_t tab[10];
    for (i=0; i<10; i++)
        tab[i]=0;

    init(tab, 10);

    for (i=0; i<10; i++)
        printf("%d\n", tab[i]);
    return EXIT_SUCCESS;
}
```

A la fin du programme, le tableau sera rempli de 4.



## Les tableaux statiques : contraintes

- Une fonction **ne peut pas renvoyer un tableau statique** (c'est un choix des responsables du langage C qui pourrait être changé, mais qui demeure aujourd'hui).
- Une fonction **peut prendre en paramètre un tableau statique** : dans ce cas, il est **passé par référence**, c'est-à-dire que tout modification de valeur du tableau dans la fonction sera répercutée dans la fonction appelante.
- Une fonction peut prendre en paramètre un tableau statique à plusieurs dimensions, mais il faut préciser les tailles de toutes les dimensions supérieures à la première (le premier crochet peut être vide, pas les autres) :

```
void f (int32_t p[])      //OK
void f (int32_t p[12])    //OK
void f (int32_t p[][34])  //OK
void f (int32_t p[56][78]) //OK
void f (int32_t p[][][])   //Pas OK, il faut spécifier la valeur dans le second crochet
```



13 Les structures et union



## Les structures

On peut définir, en C, une structure ainsi (en dehors de toute fonction) :

```
struct nom_structure
{
    type1 nom_variable1;
    type2 nom_variable2;
    etc...
};
```

Ici, cela crée une boîte dans laquelle sont stockées n variables de type identifié (type1, type2, etc). Pour déclarer une variable de ce type, on fera (dans une fonction) :

```
struct nom_structure x;
```

Ici, on crée une variable x qui est un type composé : x est une boîte possédant plusieurs variables à l'intérieur.



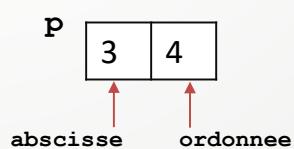
## Les structures

Par exemple, pour définir des coordonnées d'un point :

```
struct coordonnees
{
    double abscisse;
    double ordonnee;
};
```

Puis, dans le code principal, on fera :

```
struct coordonnees p;
p.abscisse = 3;
p.ordonnee = 4;
```





## Les structures

On peut aussi définir la structure de façon à ne plus avoir à écrire le mot clef struct :

```
typedef struct coord
{
    double abscisse;
    double ordonnee;
} coordonnees;
```

Puis, dans le code principal, on fera :

```
coordonnees p;
p.abscisse = 3;
p.ordonnee = 4;
```



## Les structures

On peut finalement voir une structure comme un tableau statique où les cases ne contiennent pas forcément le même type de données...



## Les union

Les union se définissent comme les structures, mis à part que le mot clef **struct** est remplacé par **union** :

```
union nom_structure
{
    type1 nom_variable1;
    type2 nom_variable2;
    etc...
};
```

Puis, dans le code principal, on fera :

```
union nom_structure x;
```



## Les union

La différence principale vient du fait que, dans une union, une seule « case mémoire » est allouée, et utilisée pour stocker une seule variable sous « différentes formes ». Par exemple :

```
union test
{
    int32_t a
    double b;
};
```

Et dans le main :

```
union test x;
x.a = -3;
printf("%d %.20lf\n", x.a, x.b);
```

Cela affichera

```
-3 0.00000000000000000000000000
```



## Les union

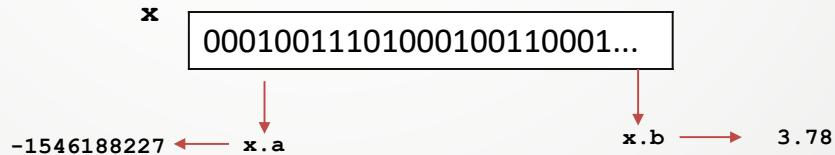
Si on fait maintenant :

```
x.b = 3.78;  
printf("%d %.20lf\n", x.a, x.b);
```

Cela affichera

```
-1546188227 3.7799999999999980460
```

x.a et x.b sont la même case mémoire, mais interprétée différemment (la première est vue comme un entier, la seconde est vue comme un double).



## Les union

Application pratique : afficher tous les double possibles

```
union test  
{  
    uint64_t a;  
    double b;  
};  
  
int main()  
{  
    union test x;  
  
    x.a = 1;  
    while (x.a != 0)  
    {  
        printf("%lf\n", x.b);  
        x.a++;  
    }  
  
    printf("%lf\n", x.b);  
    return EXIT_SUCCESS;  
}
```



# N13

## Les énumérations



## Les énumérations

### Les énumérations

Les énumérations sont utiles lorsque l'on souhaite définir une variable qui ne peut prendre que certaines valeurs d'une liste précise.

On la définit ainsi :

```
enum nom_enumeration {PREMIER_ELEMENT, SECOND_ELEMENT, ..., NIEME_ELEMENT};
```

Par exemple, si l'on souhaite définir un type énuméré pour les jours de la semaine, on fera :

```
enum jour {LUNDI, MARDI, MERCRIDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

Ici, même si ce n'est pas du tout utile de le savoir, LUNDI vaut 0, MARDI vaut 1, etc.



## Déclaration d'une énumération

On peut ensuite, dans une fonction, déclarer une variable du type de l'énumération :

```
enum jour j;
```

Pour donner une valeur à j, on pourra faire :

```
j = JEUDI;
```

Si l'on attribue à j une autre valeur que celle de l'énumération, on aura une erreur.



## Enumérations et switch/case

Pour tester les différentes valeurs d'une énumération, on utilisera un switch/case :

```
j=MERCREDI;
switch(j)
{
    case (LUNDI): printf("C'est Lundi\n"); break;

    case (MARDI): printf("C'est Mardi\n"); break;

    case (MERCREDI): printf("C'est Mercredi\n"); break;

    case (JEUDI): printf("C'est Jeudi\n"); break;

    case (VENDREDI): printf("C'est Vendredi\n"); break;

    case (SAMEDI): printf("C'est Samedi\n"); break;

    case (DIMANCHE): printf("C'est Dimanche\n"); break;
}
```

Ici, le programme affichera bien « C'est Mercredi ».



## Enumérations et switch/case

L'intérêt des énumérations et switch/case réside dans le cas où l'on oublie un cas, sans mettre un cas par défaut :

```
switch(j)
{
    case (LUNDI): printf("C'est Lundi\n"); break;
    case (MARDI): printf("C'est Mardi\n"); break;
    case (MERCREDI): printf("C'est Mercredi\n"); break;
    case (VENDREDI): printf("C'est Vendredi\n"); break;
    case (SAMEDI): printf("C'est Samedi\n"); break;
    case (DIMANCHE): printf("C'est Dimanche\n"); break;
}
```

A la compilation, on aura un avertissement :

```
warning:enumeration value 'JEUDI' not handled in switch
```



## Enumérations vs define

Pourquoi ne pas utiliser des define pour les jours de la semaine ?

```
#define LUNDI 0
#define MARDI 1
#define MERCREDI 2
#define JEUDI 3
#define VENDREDI 4
#define SAMEDI 5
#define DIMANCHE 6

int main()
{
    uint32_t j;

    j = MERCREDI;

    return EXIT_SUCCESS;
}
```



## Enumérations vs define

- Utiliser des define amène au risque que deux jours différents aient la même valeur. Avec les énumérations, ce risque n'existe pas.
- Utiliser des enumérations est totalement adapté pour des variables contenant des éléments faisant partie d'une liste. C'est une indication, pour un relecteur de votre code, de l'objectif de votre variable.
- Grâce au switch case et au compilateur, nous avons une garantie que tous les cas d'une énumération sont traités, ce qui n'est pas le cas avec des define. Ceci est très utile si l'on peut être amené, plus tard dans le code, à rajouter des éléments dans l'énumération.



## Déclaration d'un type énuméré

Comme pour les structures, on peut utiliser un raccourci syntaxique pour déclarer un type énuméré :

```
typedef enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE} jour;
```

On peut alors déclarer une variable de type jour en faisant :

```
jour j;
```

# Programmation avancée et Structures de données

## Les pointeurs et tableaux dynamiques

### Table des matières

<b>1 Qu'est-ce qu'un pointeur ?</b>	<b>1</b>
1.1 Déclaration d'un pointeur . . . . .	2
1.2 Attribution de valeur . . . . .	2
1.3 Modification de valeur . . . . .	2
1.4 Fiche d'identité des pointeurs . . . . .	4
<b>2 Utilisation des pointeurs pour les fonctions</b>	<b>4</b>
2.1 Passage par valeur . . . . .	4
2.2 Passage par référence . . . . .	4
2.3 Passage par valeur ou par référence ? . . . . .	6
<b>3 Pointeurs et tableaux dynamiques</b>	<b>6</b>
3.1 L'allocation mémoire . . . . .	6
3.2 Allocation statique . . . . .	6
3.3 La fonction malloc - l'allocation mémoire à la demande . . . . .	6
3.4 L'opérateur sizeof . . . . .	8
3.5 Tableaux dynamiques 1d . . . . .	9
3.5.1 Allocation de tableaux dynamiques . . . . .	9
3.5.2 Accès au tableau dynamique - la "mauvaise" façon . . . . .	10
3.5.3 Accès au tableau dynamique - la bonne façon . . . . .	11
3.5.4 Attention : pas de pointeurs retournés vers des zones locales... . . . . .	12
3.6 Même les pointeurs sont passés par valeur . . . . .	13
3.7 Autres fonctions à connaître . . . . .	14
3.7.1 Libération de la mémoire allouée . . . . .	14
3.7.2 Les fonctions calloc, realloc et memcpy . . . . .	15
3.8 Tableaux dynamiques 2d (et nd par extension) . . . . .	15
3.8.1 Structure d'un tableau dynamique 2d . . . . .	15
3.8.2 Déclaration d'un tableau dynamique 2d . . . . .	15
3.8.3 Allocation d'un tableau dynamique 2d . . . . .	16
3.8.4 Ecriture d'une donnée dans un tableau 2d . . . . .	16
3.8.5 Libération mémoire d'un tableau 2d . . . . .	16
3.8.6 Les tableaux nd . . . . .	17
3.9 Allocation de structures et de tableaux de structures . . . . .	17
3.9.1 Allocation d'une structure . . . . .	17
3.9.2 Allocation d'un tableau de structures . . . . .	17
<b>4 Comprendre ce qu'attend une fonction selon ses paramètres</b>	<b>18</b>
<b>5 Avec de grands pouvoirs viennent de grandes responsabilités</b>	<b>19</b>

### 1 Qu'est-ce qu'un pointeur ?

Un pointeur est une variable (tout comme `int32_t`, `double`, ...) qui contient une valeur. A la différence d'une variable "classique", un pointeur ne contient pas une valeur représentant une quantité mais représentante une adresse dans la mémoire du programme. En général, on trouvera, à l'adresse contenue par un pointeur, une donnée utile pour un programme.

## 1.1 Déclaration d'un pointeur

Pour déclarer un pointeur, il faut tout d'abord décider de ce que contiendra la donnée indiquée par l'adresse du pointeur. On déclarera un pointeur ainsi :

```
1 <type de donnee>* nom_pointeur;
```

Si l'on souhaite déclarer un pointeur **ptr** qui contiendra l'adresse d'une variable de type **uint32\_t**, on fera :

```
1 uint32_t* ptr;
```

Un pointeur dont l'adresse ne pointe pas vers un type particulier de données pourra être déclaré avec le mot clef **void\***.

## 1.2 Attribution de valeur

Lorsqu'on déclare une variable, cette dernière contient une valeur non initialisée que l'on ne doit pas exploiter. De même, lorsqu'on déclare un pointeur, ce dernier contient une adresse non initialisée, qui pointe vers une zone mémoire indéfinie du programme. Afin d'initialiser un pointeur, il faudra lui attribuer l'adresse d'une variable du programme à l'aide du symbole **&**, comme ainsi :

```
1 uint32_t i = 7;
2 uint32_t* p;
3 p = &i; //On place dans p l'adresse de i
```

A la fin de ce programme, la variable **p** contiendra l'adresse de la variable **i**, qui elle-même contient la valeur 7. La figure 1 montre l'état de la mémoire après l'exécution de ce code :

On peut afficher l'adresse contenue par le pointeur **p** ainsi :

```
1 printf("%p\n", p);
```

## 1.3 Modification de valeur

Avec un pointeur, on peut modifier la valeur de la variable dont l'adresse est contenue dans le pointeur, à l'aide du symbole **\***. En reprenant l'exemple précédent :

```
1 *p = 4;
```

La variable **\*p** n'indique pas le pointeur **p**, mais la variable dont l'adresse est contenue dans **p**. La figure 2 montre l'état de la mémoire à l'issue de cette commande :

On peut aussi modifier la valeur de l'adresse contenue dans un pointeur, soit directement avec le symbole **=**, soit en faisant de l'arithmétique :

```
1 uint32_t* t;
2 t = p+1;
```

Le pointeur **t** contiendra l'adresse de **p** plus une fois la taille d'un **uint32\_t**. Le pointeur **t** contiendra ainsi l'adresse de l'entier situé juste à côté de **i**, comme montré sur la figure 3 :

Si l'on avait fait :

FIGURE 1 – Etat de la mémoire après l'attribution de l'adresse d'une variable **i** dans un pointeur **p**

FIGURE 2 – Etat de la mémoire après l'attribution de la valeur 4 à l'adresse pointée par p

```
1 t = p+3;
```

Dans ce cas, nous aurions, dans la variable **t**, l'adresse de **p** plus trois fois la taille d'un **uint32** **t**. Le pointeur **t** contiendra ainsi l'adresse de l'entier situé "à trois entiers de distance" de **i**. La figure 4 montre l'état de la mémoire à l'issue de cette commande :

Bien que l'arithmétique des pointeurs ne soit pas utilisée telle quelle dans un code, elle est très utile pour comprendre le fonctionnement des tableau dynamiques plus loin dans ce cours...

FIGURE 4 – Etat de la mémoire après l'attribution, dans t, de la valeur p+3

## 1.4 Fiche d'identité des pointeurs

Type	
Taille	
Signé / Non signé	Non signé
Plage de valeurs	Dépend de la mémoire disponible sur la machine
Identificateur d'affichage	%p

## 2 Utilisation des pointeurs pour les fonctions

En général, on ne peut pas modifier les paramètres d'une fonction puis récupérer ces modifications dans le corps principal du programme. Ceci est dû au mécanisme de passage par valeur.

### 2.1 Passage par valeur

Considérez le code suivant :

```
1 void swap(uint32_t a, uint32_t b)
2 {
3     uint32_t tmp = a;
4     a = b;
5     b = tmp;
6     // Que penser de a = a+b et b = a-b ?
7 }
8
9 int main()
10 {
11     uint32_t i, j;
12     i=3;
13     j=4;
14     swap(i, j);
15     printf("i vaut %d et j vaut %d\n", i, j);
16     return EXIT_SUCCESS;
17 }
```

Dans cet exemple, l'affichage montre que les valeurs contenues dans les variables **i** et **j** n'ont pas été échangées. En vérité, les variables **i** et **j** ont été passées, depuis la fonction principale à la fonction **swap**, par valeur. Ceci signifie que ce ne sont pas les variables **i** et **j** qui sont transmises à la fonction **swap**, mais une copie de leur valeur, comme expliqué sur la figure 5.

Une solution à ce problème serait de renvoyer en sortie les deux entiers **i** et **j** et de les "capturer" dans la fonction principale, à l'aide d'un tableau ou d'une structure de données (car une fonction ne peut renvoyer qu'une seule variable).

### 2.2 Passage par référence

Le passage par référence est un terme décrivant l'utilisation des pointeurs pour permettre à une fonction de modifier ses propres paramètres. Considérez le code suivant :

```
1 void swap(uint32_t *a, uint32_t *b)
2 {
3     uint32_t tmp = *a;
4     *a = *b;
5     *b = tmp;
6 }
7
8 int main()
9 {
10     uint32_t i, j;
11     i=3;
12     j=4;
13     swap(&i, &j);
14     printf("i vaut %d et j vaut %d\n", i, j);
```

FIGURE 5 – Exemple de fonction appelée avec passage des paramètres par valeur

```
15     return EXIT_SUCCESS;  
16 }
```

Ici, nous avons bien un passage par valeur qui a été effectué : ce sont les valeurs des adresses des variables **i** et **j** qui ont été transmises à la fonction **swap**, lui permettant de directement modifier les valeurs de ces variables. Ce mécanisme est illustré sur la figure 6.

### 2.3 Passage par valeur ou par référence ?

Par défaut, toutes les variables en C sont passées par valeur en tant que paramètre d'une fonction. Cependant, il faut noter que les tableaux statiques font exception à la règle : modifier un tableau statique dans une fonction le modifiera aussi dans la fonction principale.

## 3 Pointeurs et tableaux dynamiques

Les tableaux statiques permettent de construire des tableaux avec une taille prédéfinie au moment de la compilation. A l'aide des pointeurs, il est possible de construire des tableaux dont la taille est décidée au moment de l'exécution du programme.

### 3.1 L'allocation mémoire

Dans un programme informatique, on évite toujours d'écrire des données dans des emplacements non prévus pour cela. Dans un programme, lorsqu'on écrit une valeur, on l'écrit en général dans une variable ou un tableau. Ces emplacements sont des zones mémoires allouées par le système d'exploitation au programme, autrement dit des zones mémoires réservées où écrire une donnée.

Ecrire une donnée dans un endroit non alloué par le système d'exploitation peut se révéler "dangereux" pour plusieurs raisons :

- En écrivant dans un endroit qui ne nous a pas été alloué par le système, on peut écrire dans une zone allouée pour une autre donnée, et l'écraser.
- Inversement, on peut se retrouver à écrire la donnée dans un endroit non protégé, ce qui l'exposera à être écrasée plus tard.
- On peut écrire la donnée dans un endroit protégé du système, comme dans le code même du programme ou du système d'exploitation.

En général, si le système détecte un tel comportement dans un programme, il y met fin (sous Linux, l'erreur segmentation fault apparaît alors). Un mécanisme pour écrire dans une zone mémoire non allouée par le système est d'écrire, dans un tableau, dans une case en dehors de ce tableau ; le programme sortira alors de la zone mémoire allouée au tableau, et écrira une donnée dans un endroit où il ne devait pas.

### 3.2 Allocation statique

Dès que l'on déclare une variable (un entier, un double, etc...) ou un tableau statique, une zone mémoire est allouée par le système à ces variables. Ces allocations mémoires, dites statiques, sont implicitement réalisées par le programme, et le programmeur n'a rien à faire d'autre que de déclarer les variables pour pouvoir ensuite les utiliser.

### 3.3 La fonction malloc - l'allocation mémoire à la demande

On peut demander implicitement à un programme de réserver de la mémoire pour des données, en déclarant des variables ou des tableaux statiques dans son code. Dans ce cas, des emplacements mémoire seront réservés pour les variables le temps que la fonction contenant ces variables s'exécute, puis seront libérés.

Il est possible de demander, pendant l'exécution d'un programme, à réserver explicitement une zone mémoire à l'aide de la fonction **malloc** contenue dans la bibliothèque **stdlib.h**. Cette fonction prend en paramètre un entier égal au nombre d'octets que l'on souhaite réserver dans la mémoire, et renvoie l'adresse (un pointeur non typé, donc **void\***) de la zone alors réservée :

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

Si l'on souhaite réserver 2 octets et stocker l'adresse de la zone mémoire allouée dans un pointeur, on fera :

FIGURE 6 – Exemple de fonction appelée avec passage des paramètres par référence

FIGURE 7 – Etat de la mémoire après l'appel de malloc

FIGURE 8 – Etat de la mémoire après la modification de la valeur de \*p

```
1 void *p;  
2 p = malloc(2);
```

La figure 7 schématise l'état de la mémoire à l'issue de l'exécution de ce code.

Si l'on souhaite réserver suffisamment de place pour un entier signé de 4 octets (32 bits), on fera :

```
1 int32_t *p;  
2 p = (int32_t *) malloc(4); //On type le retour de malloc, car on sait que l'on  
pourra y stocker un int32_t
```

On note ici que l'on a typé le pointeur **p**, afin de l'utiliser directement comme un pointeur sur entier 32 bits, rendant l'écriture de la suite du programme plus simple. Par exemple, on pourra faire :

```
1 *p = 8;  
2 printf("%d\n", *p);
```

La figure 8 schématise l'état de la mémoire à l'issue de l'exécution de ce code.

Si l'allocation mémoire échoue, le pointeur renvoyé par **malloc** est égal à 0 : en C, le pointeur valant 0 est appelé **NULL**.

Il faut être prudent avec une zone mémoire allouée explicitement dans un pointeur : comme elle n'est accessible qu'à travers un pointeur, si on perd la valeur du pointeur, on perd la zone mémoire. On peut cependant créer un second pointeur pour sauvegarder l'adresse de la zone mémoire :

```
1 int32_t *t = p;
```

Dans ce cas, on sauvegarde dans **t** l'adresse contenue dans **p**. On pourra donc accéder à la zone mémoire allouée en utilisant **\*p** ou **\*t**. La figure 9 schématise l'état de la mémoire à l'issue de l'exécution de ce code.

### 3.4 L'opérateur sizeof

L'opérateur **sizeof** permet d'obtenir, en octet, la taille mémoire d'un type de variable. L'opérateur renvoie une valeur de type **size\_t**, que l'on pourra stocker dans un entier non signé.

FIGURE 9 – Etat de la mémoire après la création du pointeur t

FIGURE 10 – Etat de la mémoire après l'allocation mémoire sur p

Par exemple, si l'on souhaite afficher la taille d'un double, on pourra faire :

```
1 printf("%u\n", sizeof(double));
```

Ceci affichera (certainement) 8 à l'écran, car un **double** fait (en général) 8 octets.

### 3.5 Tableaux dynamiques 1d

#### 3.5.1 Allocation de tableaux dynamiques

Pour allouer un tableau, il suffit de demander à la fonction **malloc** d'allouer suffisamment d'espace mémoire pour stocker toutes les valeurs du tableau.

Imaginons que l'on ait une variable **n** qui contiennent le nombre de cases d'un tableau d'entiers non signés de 4 octets que l'on souhaite allouer. On a donc besoin d'allouer  $n \times \text{sizeof}(\text{uint32\_t})$  octets dans la mémoire. On fera alors :

```
1 uint32_t n = 30;
2 uint32_t *p;
3
4 p = (uint32_t *) malloc( n * sizeof(uint32_t) );
5 if( p==NULL) //Si l'allocation memoire a echoue
6 {
7     printf("L' allocation memoire a echoue\n");
8     return EXIT_FAILURE;
9 }
```

Dans ce code, on alloue une zone mémoire de qui pourra contenir  $n$  entiers de 32 bits, c'est à dire que l'on a alloué  $4n$  octets. On teste si l'allocation mémoire s'est bien déroulée en comparant le pointeur **p** avec la valeur **NULL**. La figure 10 schématisse l'état de la mémoire à l'issue de l'exécution de ce code.

Comment peut-on accéder à cette zone mémoire et, surtout, pourquoi parle-t-on de tableau ?

FIGURE 11 – Etat de la mémoire après la modification de la valeur de \*p

FIGURE 12 – Etat de la mémoire après la modification de la valeur de \*(p+1) ou \*t

### 3.5.2 Accès au tableau dynamique - la "mauvaise" façon

Le résultat de l'allocation nous renvoie un pointeur, c'est à dire l'adresse de la zone mémoire allouée par le système à l'aide de **malloc**. En reprenant l'exemple précédent, on peut écrire/lire facilement les quatre premiers octets de la zone mémoire en faisant :

```
1 *p = 8;
2 printf("Les 4 premiers octets de la zone memoire valent %d\n", *p);
```

Ici, le programme devrait afficher 8, car nous avons écrit 8 sur les 4 octets pointés par l'adresse contenue dans **p**. La figure 11 schématise l'état de la mémoire à l'issue de l'exécution de ce code.

Afin d'écrire dans les 4 octets suivants, il faut déplacer le pointeur **p** de 4 octets ; comme on l'a vu précédemment, on peut faire :

```
1 uint32_t* t;
2 t = p+1;
3 *t = 16;
4 printf("Les 4 premiers octets de la zone memoire valent %d\n", *p);
5 printf("Les 4 octets suivants de la zone memoire valent %d\n", *t);
```

On peut aussi éviter d'utiliser un second pointeur, et faire directement :

```
1 *(p+1) = 16;
2 printf("Les 4 premiers octets de la zone memoire valent %d\n", *p);
3 printf("Les 4 octets suivants de la zone memoire valent %d\n", *(p+1));
```

La figure 12 schématise l'état de la mémoire à l'issue de l'exécution de ce code.

On peut ainsi balayer toute la zone mémoire allouée et y écrire/lire des données, comme avec un tableau statique. Si l'on souhaite, par exemple, remplir la zone de 0, on fera :

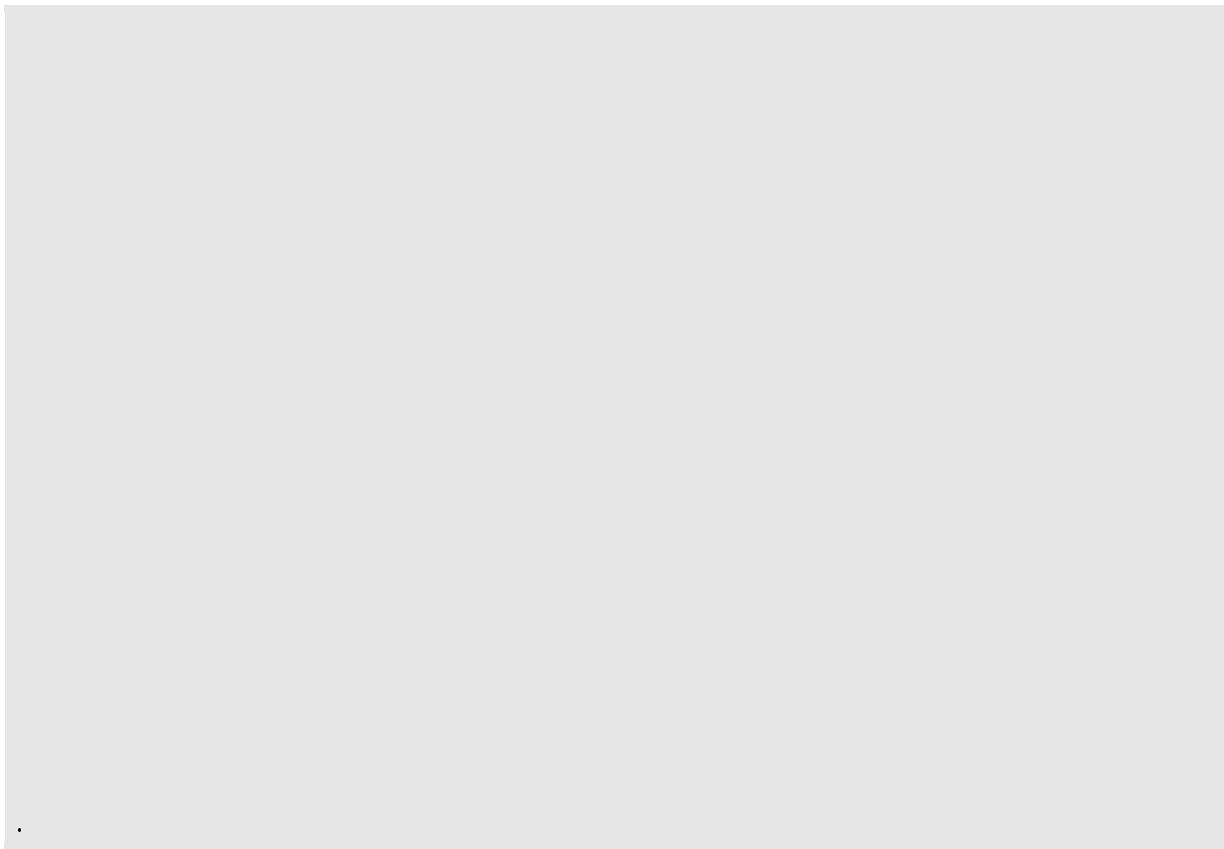
```
1 for (i=0; i<n; i++)
2 {
3     *(p+i) = 0;
```

FIGURE 13 – Etat de la mémoire après l'exécution du code d'allocation d'un tableau de **n** réels

```
4 }
```

### Questions

A votre tour : écrivez un programme demandant à l'utilisateur d'entrer une valeur entière non signée **n** ainsi qu'une valeur réelle **k**, et qui construit un tableau de réels de **n**, et dont chaque case vaut **k**.



La figure 13 schématise l'état de la mémoire à l'issue de l'exécution de ce code.

#### 3.5.3 Accès au tableau dynamique - la bonne façon

Le langage C possède néanmoins un raccourci syntaxique permettant d'accéder de manière plus agréable aux zones mémoire. Si, comme dans les exemples précédents, on considère que **p** est un pointeur vers une zone mémoire allouée, alors les deux lignes suivantes de code sont équivalentes :

```
1 *(p+7) = 4;
```

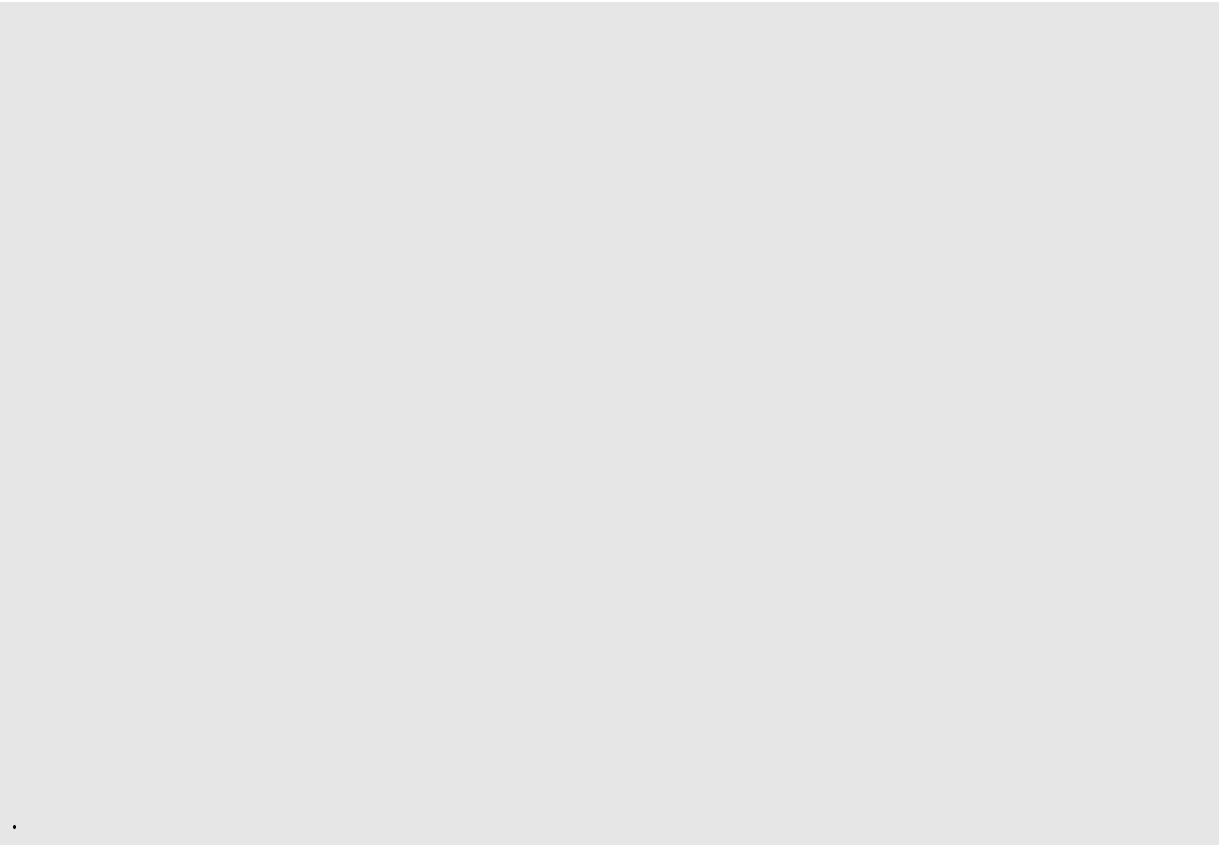
```
1 p[7] = 4;
```

Le pointeur **p**, pour peu qu'il pointe vers une zone mémoire correctement allouée, peut donc être utilisé comme un tableau. Si, comme précédemment, **p** pointe vers une zone mémoire de  $n \times \text{sizeof}(\text{uint32\_t})$  octets, alors, pour initialiser toutes les entiers de cette zone à 0, on fera :

```
1 for (i=0; i<n; i++)
2 {
3     p[i] = 0;
4 }
```

### Questions

A votre tour : ré-écrivez le programme demandant à l'utilisateur d'entrer une valeur entière non signée **n** ainsi qu'une valeur réelle **k**, et qui construit un tableau de réels de **n**, et dont chaque case vaut **k**.



#### 3.5.4 Attention : pas de pointeurs retournés vers des zones locales...

Considérez le code suivant :

```
1 #define N 100
2
3 int main()
4 {
5     uint32_t *p;
6
7     p = alloc_bizarre();
8     return EXIT_SUCCESS;
9 }
10
11 uint32_t* alloc_bizarre()
```

FIGURE 14 – Etat de la mémoire pendant et après l'exécution la fin de la fonction *alloc\_bizarre*

```

12 {
13     uint32_t tab[N];
14     uint32_t i;
15
16     for(i=0; i<N; i++)
17         tab[i] = 0;
18
19     return &(tab[0]);
20 }
```

Le problème dans le code précédent est que, une fois la fonction **alloc\_bizarre** terminée, toutes les variables statiques de la fonction, y compris le tableau *tab*, sont détruites. Une fois de retour dans le main, le pointeur *p* pointe donc à une adresse non allouée, comme illustré sur la figure 14.

Toutes les variables (variables, tableau, structures, ...) déclarées directement dans une fonction sont *locales* à la fonction : une fois celle-ci terminée, ses variables locales sont détruites. Les zones mémoires allouées avec **malloc** ne sont pas locales, et sont allouées dans la mémoire globale du programme : elles persistent même après la fin de la fonction où avait été appelé le **malloc**.

### 3.6 Même les pointeurs sont passés par valeur...

Regardez le code ci-dessous, où la fonction **mon\_alloc** réalise une allocation dynamique d'un tableau qu'elle range dans un pointeur passé en paramètre :

```

1 int main()
2 {
3     uint32_t taille = 100;
4     uint32_t *p;
5
6     mon_alloc(p, taille);
7     return EXIT_SUCCESS;
8 }
9
10 void mon_alloc(uint32_t* ptr, uint32_t nb_elem)
11 {
12     ptr = malloc(nb_elem * sizeof(uint32_t));
13 }
```

Ici, comme montré sur la figure 15, la valeur attribuée au pointeur *ptr* dans la fonction **mon\_alloc** ne modifie pas la valeur du pointeur *p* de la fonction principale : le tableau construit dans la fonction est perdu, et le pointeur *p* n'adresse pas une zone mémoire valide dans la fonction principal : il est dangereux (et interdit) de l'utiliser comme un tableau pour y écrire ou lire des données.

Si on souhaitait faire une fonction qui alloue un tableau dynamique, il existe deux possibilités : la première serait de passer le pointeur par référence, pour en modifier la valeur et que cette modification soit "acquise" par la fonction principale. La seconde serait de renvoyer la valeur du pointeur ; la fonction ne prendrait qu'un seul paramètre. Les codes ci-dessous illustrent ces deux solutions.

FIGURE 15 – Appel d'une fonction avec un pointeur passé par valeur ou par référence

```

1 void mon_alloc_v2(uint32_t** ptr, uint32_t nb_elem)
2 {
3     *ptr = malloc(nb_elem * sizeof(uint32_t));
4 }

1 uint32_t* mon_alloc_v3(uint32_t nb_elem)
2 {
3     uint32_t *ptr = malloc(nb_elem * sizeof(uint32_t));
4     return ptr;
5 }
```

### 3.7 Autres fonctions à connaître

#### 3.7.1 Libération de la mémoire allouée

La mémoire allouée dynamiquement n'est pas détruite (ou libérée) une fois que la fonction dans laquelle s'est déroulée l'allocation se termine. La mémoire allouée reste réservée jusqu'à ce que le programme principal se termine, ou que l'utilisateur appelle la fonction **free** explicitement sur la zone mémoire.

La fonction **free** a pour effet de forcer le système d'exploitation à libérer une zone mémoire. La fonction prend en paramètre une adresse (un pointeur) du début d'une zone mémoire allouée à l'aide d'une fonction d'allocation dynamique. Si l'on passe à la fonction une zone qui n'a pas été allouée dynamiquement, ou bien qui n'est pas le début (mais le milieu par exemple) d'une zone allouée dynamiquement, alors le programme s'interrompra brutalement, vous signifiant son désaccord profond avec vos instructions...

Le code suivant montre comment on peut allouer, puis libérer, un tableau de 10 valeurs flottantes :

```

1 float *t;
2
3 t = malloc(10 * sizeof(float));
4 if( t == NULL )
5 {
6     printf("Erreur d'allocation memoire.\n");
7     return EXIT_FAILURE;
```

FIGURE 16 – Structure d'un tableau 2d dans la mémoire

```
8 }
9
10 //Libération de la zone mémoire pointée par t
11 free(t);
12 return EXIT_SUCCESS;
```

Il est important, pour éviter qu'un programme ne consomme trop de mémoire pendant son exécution, de libérer les zones allouées dynamiquement dès qu'elles ne sont plus nécessaires. Dans certains langages de programmation, comme Java, la suppression mémoire des zones allouées dynamiquement est réalisée automatiquement, à l'aide d'une routine qui détecte quand une zone mémoire ne sera plus utile (le Garbage Collector).

### 3.7.2 Les fonctions `calloc`, `realloc` et `memcpy`

La fonction `calloc` prend deux paramètres : un nombre de cases  $n$  à allouer, ainsi que la taille mémoire  $t$  (en octet) de chaque case à allouer, et retourne un pointeur sur une zone mémoire de  $n \times t$  octets allouée par le système, dont chacune des  $n$  cases a été allouée à 0.

La fonction `realloc` permet de prendre en paramètre une zone mémoire allouée dynamiquement et de l agrandir ou de la rétrécir.

La fonction `memcpy` permet de recopier l'intégralité d'une zone mémoire dans une autre.

## 3.8 Tableaux dynamiques 2d (et nd par extension)

### 3.8.1 Structure d'un tableau dynamique 2d

Un tableau dynamique 2d est un assemblage de tableaux dynamiques 1d. Un tel tableau est constitué de multiples tableaux dynamiques 1d, représentant les colonnes ou les lignes du tableau 2d. Un autre tableau 1d servira à conserver les adresses mémoire de ces multiples tableaux. Enfin, un pointeur pointera sur ce dernier tableau. La figure 16 schématisé la structure d'un tableau 2d.

### 3.8.2 Déclaration d'un tableau dynamique 2d

Un tableau dynamique 1d peut être représenté, comme on l'a vu précédemment, par un pointeur sur une zone mémoire. Un tableau dynamique 2d possède une "couche" de pointeurs supplémentaires : un

tableau regroupe toutes les adresses des tableaux 1d, et le tableau 2d est donc un pointeur vers ce tableau. Un tableau dynamique 2d sera donc représenté par un pointeur sur pointeur.

Si l'on souhaite faire un tableau 2d d'entiers 8 bits, on déclarera un pointeur :

```
1 uint8_t **tab;
```

### 3.8.3 Allocation d'un tableau dynamique 2d

Pour allouer un tableau dynamique 2d, il faut tout d'abord allouer une zone mémoire au pointeur **tab**. Cette zone mémoire sera un tableau 1d qui contiendra lui-même des tableaux 1d, donc des pointeurs sur le type de données à stocker. En reprenant l'exemple précédent, on fera :

```
1 //La premiere allocation
2 tab = (uint8_t**) malloc(largeur * sizeof(uint8_t*));
3
4 if(tab == NULL)
5 {
6     printf("Erreur d'allocation memoire (1).\n");
7     return EXIT_FAILURE;
8 }
```

La seconde allocation consiste à allouer tous les tableaux 1d qui seront raccrochés au tableau précédemment alloué. En reprenant l'exemple précédent :

```
1 //La seconde allocation
2 for(i=0; i<largeur; i++)
3 {
4     tab[i] = (uint8_t*) malloc(hauteur * sizeof(uint8_t));
5
6     if(tab[i] == NULL)
7     {
8         printf("Erreur d'allocation memoire (2).\n");
9         return EXIT_FAILURE;
10    }
11 }
```

Ce système d'allocation permet d'arriver à une situation similaire à celle décrite sur la figure 16.

### 3.8.4 Ecriture d'une donnée dans un tableau 2d

Une fois l'allocation précédent correctement réalisée, on peut utiliser le pointeur **tab** comme un tableau 2d classique. On pourra, pour placer une valeur à la case de coordonnées (*i*; *j*), faire :

```
1 tab[i][j] = 3;
```

Ici, il faudra s'assurer que **i** soit compris entre 0 et **largeur - 1**, et que **j** soit compris entre 0 et **hauteur - 1**.

A noter qu'avec ce système, il est possible de faire des tableaux dont chaque colonne (ou ligne, selon le point de vue) n'a pas la même taille. Ce peut être intéressant pour stocker, par exemple, des matrices symétriques.

### 3.8.5 Libération mémoire d'un tableau 2d

Pour libérer un tableau dynamique 2d de la mémoire, on libérera les éléments dans l'ordre inverse de leur allocation. En reprenant l'exemple précédent, on fera :

```
1 for(i=0; i<largeur; i++)
2 {
3     free(tab[i]);
4 }
5
6 free(tab);
7 }
```

FIGURE 17 – Etat de la mémoire après l'allocation d'une variable de type **generique**

### 3.8.6 Les tableaux nd

La méthode expliquée précédemment peut facilement se généraliser à des tableaux de n dimensions. Il faut alors déclarer le tableau comme un pointeur sur pointeur sur ... sur pointeur, et utiliser un première boucle pour faire la première allocation, puis deux boucles imbriquées pour faire la seconde allocation, puis ... puis (n-1) boucles imbriquées pour la dernière allocation.

## 3.9 Allocation de structures et de tableaux de structures

### 3.9.1 Allocation d'une structure

Si l'on possède une structure de donnée déjà définie, il est possible d'allouer une telle structure en déclarant une variable du type de la structure. Par exemple, si l'on possède une structure **generique**, on peut construire une variable de type **generique** en faisant :

```
1 generique a;
```

On peut aussi construire une structure **generique** en faisant une allocation dynamique sur un pointeur :

```
1 generique *b;
2
3 b = malloc(1 * sizeof(generique))
4 if(b==NULL)
5 {
6     ...
7 }
```

Dans le premier cas, si la structure **generique** possède un champ "taille", on pourra y placer la valeur 2 en faisant :

```
1 a.taille = 2;
```

Dans le second cas, la variable **b** n'est pas une variable de type **generique**, mais un pointeur sur **generique**. On devra alors, pour l'accès aux champs, remplacer le signe point par une flèche :

```
1 b->taille = 2;
```

La figure 17 montre l'état de la mémoire après l'allocation d'une variable de type **generique**.

### 3.9.2 Allocation d'un tableau de structures

Pour allouer un tableau de variables de type **generique**, on fera (si l'on souhaite n cases) :

```
1 generique *b;
2
3 b = malloc(n * sizeof(generique))
4 if(b==NULL)
5 {
6     ...
7 }
```

FIGURE 18 – Etat de la mémoire après l’allocation d’un tableau de `generique`

```
7 }
```

Dans ce cas, pour placer 2 dans le champ "taille" de la septième case, on fera :

```
1 b[6].taille = 2;
```

La variable `b[6]` n'est pas un pointeur sur un `generique`, mais directement un `generique`; voilà pourquoi on utilise un point et non une flèche pour accéder à ses champs.

La figure 18 montre l'état de la mémoire après l'allocation d'un tableau de `generique`.

## 4 Comprendre ce qu'attend une fonction selon ses paramètres

Après la lecture de ce chapitre, il est possible de comprendre ce qu'attend une fonction, et ce que fera cette fonction, selon les paramètres qu'elle prend en entrée :

- Si une fonction prend en entrée une variable, c'est qu'elle a besoin de la valeur contenue dans cette variable pour effectuer un calcul.
- Si une fonction prend en entrée un pointeur sur une variable, c'est soit :
  - qu'elle a besoin de la variable passée par référence. Elle va donc probablement modifier sa valeur et vous pourrez récupérer ces modifications.
  - qu'elle besoin d'un tableau 1d de la variable. Dans ce cas, vous devriez avoir, parmi les autres paramètres de la fonction, la taille du tableau.
- Si une fonction prend en entrée un pointeur sur pointeur sur une variable, c'est soit :
  - qu'elle a besoin d'un pointeur passé par référence, car elle a besoin de modifier la valeur du pointeur. En général, modifier la valeur d'un pointeur signifie que l'on va allouer quelque chose sur le pointeur (ici, la fonction pourrait donc allouer un tableau 1d du type de la variable pointée, et l'adresse du tableau serait ainsi récupérée dans la fonction principale).
  - qu'elle besoin d'un tableau 2d de la variable. Dans ce cas, vous devriez avoir, parmi les autres paramètres de la fonction, les tailles du tableau.

### Questions

Que pensez-vous du code suivant ?

```
1 void f(uint32_t **a);  
2  
3 int main()  
4 {  
5     uint32_t v = 3;  
6  
7     f(&(&v));  
8  
9     return EXIT_SUCCESS;  
10 }
```

## 5 Avec de grands pouvoirs viennent de grandes responsabilités

L'utilisation des pointeurs permet d'avoir un contrôle assez fin de la gestion mémoire d'un programme, grâce aux allocations et libérations mémoire. Cependant, il faut rester prudent et concentré lors de l'écriture de son programme, afin de s'assurer qu'à aucun moment, on ne lit des zones non allouées de la mémoire.

Lorsque l'on lit ou écrit dans une zone non allouée d'un programme, l'erreur de segmentation qui pourrait survenir (si elle survient) n'apparaît pas forcément dès que le programme écrit ou lit des données dans des endroits interdits. Débugger son programme avec des printf risque d'être difficile, car l'erreur ne se manifestera pas sur le terminal au moment où le programme fait la faute.

Il est fortement encouragé d'utiliser des outils spécialisés dans ce type de débogage pour trouver l'erreur d'un programme en cas d'erreur de segmentation. Le programme **valgrind** sous Linux est un excellent programme pour cela. Pour l'utiliser, on compile tout d'abord son programme avec les options de débogage activées :

```
1 gcc -g monprog.c -o monprog
```

On peut alors exécuter son programme avec valgrind :

```
1 valgrind monprog
```

Le programme s'exécutera alors (très lentement) et toute erreur de segmentation sera immédiatement signalée, avec le numéro de ligne où se passe le problème.

L'outil valgrind permet aussi de vérifier si toutes les zones mémoire allouées ont été libérées à la fin d'un programme (voir le TP2), et permet aussi de faire du profiling, c'est à dire de voir dans quelles zones du code un programme "perd" le plus de temps.

## Programmation avancée et Structures de données Fonctions sur les listes chaînées simples

### Table des matières

<b>1</b>	<b>Principe d'une liste chaînée</b>	<b>1</b>
<b>2</b>	<b>Déclaration des structures associées</b>	<b>2</b>
2.1	Déclaration du maillon . . . . .	2
2.2	Déclaration de la liste . . . . .	2
<b>3</b>	<b>Allocation des structures associées</b>	<b>2</b>
3.1	Allouer un maillon . . . . .	2
3.2	Allouer une liste . . . . .	3
<b>4</b>	<b>Ajout d'éléments dans la liste</b>	<b>3</b>
4.1	Ajouter en tête de liste . . . . .	3
4.2	Exemple d'ajout en tête de liste . . . . .	3
4.3	Ajouter en queue de liste . . . . .	3
4.4	Exemple d'ajout en queue de liste . . . . .	4
<b>5</b>	<b>Suppression d'éléments dans la liste</b>	<b>5</b>
5.1	Supprimer la tête de liste . . . . .	5
5.2	Exemple de suppression de tête de liste . . . . .	5
5.3	Supprimer la queue de liste . . . . .	6
5.4	Exemple de suppression de queue de liste . . . . .	6
5.5	Vider une liste . . . . .	7
<b>6</b>	<b>Fonction de test sur la liste</b>	<b>7</b>
<b>7</b>	<b>Parcours de la liste</b>	<b>8</b>
7.1	Parcours de la liste entière . . . . .	8
7.2	Trouver une donnée à une position particulière . . . . .	9
7.3	Exemple de recherche d'un maillon à une position particulière . . . . .	9
7.4	Ajouter une donnée à une position particulière . . . . .	9
7.5	Exemple d'ajout d'un maillon à une position particulière . . . . .	10
7.6	Supprimer une donnée à une position particulière . . . . .	11
7.7	Exemple de suppression d'un maillon à une position particulière . . . . .	12
<b>8</b>	<b>Comparatif tableau/liste</b>	<b>13</b>
<b>9</b>	<b>Conclusion</b>	<b>13</b>

### 1 Principe d'une liste chaînée

Une liste chaînées est une structure de données permettant de stocker des données dont on ne connaît, à priori, la quantité. Cette structure a pour avantage de pouvoir grandir et rétrécir à volonté, occupant ainsi, dans l'ordinateur, l'espace mémoire nécessaire et suffisant à chaque instant.

Le défaut de cette structure est qu'il peut être long de retrouver une donnée, même si on connaît sa position dans la liste, contrairement aux tableaux où l'accès à une donnée dont on connaît la position se fait en temps constant.

Pour se représenter cette structure, on peut s'imaginer une longue chaîne constituée de maillons. Chaque maillon est relié à deux autres maillons, un avant lui et un après lui, sauf les maillons aux extrémités. Si l'on souhaite agrandir la chaîne et y rajouter un maillon, on peut soit ajouter le maillon au début ou à la fin de la chaîne, soit détacher deux maillons de la chaîne, et y placer notre maillon.

La figure 1 représente l'architecture générale d'une liste chaînée.

## 2 Déclaration des structures associées

### 2.1 Déclaration du maillon

Pour constituer une liste chaînée, il faut avant tout décrire comment y seront stockées les données. Pour cela, on crée une structure **maillon** qui contiendra la donnée **data** à stocker.

```
1 typedef struct _maillon
2 {
3     type_data data; //type_data aura été défini précédemment
4     struct _maillon *suivant;
5 } maillon;
```

La figure 1 montre l'aspect des maillons d'une liste chaînée.

### 2.2 Déclaration de la liste

Une liste chaînée est un ensemble de maillons, "attachés" les uns aux autres à l'aide de pointeurs. Pour sauvegarder l'intégralité de la liste, on sauvegarde, dans une variable (ou une structure dédiée), son tout premier maillon. A partir de ce maillon, on pourra retrouver tous les autres.

```
1 typedef struct
2 {
3     uint64_t taille; //Pas obligatoire, mais vraiment pratique dans certains cas
4     maillon *tete;
5     maillon *queue; //Pas obligatoire non plus, mais pratique dans le cas des files
6 } liste;
```

La figure 1 montre où se situe la **liste** dans la structure de données.

## 3 Allocation des structures associées

L'allocation des structures de données associées à la liste chaînée se fait de façon classique, avec la fonction **malloc**.

### 3.1 Allouer un maillon

```
1 maillon *new_maillon(type_data d)
2 {
```

FIGURE 1 – Architecture mémoire d'une liste chaînée

```

3 maillon *m = malloc(sizeof(maillon));
4 if(m==NULL)
5 {
6     assert(0);
7 }
8 m->data = d;
9 return m;
10 }
```

## 3.2 Allouer une liste

```

1 liste *new_liste()
2 {
3     liste *r = malloc(sizeof(liste));
4     if(r==NULL)
5     {
6         assert(0);
7     }
8     r->taille=0;
9     r->tete = NULL;
10    r->queue = NULL;
11    return r;
12 }
```

# 4 Ajout d'éléments dans la liste

## 4.1 Ajouter en tête de liste

```

1 void add_tete(liste *l, type_data d)
2 {
3     maillon *m = new_maillon(d);
4     m->suivant = l->tete;
5     l->tete = m;
6
7     if( l->taille == 0 )
8     {
9         l->queue = m;
10    }
11
12    l->taille += 1;
13 }
```

## 4.2 Exemple d'ajout en tête de liste

```

1 liste *l = new_liste();
2
3 add_tete(l, d1);
4 add_tete(l, d2);
5 add_tete(l, d3);
```

La figure 2 montre l'état de la mémoire après chaque ligne de code de l'exemple précédent.

## 4.3 Ajouter en queue de liste

FIGURE 2 – Ajout d’éléments en tête d’une liste chaînée

```
1 void add_queue(liste *l, type_data d)
2 {
3     maillon *m = new_maillon(d);
4     m->suivant = NULL;
5     if( l->taille > 0)
6     {
7         l->queue->suivant = m;
8     }
9     else
10    {
11        l->tete = m;
12    }
13    l->queue = m;
14
15    l->taille += 1;
16 }
```

Notez bien que, dans la fonction d’ajout en queue, on ne peut rajouter un maillon à la suite de la queue de la liste que si cette dernière a au moins un maillon (et donc une queue). Si ce n’est pas le cas, il faut rajouter le maillon directement sur la queue.

#### 4.4 Exemple d’ajout en queue de liste

```
1 liste *l = new_liste();
2
3 add_queue(l, d1);
4 add_queue(l, d2);
5 add_queue(l, d3);
```

La figure 3 montre l’état de la mémoire après chaque ligne de code de l’exemple précédent.

FIGURE 3 – Ajout d’éléments en queue d’une liste chaînée

## 5 Suppression d’éléments dans la liste

### 5.1 Supprimer la tête de liste

```
1 type_data rem_tete(liste *l)
2 {
3     maillon *t = l->tete;
4     type_data r = t->data;
5
6     l->tete = l->tete->suivant;
7     free(t);
8     l->taille -= 1;
9
10    if( l->taille == 0 )
11    {
12        l->queue = NULL;
13    }
14    return r;
15 }
```

Notez bien ici qu’il faut vérifier, avant d’appeler la fonction, que la liste ne soit pas vide. Notez aussi que la fonction envoie la donnée contenue dans le maillon supprimé. Le pointeur **t** sert à ne pas "oublier" le maillon supprimé afin de le libérer de la mémoire.

### 5.2 Exemple de suppression de tête de liste

```
1 liste *l = new_liste();
2
3 add_queue(l, d1);
4 add_queue(l, d2);
5 add_queue(l, d3);
6
7 d4 = rem_tete(l);
```

La figure 4 montre l'état de la mémoire avant et après l'appel de la fonction de suppression de la tête de la liste.

FIGURE 4 – Suppression de la tête d'une liste chaînée

### 5.3 Supprimer la queue de liste

```
1 type_data rem_queue(liste *l)
2 {
3     maillon *avant_dernier = NULL;
4     maillon *dernier = l->tete;
5     type_data d;
6
7     while(dernier != l->queue)
8     {
9         avant_dernier = dernier;
10        dernier = dernier -> suivant;
11    }
12
13    d = dernier->data;
14    l->queue = avant_dernier;
15    l->queue->suivant = NULL;
16    free(dernier);
17    l->taille -= 1;
18
19    if(l->taille == 0)
20    {
21        l -> tete = NULL;
22    }
23
24    return d;
25 }
```

Notez bien ici qu'il faut vérifier, avant d'appeler la fonction, que la liste ne soit pas vide. Notez aussi que la fonction envoie la donnée contenue dans le maillon supprimé. Les pointeurs **dernier** et **avant\_dernier** avancent en même temps dans la liste pour venir se positionner respectivement sur le dernier et l'avant dernier maillon.

### 5.4 Exemple de suppression de queue de liste

```
1 liste *l = new_liste();
2
3 add_queue(l, d1);
4 add_queue(l, d2);
5 add_queue(l, d3);
6
```

FIGURE 5 – Suppression de la queue d'une liste chaînée

```
7 d4 = rem_queue(l);
```

La figure 5 montre l'état de la mémoire pendant et après l'exécution de la boucle while de la fonction `rem_queue` dans le code précédent.

## 5.5 Vider une liste

Pour vider une liste complètement de la mémoire, on retire sa tête jusqu'à ce que la liste soit vide.

```
1 void vider(liste *l)
2 {
3     while( ! est_vide(l) )
4     {
5         rem_tete(l);
6     }
7
8     free(l);
9 }
```

## 6 Fonction de test sur la liste

Si on a, dans notre structure de liste, un champ `taille` que l'on a gardé à jour, alors on peut faire :

```
1 _Bool est_vide( liste *l )
2 {
3     return l->taille == 0;
4 }
```

Sinon, on peut faire :

```

1 _Bool est_vide( liste *l )
2 {
3     return l->tete == NULL;
4 }
```

## 7 Parcours de la liste

### 7.1 Parcours de la liste entière

Pour parcourir entièrement une liste, on utilise un maillon qui va avancer dans la liste jusqu'à en atteindre la fin.

```

1 void parcours_liste(liste *l)
2 {
3     maillon *p = l->tete;
4     while( p != NULL )
5     {
6         //Faire quelquechose avec p, comme par exemple afficher le contenu de p
7         p = p->suivant;
8     }
9 }
```

La figure 6 montre l'état de la mémoire pendant l'exécution de la fonction précédente sur une liste de trois éléments.

FIGURE 6 – Parcours d'une liste chaînée

## 7.2 Trouver une donnée à une position particulière

Pour trouver une donnée à une position particulière, on parcourt la liste et on s'arrête lorsque l'on aura passé un certain nombre de maillons. Dans le code suivant, la tête de la liste est considérée être à la position 0.

```
1 maillon *trouver_maillon(liste *l, uint64_t position)
2 {
3     maillon *r = l->tete;
4     uint64_t i;
5
6     if( position >= l->taille)
7     {
8         //Erreur, on demande une position en dehors de la liste
9         return NULL;
10    }
11
12    for (i=0; i<position; i++)
13    {
14        r = r->suivant;
15    }
16    return r;
17 }
18
19
20 type_data trouver_donnee(liste *l, uint64_t position)
21 {
22     maillon *r = trouver_maillon(l, position);
23
24     if( r == NULL )
25     {
26         //Erreur, on a demandé une position en dehors de la liste
27         assert(0);
28     }
29
30     return r->data;
31 }
```

## 7.3 Exemple de recherche d'un maillon à une position particulière

```
1 liste *l = new_liste();
2 maillon *m;
3
4 add_queue(l, d1);
5 add_queue(l, d2);
6 add_queue(l, d3);
7 add_queue(l, d4);
8
9 m=trouver_maillon(l, 2);
```

La figure 7 montre l'état de la mémoire pendant l'exécution des différentes étapes de la fonction **trouver\_maillon** dans l'exemple précédent.

## 7.4 Ajouter une donnée à une position particulière

Pour ajouter une donnée à une position particulière, on recherche le maillon situé juste avant cette position. On intercale alors notre donnée entre le maillon sélectionné et son suivant.

```
1 void add_a_la_position(liste *l, uint64_t position, type_data d)
2 {
3     maillon *m, *r;
```

FIGURE 7 – Recherche d'un maillon à une position particulière dans une liste chaînée

```
4 if(position > l->taille)
5 {
6     //Erreur, on demande de ajouter dans une position en dehors de la liste
7     assert(0);
8 }
9 else if( position == 0)
10 {
11     add_tete(l, d);
12 }
13 else if( position == l->taille )
14 {
15     add_queue(l, d);
16 }
17 else
18 {
19     r = trouver_maillon(l, position-1);
20     m = new_maillon(d);
21
22     m->suivant = r->suivant;
23     r->suivant = m;
24     l->taille += 1;
25 }
26 }
```

## 7.5 Exemple d'ajout d'un maillon à une position particulière

```
1 liste *l = new_liste();
2 maillon *m;
3
4 add_tete(l, d1);
5 add_tete(l, d2);
```

```

6 add_tete(l, d3);
7 add_tete(l, d4);
8
9 m=add_a_la_position(l, 3);

```

La figure 8 montre l'état de la mémoire pendant l'exécution des différentes étapes de la fonction **add\_a\_la\_position** dans l'exemple précédent.

FIGURE 8 – Ajout d'un maillon à une position particulière dans une liste chaînée

## 7.6 Supprimer une donnée à une position particulière

Pour supprimer une donnée à une position particulière, on recherche le maillon situé juste avant cette position. On supprime alors le maillon voisin du maillon trouvé, qu'on lie avec le voisin du maillon supprimé.

```

1 type_data rem_a_la_position(liste *l, uint64_t position)
2 {
3     maillon *m, *r;
4     type_data d;
5
6     if(position >= l->taille)
7     {
8         //Erreur, on demande a supprimer dans une position en dehors de la liste
9         assert(0);
10    }
11    else if( position == 0)
12    {
13        return rem_tete( l );
14    }
15    else if( position == l->taille - 1)
16    {
17        return rem_queue( l );
18    }
19    else
20    {

```

```

21     r=trouver_maillon(l, position-1);
22     m = r->suivant;
23     d = m->data;
24     r->suivant = m->suivant;
25
26     free(m);
27     l->taille -= 1;
28     return d;
29 }
30 }
```

## 7.7 Exemple de suppression d'un maillon à une position particulière

```

1 liste *l = new_liste();
2
3 add_tete(l, d1);
4 add_tete(l, d2);
5 add_tete(l, d3);
6 add_tete(l, d4);
7
8 d=rem_a_la_position(l, 2);
```

La figure 9 montre l'état de la mémoire pendant l'exécution des différentes étapes de la fonction **rem\_a\_la\_position** dans l'exemple précédent.

FIGURE 9 – Suppression d'un maillon à une position particulière dans une liste chaînée

## 8 Comparatif tableau/liste

Le tableau suivant récapitule les complexités (complexité pire cas) de différentes opérations sur une liste et sur un tableau :

	Liste	Tableau
Ajout d'un élément au début		
Ajout d'un élément en fin		
Ajout d'un élément à une position quelconque		
Lire une valeur au début		
Lire une valeur à la fin		
Lire une valeur à une position quelconque		
Supprimer un élément au début		
Supprimer un élément à la fin		
Supprimer un élément à une position quelconque		

## 9 Conclusion

Les listes chaînées sont des structures souples dont la taille mémoire varie en fonction de la quantité de données stockées, contrairement aux tableaux où la taille est fixe (bien que l'on puisse les agrandir avec la fonction `realloc`).

L'ajout ou la suppression d'éléments dans une liste se fait en temps constant si les éléments doivent être placés au début ou à la fin de la liste ; si la position est quelconque, le temps d'exécution de l'ajout dépend de la position à laquelle on souhaite placer l'élément.

L'accès à un élément particulier n'est pas aussi efficace qu'avec le tableau, où l'accès de fait en temps constant. Dans une liste, le temps d'accès à un élément dépend de sa position dans la liste.

Les listes sont donc des structures efficaces lorsque l'on souhaite stocker des données dont on ne connaît, à priori, pas la quantité. Ce sont des structures moins efficaces que les tableaux s'il y a besoin d'accéder à des données dans un ordre quelconque.

# Programmation avancée et Structures de données

## Listes spéciales : files, piles et listes doublement chaînées

### Table des matières

<b>1 Les files</b>	<b>1</b>
1.1 Les structures de données associées . . . . .	2
1.1.1 Déclaration du maillon . . . . .	2
1.1.2 Déclaration de la file . . . . .	2
1.2 Allocation des structures associées . . . . .	2
1.2.1 Allouer un maillon . . . . .	2
1.2.2 Allouer une liste . . . . .	2
1.3 Ajout d'éléments dans la file . . . . .	2
1.4 Suppression d'éléments dans la file . . . . .	3
1.5 A savoir sur les files . . . . .	3
<b>2 Les piles</b>	<b>3</b>
2.1 Les structures de données associées . . . . .	3
2.1.1 Déclaration du maillon . . . . .	3
2.1.2 Déclaration de la pile . . . . .	4
2.2 Allocation des structures associées . . . . .	4
2.2.1 Allouer un maillon . . . . .	4
2.2.2 Allouer une pile . . . . .	4
2.3 Ajout d'éléments dans la pile . . . . .	4
2.4 Suppression d'éléments dans la pile . . . . .	4
<b>3 Les listes doublement chaînées</b>	<b>5</b>
3.1 Déclaration des structures associées . . . . .	5
3.1.1 Déclaration du maillon . . . . .	5
3.1.2 Déclaration de la liste . . . . .	5
3.2 Allocation des structures associées . . . . .	6
3.2.1 Allouer un maillon . . . . .	6
3.2.2 Allouer une liste doublement chaînée . . . . .	6
3.3 Ajout d'éléments dans la liste . . . . .	6
3.3.1 Ajouter en tête de liste . . . . .	6
3.3.2 Ajouter en queue de liste . . . . .	7
3.3.3 Ajouter une donnée à une position particulière . . . . .	7
3.3.4 Exemple d'ajout dans une liste doublement chaînée . . . . .	8
3.4 Suppression d'éléments dans la liste . . . . .	8
3.4.1 Supprimer la tête de liste . . . . .	8
3.4.2 Supprimer la queue de liste . . . . .	9
3.4.3 Supprimer une donnée à une position particulière . . . . .	9
3.4.4 Exemple de suppression dans une liste doublement chaînée . . . . .	10
3.5 Stratégie d'accélération de l'ajout et de la suppression . . . . .	10

### 1 Les files

Les files sont des listes où l'insertion se fait en queue, et la suppression/lecture en tête. On peut les comparer à une file d'attente : les nouvelles personnes arrivent à la fin de la file d'attente, et celles qui sortent de la file sont celles en tête de file.

## 1.1 Les structures de données associées

### 1.1.1 Déclaration du maillon

Une structure similaire à celle d'un maillon de liste.

```
1 typedef struct _maillon
2 {
3     type_data data; //type_data aura été défini précédemment
4     struct _maillon *suivant;
5 } maillon;
```

### 1.1.2 Déclaration de la file

Une structure similaire à celle d'une liste, si ce n'est qu'il est très recommandé ici, pour l'ajout en queue, d'avoir un pointeur sur la queue de file (sauf si l'on souhaite perdre du temps à parcourir la file à chaque ajout d'un nouvel élément).

```
1 typedef struct
2 {
3     uint64_t taille; //Pas obligatoire, mais vraiment pratique dans certains cas
4     maillon *tete;
5     maillon *queue; //Obligatoire ici
6 } file;
```

## 1.2 Allocation des structures associées

### 1.2.1 Allouer un maillon

```
1 maillon *new_maillon(type_data d)
2 {
3     maillon *m = malloc(sizeof(maillon));
4     if(m==NULL)
5     {
6         assert(0);
7     }
8     m->data = d;
9     return m;
10 }
```

### 1.2.2 Allouer une liste

```
1 file *new_file()
2 {
3     file *r = malloc(sizeof(file));
4     if(r==NULL)
5     {
6         assert(0);
7     }
8     r->taille=0;
9     r->tete = NULL;
10    r->queue = NULL;
11    return r;
12 }
```

## 1.3 Ajout d'éléments dans la file

L'ajout d'un élément dans une file ou une pile s'appelle le **push**. Pour une file, cet ajout se fait en queue .

```

1 void push(file *l, type_data d)
2 {
3     add_queue(l, d); //On ne peut pas vraiment appeler cette fonction,
4                 //a cause des types de parametres, mais c'est le meme code
5 }
```

## 1.4 Suppression d'éléments dans la file

La suppression d'un élément dans une file ou une pile s'appelle le **pop**. Pour une file, cet ajout se fait en tête. A savoir que, normalement, la suppression est le seul moyen de lire des éléments dans une file ou une pile.

```

1 type_data pop(file *l)
2 {
3     return rem_tete(l); //On ne peut pas vraiment appeler cette fonction,
4                 //a cause des types de parametres, mais c'est le meme code
5 }
```

## 1.5 A savoir sur les files

L'utilisation d'une file se limite normalement à ces fonctions. Il n'est normalement pas autorisé, dans une file, d'insérer des éléments en tête ou en milieu de file. Rien n'empêche de le faire, mais dans ce cas, d'un point de vue "philomatique" (philosophie + informatique), ce n'est plus une file mais une liste que l'on a.

### Questions

A votre avis, est-il possible de faire une file en inversant les pointeurs : on réalise l'ajout en tête, et la suppression en queue ? Si oui, est-ce recommandé ?

## 2 Les piles

Les piles sont des listes où l'insertion et la suppression/lecture se font en tête. On peut les comparer à une tas de documents : le dernier document rajouté est au-dessus du tas, et lorsqu'on prend un document, on prend celui en haut du tas (donc le dernier ajouté).

### 2.1 Les structures de données associées

#### 2.1.1 Déclaration du maillon

Une structure similaire à celle d'un maillon de liste.

```

1 typedef struct _maillon
2 {
3     type_data data; //type_data aura ete defini precedemment
4     struct _maillon *suivant;
5 } maillon;
```

### 2.1.2 Déclaration de la pile

Une structure similaire à celle d'une liste, si ce n'est que l'on ne gère pas le pointeur de queue ici car il n'est pas utile (rien n'empêche d'en avoir un, mais il ne servira pas à la structure si on la considère réellement comme une pile).

```
1 typedef struct
2 {
3     uint64_t taille; //Pas obligatoire, mais vraiment pratique dans certains cas
4     maillon *tete;
5 } pile;
```

## 2.2 Allocation des structures associées

### 2.2.1 Allouer un maillon

```
1 maillon *new_maillon(type_data d)
2 {
3     maillon *m = malloc(sizeof(maillon));
4     if(m==NULL)
5     {
6         assert(0);
7     }
8     m->data = d;
9     return m;
10 }
```

### 2.2.2 Allouer une pile

```
1 pile *new_pile()
2 {
3     pile *r = malloc(sizeof(pile));
4     if(r==NULL)
5     {
6         assert(0);
7     }
8     r->taille=0;
9     r->tete = NULL;
10    return r;
11 }
```

## 2.3 Ajout d'éléments dans la pile

On ajoute forcément en tête de pile. Le code est différent de celui des listes, étant donné que l'on n'a pas besoin de gérer le pointeur de queue.

```
1 void push(pile *l, type_data d)
2 {
3     maillon *m = new_maillon(d);
4     m->suivant = l->tete;
5     l->tete = m;
6     l->taille += 1;
7 }
```

## 2.4 Suppression d'éléments dans la pile

On retire forcément les éléments en tête de pile. Le code est différent de celui des listes, étant donné que l'on n'a pas besoin de gérer le pointeur de queue.

```

1 type_data pop(pile *l)
2 {
3     maillon *t = l->tete;
4     type_data r = t->data;
5
6     l->tete = l->tete->suivant;
7     free(t);
8     l->taille -= 1;
9
10    return r;
11 }
```

### 3 Les listes doublement chaînées

Une liste doublement chaînée est une liste dans laquelle chaque maillon est relié au maillon le suivant ainsi qu'au maillon le précédent. Cela permet de faciliter certaines opérations. La figure 1 montre l'architecture générale d'une liste doublement chaînée.

FIGURE 1 – Architecture générale d'une liste doublement chaînée

#### 3.1 Déclaration des structures associées

##### 3.1.1 Déclaration du maillon

```

1 typedef struct _maillon
2 {
3     type_data data; //type_data aura été défini précédemment
4     struct _maillon *suivant;
5     struct _maillon *precedent;
6 } maillon;
```

##### 3.1.2 Déclaration de la liste

Ici, la liste choisie possède un pointeur de queue et un pointeur de tête. Il n'est cependant pas nécessaire, pour gérer la liste doublement chaînée, d'avoir ces deux pointeurs : un seul des deux suffit

car on peut retrouver, à partir de la seule tête ou la seule queue, tous les autres maillons de la liste. La présence de ces deux pointeurs permet de rajouter et supprimer des éléments en tête ou en queue de liste de façon très rapide.

```
1 typedef struct
2 {
3     uint64_t taille; //Pas obligatoire, mais vraiment pratique dans certains cas
4     maillon *tete;
5     maillon *queue;
6 } liste_double;
```

## 3.2 Allocation des structures associées

### 3.2.1 Allouer un maillon

```
1 maillon *new_maillon(type_data d)
2 {
3     maillon *m = malloc(sizeof(maillon));
4     if(m==NULL)
5     {
6         assert(0);
7     }
8     m->data = d;
9     return m;
10 }
```

### 3.2.2 Allouer une liste doublement chaînée

```
1 liste_double *new_liste_double()
2 {
3     liste_double *r = malloc(sizeof(liste_double));
4     if(r==NULL)
5     {
6         assert(0);
7     }
8     r->taille=0;
9     r->tete = NULL;
10    r->queue = NULL;
11    return r;
12 }
```

## 3.3 Ajout d'éléments dans la liste

### 3.3.1 Ajouter en tête de liste

```
1 void add_tete(liste_double *l, type_data d)
2 {
3     maillon *m = new_maillon(d);
4     m->suivant = l->tete;
5     m->precedent = NULL;
6
7     if( l->taille == 0 )
8     {
9         l->queue = m;
10    }
11    else
12    {
13        l->tete->precedent = m;
```

```

14 }
15 l->tete = m;
16 l->taille += 1;
17 }
```

### 3.3.2 Ajouter en queue de liste

```

1 void add_queue(liste_double *l, type_data d)
2 {
3     maillon *m = new_maillon(d);
4     m->suivant = NULL;
5     m->precedent = l->queue;
6     if( l->taille > 0)
7     {
8         l->queue->suivant = m;
9     }
10    else
11    {
12        l->tete = m;
13    }
14    l->queue = m;
15
16    l->taille += 1;
17 }
```

### 3.3.3 Ajouter une donnée à une position particulière

```

1 void add_position(liste_double *l, uint64_t position, type_data d)
2 {
3     maillon *m, *prec, *suiv;
4
5     if(position > l->taille)
6     {
7         //Erreur, on demande a ajouter dans une position en dehors de la liste
8         assert(0);
9     }
10    else if( position == 0)
11    {
12        add_tete(l, d);
13    }
14    else if( position == l->taille )
15    {
16        add_queue(l, d);
17    }
18    else
19    {
20        //On ne peut pas appeler directement cette fonction a cause des parametres,
21        //mais le code est le meme que pour la fonction donnee sur les listes.
22        prec=trouver_maillon(l, position-1);
23        suiv = prec->suivant;
24        m = new_maillon(d);
25
26        prec->suivant = m;
27        m->suivant = suiv;
28        suiv->precedent = m;
29        m->precedent = prec;
30
31        l->taille += 1;
32    }
33 }
```

```
32     }
33 }
```

### 3.3.4 Exemple d'ajout dans une liste doublement chaînée

```
1 liste_double *l = new_liste_double();
2
3 add_tete(l, d1);
4 add_tete(l, d2);
5 add_tete(l, d3);
6 add_tete(l, d4);
7
8 add_position(l, 2, d5);
```

La figure 2 montre l'état de la mémoire au début et à la fin de l'exécution de la fonction **add\_position**.

FIGURE 2 – Ajout d'éléments dans une liste doublement chaînée

## 3.4 Suppression d'éléments dans la liste

### 3.4.1 Supprimer la tête de liste

```
1 type_data rem_tete(liste_double *l)
2 {
3     maillon *t = l->tete;
4     type_data r = t->data;
5
6     l->tete = l->tete->suivant;
7     free(t);
8     l->taille -= 1;
9
10    if( l->taille == 0 )
11    {
12        l->queue = NULL;
```

```

13 }
14 else
15 {
16     l->tete->precedent = NULL;
17 }
18 return r;
19 }
```

### 3.4.2 Supprimer la queue de liste

```

1 type_data rem_queue(liste_double *l)
2 {
3     maillon *t = l->queue;
4     type_data r = t->data;
5
6     l->queue = l->queue->precedent;
7     free(t);
8     l->taille -= 1;
9
10    if( l->taille == 0 )
11    {
12        l->tete = NULL;
13    }
14    else
15    {
16        l->queue->suivant = NULL;
17    }
18    return r;
19 }
```

### 3.4.3 Supprimer une donnée à une position particulière

```

1 type_data rem_position(liste_double *l, uint64_t position)
2 {
3     maillon *m, *prec, *suiv;
4     type_data d;
5
6     if(position >= l->taille)
7     {
8         //Erreur, on demande a supprimer dans une position en dehors de la liste
9         assert(0);
10    }
11    else if( position == 0)
12    {
13        return rem_tete( l );
14    }
15    else if( position == l->taille - 1)
16    {
17        return rem_queue( l );
18    }
19    else
20    {
21        //On ne peut pas appeler directement cette fonction a cause des parametres,
22        //mais le code est le meme que pour la fonction donnee sur les listes.
23        r=trouver_maillon(l, position);
24        prec = r->precedent;
25        suiv = r->suivant;
26        d = r->data;
```

```

27     prec->suivant = suiv;
28     suiv->precedent = prec;
29
30     free(r);
31     l->taille -= 1;
32     return d;
33 }
34 }
35 }
```

### 3.4.4 Exemple de suppression dans une liste doublement chaînée

```

1 liste_double *l = new_liste_double();
2
3 add_tete(l, d1);
4 add_tete(l, d2);
5 add_tete(l, d3);
6 add_tete(l, d4);
7
8 d = rem_position(l, 2, d5);
```

La figure 3 montre l'état de la mémoire au début et à la fin de l'exécution de la fonction **rem\_position**.

FIGURE 3 – Suppression d'éléments dans une liste doublement chaînée

## 3.5 Stratégie d'accélération de l'ajout et de la suppression

Dans une liste doublement chaînée, il est possible d'accélérer, dans certains cas, l'ajout ou la suppression d'un élément dont on connaît la position, à la condition que l'on connaisse la taille de la liste.

En effet, selon la taille de la liste et la position de l'élément, on pourra faire une recherche de l'élément soit en partant de la tête de la liste, soit en partant de la queue de la liste. Cette stratégie permet de gagner, dans certains cas, en temps d'exécution, mais ne permet pas d'améliorer la complexité des fonctions.

## Programmation avancée et Structures de données Fonctions sur les tables de hachage et arraylist

### Table des matières

<b>1 Table de hachage (chaînée)</b>	<b>1</b>
1.1 Les structures de données associées . . . . .	2
1.2 Initialisation de la structure . . . . .	2
1.3 Destruction de la structure . . . . .	3
1.4 La fonction de hachage . . . . .	3
1.5 L'ajout d'une donnée dans la table de hachage . . . . .	3
1.6 Exemple d'ajout d'une donnée dans une table de hachage . . . . .	4
1.7 La recherche d'une donnée . . . . .	5
<b>2 ArrayList</b>	<b>5</b>
2.1 Version sans libération de la mémoire . . . . .	5
2.1.1 Les structures de données associées . . . . .	5
2.1.1.1 Déclaration d'un maillon . . . . .	5
2.1.1.2 Déclaration de l'arraylist . . . . .	5
2.1.2 Modification des fonctions existantes sur les listes . . . . .	6
2.1.2.1 Fonctions d'ajout d'éléments dans les listes . . . . .	6
2.1.2.2 Fonctions de suppression d'éléments des listes . . . . .	7
2.1.3 Initialisation de l'arraylist . . . . .	7
2.1.3.1 Code de l'initialisation . . . . .	7
2.1.3.2 Exemple d'initialisation . . . . .	8
2.1.4 Ajout d'une donnée dans l'arraylist . . . . .	8
2.1.4.1 Code de l'ajout de donnée . . . . .	8
2.1.4.2 Exemple d'ajout de données . . . . .	9
2.1.5 Supprimer une donnée de l'arraylist . . . . .	9
2.1.5.1 Code de la suppression de donnée . . . . .	9
2.1.5.2 Exemple de suppression de données . . . . .	9
2.1.6 Libération mémoire de l'arraylist . . . . .	9
2.2 Version avec libération de la mémoire . . . . .	10
2.2.1 Les structures de données associées . . . . .	10
2.2.1.1 Déclaration des maillons . . . . .	10
2.2.1.2 Déclaration de l'arraylist . . . . .	10
2.2.2 Initialisation de l'arraylist . . . . .	10
2.2.3 Libération mémoire de l'arraylist . . . . .	11

### 1 Table de hachage (chaînée)

La table de hachage est une structure permettant d'allier la rapidité d'accès d'un tableau et la souplesse d'une liste chaînée. Le principal défaut d'une liste chaînée est le temps d'accès à ses éléments : lorsque la liste est d'une taille assez importante, la recherche d'un élément peut y prendre du temps (le temps de parcourir la liste dans son intégralité si nécessaire).

Une idée pour diminuer ce temps est de diviser la liste en de multiples sous listes, chacune associées à la case d'un tableau de liste. Avant de ranger un élément, on utilise une fonction (dite fonction de hachage) permettant de savoir quel élément doit aller dans quelle sous liste. Pour rechercher un élément, toujours grâce à la même fonction que précédemment, on sait dans quelle sous liste l'élément, s'il était présent dans notre structure, serait rangé.

La table de hachage permet donc de diminuer, en divisant une liste en de multiples sous listes, le temps de parcours de la structure et accélérer, par exemple, la recherche d'un élément. La figure 1 illustre la structure, en mémoire, d'une table de hachage.

FIGURE 1 – Architecture générale d'une table de hachage

## 1.1 Les structures de données associées

Un maillon classique de liste

```
1 typedef struct _maillon
2 {
3     type_data data; //type_data aura été défini précédemment
4     struct _maillon *suivant;
5 } maillon;
```

On déclare ensuite une structure de liste

```
1 typedef struct
2 {
3     uint32_t taille;
4     maillon *tete;
5     maillon *queue;
6 } liste;
```

Puis un tableau de listes : c'est ce tableau la table de hachage.

```
1 typedef struct
2 {
3     liste **tab;
4     uint32_t taille;
5 } tab_hachage;
```

## 1.2 Initialisation de la structure

On alloue l'intégralité du tableau de liste, et initialiser chacune des listes qui s'y trouvent.

```

1 tab_hachage* new_tab_hachage(uint32_t taille)
2 {
3     uint32_t i;
4
5     tab_hachage *th = malloc( sizeof(tab_hachage) );
6     assert(th != NULL);
7
8     th->tab = malloc(taille*sizeof(liste*));
9     assert(th->tab != NULL);
10
11    for( i=0; i<taille; i=i+1 )
12    {
13        th->tab[i] = new_liste();
14    }
15
16    th->taille = taille;
17    return th;
18 }
```

### 1.3 Destruction de la structure

On détruit la table de hachage en détruisant chacune des listes qui s'y trouvent, puis en détruisant le tableau de listes.

```

1 void vider_th(tab_hachage *th)
2 {
3     uint32_t i;
4
5     for( i=0; i<taille; i=i+1 )
6     {
7         vider(th->tab[i]);
8     }
9
10    free(th->tab);
11    free(th);
12 }
```

### 1.4 La fonction de hachage

La fonction de hachage est le coeur de la structure de table de hachage. C'est une fonction **déterministe** qui calcule dans quelle liste du tableau il faudra ajouter une donnée précise :

```

1 uint32_t hachage( tab_hachage *th, type_data d )
2 {
3     //On réalise un calcul déterministe qui, à partir d'une donnée d
4     //calcule un nombre entier r
5     //Aucun appel à une fonction aléatoire ne doit être fait ici
6     return r % th->taille;
7     //La fonction de hachage doit faire en sorte de répartir uniformément
8     //les différentes données dans les différentes listes de la table de hachage
9 }
```

Il existe de nombreux algorithmes de hachage, avec de très bonnes propriétés de répartition des éléments. SHA1 et MD5 sont des fonctions souvent utilisées. La seconde est disponible dans la librairie openssl.

### 1.5 L'ajout d'une donnée dans la table de hachage

On ajoute la donnée dans la liste indiquée par la fonction de hachage :

```

1 void add_tete_th( tab_hachage *th, type_data d )
2 {
3     uint32_t p = hachage( th, d );
4     add_tete( th->tab[p], d );
5 }
```

## 1.6 Exemple d'ajout d'une donnée dans une table de hachage

Voici un exemple d'ajout d'une donnée dans une table de hachage. La figure 2 illustre la structure de la mémoire après certaines étapes du code suivant :

```

1 type_data d1, d2, d3, d4;
2 tab_hachage *th = new_tab_hachage(3);
3
4 add_tete_th(th, d1);
5 add_tete_th(th, d2);
6 add_tete_th(th, d3);
7 add_tete_th(th, d4);
```

FIGURE 2 – Architecture mémoire de la table de hachage pendant les étapes d'ajout d'éléments

## 1.7 La recherche d'une donnée

Si on souhaite récupérer le maillon contenant une donnée précise :

```
1 maillon* rechercher( tab_hachage *th, type_data d )
2 {
3     maillon *e;
4     uint32_t p = hachage( th, d );
5     e = th->tab[p]->tete;
6
7     while( e != NULL )
8     {
9         if ( e->data == d )
10        {
11            return e;
12        }
13        e = e->suivant;
14    }
15    return NULL;
16 }
```

## 2 ArrayList

Un autre problème des listes chaînées est le grand nombre d'allocations mémoire qui sont réalisées lors de l'ajout d'éléments, ou de suppressions mémoire effectuées lors de la suppression d'éléments. Or, chaque appel aux fonctions **malloc** ou **free** prend un temps assez long (par rapport aux autres instructions), et il est recommandé d'en diminuer la fréquence d'appel.

Les arraylist apportent une solution à ce problème, en préalloquant de nombreux maillons dès le départ. En effet, il est plus rapide d'effectuer une allocation de  $n$  éléments que  $n$  allocations d'un seul élément.

La figure 3 montre la structure mémoire d'une arraylist.

### 2.1 Version sans libération de la mémoire

#### 2.1.1 Les structures de données associées

##### 2.1.1.1 Déclaration d'un maillon

Un maillon classique de liste

```
1 typedef struct _maillon
2 {
3     type_data data; //type_data aura été défini précédemment
4     struct _maillon *suivant;
5 } maillon;
```

##### 2.1.1.2 Déclaration de l'arraylist

On déclare d'abord une structure de liste de maillons

```
1 typedef struct
2 {
3     uint32_t taille;
4     maillon *tete;
5     maillon *queue;
6 } liste;
```

Ensuite, on peut déclarer la structure d'arraylist qui contient, entre autre, une liste de maillons "libres" (qui ne possèdent pas de données intéressantes, mais serviront à stocker des données plus tard) et une liste de maillons "occupés" (qui possèdent des données intéressantes).

```
1 typedef struct
2 {
3     liste *occupes;
4     liste *libres;
5 } arraylist;
```

FIGURE 3 – Architecture générale d'une arraylist

### 2.1.2 Modification des fonctions existantes sur les listes

**2.1.2.1 Fonctions d'ajout d'éléments dans les listes** Les fonctions d'ajout dans les listes devront maintenant proposer deux possibilités : soit l'ajout directement d'une valeur à la liste, soit l'ajout d'un maillon déjà construit. Dans tous les cas, il ne fait plus faire d'allocation dynamique dans ces fonctions, car les maillons à rajouter seront déjà alloués.

```
1 void add_tete_maillon(liste *l, maillon *m)
2 {
3     assert(m!=NULL);
4     m->suivant = l->tete;
5     l->tete = m;
6     if( l->taille == 0 )
7     {
8         l->queue = m;
9     }
10    l->taille += 1;
11 }
12
13 void add_tete_data(liste *l, type_data d)
14 {
15     add_tete_maillon(l, new_maillon(d));
```

```
16 }
```

On fait pareil pour les autres fonctions d'ajout : ajout en queue, ajout après, etc...

**2.1.2.2 Fonctions de suppression d'éléments des listes** Les fonctions de suppression des listes doivent aussi être modifiées, afin de ne plus faire d'appel explicite à la fonction **free**, mais seulement retourner le maillon supprimé. En effet, on souhaite éviter les appels répétitifs à la fonction **free** ; les maillons supprimés de notre liste devront réintégrer la liste des maillons "libres", et ne pas être supprimés de la mémoire.

```
1 maillon* rem_tete_maillon(liste *l)
2 {
3     maillon *t = l->tete;
4
5     l->tete = l->tete->suivant;
6     l->taille -= 1;
7
8     if( l->taille == 0 )
9     {
10         l->queue = NULL;
11     }
12     return t;
13 }
```

On fait pareil pour les autres fonctions de suppression : suppression en queue, suppression après, etc...

### 2.1.3 Initialisation de l'arraylist

**2.1.3.1 Code de l'initialisation** On initialise l'arraylist en créant un **tableau** de maillons "en avance" que l'on chaîne ensuite dans la liste représentant les maillons libres.

```
1 void agrandir_liste_libre(arraylist *arl, uint32_t taille)
2 {
3     maillon *t;
4     uint32_t i;
5
6     //Pour avoir un code plus simple, on execute cette fonction seulement si la liste
7     //libres est bien vide
8     assert( est_vide(arl->libres) );
9
10    //On construit un tableau de maillons
11    t = malloc(taille*sizeof(maillon));
12    assert(t != NULL);
13
14    //Et on lie les maillons du tableau entre eux
15    for( i=0; i<taille-1; i=i+1 )
16    {
17        t[i].suivant = &( t[i+1] );
18    }
19    t[ taille-1 ].suivant = NULL;
20
21    //Enfin, on ajoute les maillons à la liste des maillons libres
22    arl->libres->tete = &( t[0] );
23    arl->libres->queue = &( t[taille - 1] );
24    arl->libres->taille = taille;
25
26
27 arraylist* new_arraylist(uint32_t taille_initiale)
28 {
29     arraylist *arl;
```

```

31 //On construit l'arraylist
32 arl = malloc(sizeof(arraylist));
33 assert(arl != NULL);
34
35 //On construit un tableau de maillons
36 arl->occupes = new_liste();
37 arl->libres = new_liste();
38
39 //On cree des maillons libres
40 agrandir_liste_libre(arl, taille_initiale);
41
42 return arl;
43 }
```

**2.1.3.2 Exemple d'initialisation** Le code suivant montre un exemple d'initialisation d'une arraylist. La figure 4 montre l'état de la mémoire après l'exécution du code d'initialisation.

```
1 arraylist *arl = new_arraylist(4);
```

FIGURE 4 – Etat de la mémoire après l'initialisation d'une nouvelle arraylist

#### 2.1.4 Ajout d'une donnée dans l'arraylist

**2.1.4.1 Code de l'ajout de donnée** Pour ajouter une donnée, on récupère un maillon libre, on lui place la donnée dedans, et on ajoute le maillon à la liste des maillons occupés. On donne ici un exemple pour l'ajout en tête, mais on peut adapter le raisonnement pour tout type d'ajout (ajout en queue, ajout après, ...).

```

1 void add_tete(arraylist* arl, type_data d)
2 {
3     maillon *l;
4
5     //On agrandit la liste des maillons libres si elle est vide...
6     if( est_vide(arl->libres) )
```

```

7   {
8     agrandir_liste_libre(arl, 1.5*arl->occupes->taille); //Ici, le coefficient 1.5
9     est un exemple...
10 }
11 //On recupere un maillon libre
12 l = rem_tete_maillon( arl->libres );
13 //On y stocke la donnee
14 l->data = d;
15 //Et on place le maillon dans la liste des occupes
16 add_tete_maillon(arl->occupes, l);
17 }

```

**2.1.4.2 Exemple d'ajout de données** Le code suivant montre un exemple d'ajout de données dans une arraylist. La figure 6 montre l'état de la mémoire après l'exécution du code d'ajout.

```

1 arraylist *arl = new_arraylist(3);
2
3 add_tete(arl, d1);
4 add_tete(arl, d2);
5 add_tete(arl, d3);
6 add_tete(arl, d4);
7 add_tete(arl, d5);

```

## 2.1.5 Supprimer une donnée de l'arraylist

**2.1.5.1 Code de la suppression de donnée** Pour supprimer une donnée, on récupère le maillon occupé qui nous intéresse, on prend sa donnée, et on ajoute le maillon à la liste des maillons libres. On donne ici un exemple pour la suppression de tête, mais on peut adapter le raisonnement pour tout type de suppression (suppression en queue, suppression après, ...).

```

1 type_data rem_tete(arraylist* arl)
2 {
3   maillon *l;
4   type_data d;
5
6   //On recupere le maillon de tete des maillons occupes
7   l = rem_tete_maillon( arl->occupes );
8   //On stocke la donnee
9   d = l->data;
10  //Et on place le maillon dans la liste des occupes
11  add_tete_maillon(arl->libres, l);
12
13  return d;
14 }

```

**2.1.5.2 Exemple de suppression de données** Le code suivant montre un exemple de suppression de données de l'arraylist de l'exemple précédent. La figure ?? montre l'état de la mémoire après l'exécution du code de suppression.

```

1 d = rem_tete(arl);
2 d = rem_tete(arl);

```

## 2.1.6 Libération mémoire de l'arraylist

La libération mémoire ne peut pas se faire car les maillons ayant été alloués par groupe, ils doivent être libérés de la même manière. Faire un free sur chaque maillon des deux listes ne fonctionnerait pas, car les allocations ont été faites par paquet. Il est donc nécessaire, pour faire la libération mémoire de la structure, de conserver dans une troisième liste les adresses des tableaux de maillons alloués par notre fonction d'allocation.

## 2.2 Version avec libération de la mémoire

Afin de pouvoir libérer la structure de la mémoire, nous avons besoin de faire quelques modifications.

### 2.2.1 Les structures de données associées

**2.2.1.1 Déclaration des maillons** Nous aurons besoin d'un maillon qui contient des tableaux de maillons :

```
1 typedef struct _tab_maillon
2 {
3     maillon *tab; //Un tableau de maillons
4     uint32_t taille_tab; //La taille du tableau de maillons
5     struct _tab_maillon *suivant;
6 } tab_maillon;
```

**2.2.1.2 Déclaration de l'arraylist** On déclare en plus une structure de liste de tab\_maillons

```
1 typedef struct
2 {
3     uint32_t taille;
4     tab_maillon *tete;
5     tab_maillon *queue;
6 } liste_tab_maillons;
```

Ensuite, on peut modifier la structure d'arraylist :

```
1 typedef struct
2 {
3     liste *occupes;
4     liste *libres;
5
6     liste_tab_maillons *ltm;
7 } arraylist;
```

### 2.2.2 Initialisation de l'arraylist

On initialise l'arraylist en créant un **tableau** de maillons "en avance" que l'on chaîne ensuite dans la liste représentant les maillons libres. L'adresse du tableau de maillons sera sauvegardée dans une troisième liste, pour permettre la libération mémoire plus tard.

```
1 tab_maillon* new_tab_maillon(uint32_t taille)
2 {
3     uint32_t i;
4
5     //On construit le tab_maillon
6     tab_maillon *t = malloc(sizeof(tab_maillon));
7     assert(tab_maillon != NULL);
8
9     //Et on construit le tableau de maillons associés au tab_maillon
10    t->tab = malloc(taille*sizeof(maillon));
11    assert(t->tab != NULL);
12
13    //Et on lie les maillons du tableau entre eux
14    for( i=0; i<taille-1; i=i+1 )
15    {
16        t->tab[i].suivant = &(t->tab[i+1]);
17    }
18    t->tab[ taille-1 ] = NULL;
19
20    t->taille_tab = taille;
21}
```

```

22     return t;
23 }
24
25
26 void agrandir_liste_libre(arraylist *arl, uint32_t taille)
27 {
28     tab_maillon *t;
29
30     //Pour avoir un code plus simple, on execute cette fonction seulement si la liste
31     //libres est bien vide
32     assert( est_vide(arl->libres) );
33
34     //On construit un tableau de maillons
35     t = new_tab_maillon(taille);
36
37     //Et on l'ajoute a la liste contenant tous nos tableaux
38     add_tete_maillon(arl->ltm, t); //La fonction d'ajout en tete d'une liste de
39     //tab_maillon d'un tab_maillon
40
41     //Enfin, on ajoute les maillons "manuellement" a la liste des maillons libres
42     arl->libres->tete = &( t->tab[0] );
43     arl->libres->queue = &( t->tab[t->taille_tab - 1] );
44     arl->libres->taille = t->taille_tab;
45 }
46
47 arraylist* new_arraylist(uint32_t taille_initiale)
48 {
49     arraylist *arl;
50
51     //On construit l'arraylist
52     arl = malloc(sizeof(arraylist));
53     assert(arl != NULL);
54
55     //On construit un tableau de maillons
56     arl->occupes = new_liste();
57     arl->libres = new_liste();
58     arl->ltm = new_liste_tab_maillons();
59
60     //On cree des maillons libres
61     agrandir_liste_libre(arl, taille_initiale);
62
63     return arl;
64 }
```

### 2.2.3 Libération mémoire de l'arraylist

Grâce à la liste *liste\_tab\_maillons*, nous avons sauvegardé les adresses des tableaux créés par la fonction d'allocation dynamique : il suffit de libérer ces tableaux.

```

1 void vider(arraylist *arl)
2 {
3     tab_maillon *t;
4
5     while( ! est_vide( arl->ltm ) )
6     {
7         t = rem_tete_maillon( arl->ltm );
8         free(t->tab);
9         free(t);
10    }
```

```
11     free(arl->libres);
12     free(arl->occupes);
13     free(arl->ltm);
14
15     free(arl);
16 }
17 }
```

FIGURE 5 – Etat de la mémoire après l'ajout de données dans l'arraylist

FIGURE 6 – Etat de la mémoire après la suppression d'une donnée de l'arraylist