*Chapitre V*

# Gestion mémoire

## Sommaire

Le C/C++ est un langage dans lequel la gestion mémoire se déroule de la façon suivante :

- pour toute fonction/méthode, la quantité de mémoire nécessaire à son exécution est déterminée à la compilation,
  la mémoire est réservée à l'appel de la fonction, et libérée au retour.
  à savoir les paramètres, variables locales et tableaux statiques.
- la mémoire dynamique est explicitement allouée et libérée par l'utilisateur.
  elle survit au retour des fonctions.
  il n'existe pas de ramasse-miette.

En conséquence, il n'est pas possible en C/C++ de se passer de la gestion de la mémoire.

Nous allons voir :

- les bases de l'utilisation des pointeurs et des allocations mémoires statiques et dynamiques,
- comment gérer la mémoire d'un objet dans les définitions de classe,
- les sémantiques les plus communes associées aux pointeurs à travers les pointeurs intelligents.

# 1 Mémoire

Il faut distinguer :
- la mémoire de masse (= disque dur).
- la mémoire vive (= RAM).
- les mémoires cache (= mémoire intermédiaire du processeur).
- les registres mémoire (= mémoire interne au processeur).

Les deux derniers niveaux de mémoire sont internes au processeur (dépendent du modèle de processeur).

L'utilisation de ces mémoires est la suivante :
- Tout programme ou données stockées sur le disque dur doit être chargée dans la RAM pour pouvoir être utilisée.
- Lors de l'exécution, les portions du programme et de données nécessaires de la partie du code en cours d'exécution sont transférés vers le processeur à travers le cache. A savoir,
  - ◇ le processeur n'accède jamais directement à la RAM,
  - ◇ cache = copie locale de portions de la mémoire

## 1.1 Mémoire d'un ordinateur

La mémoire d'un ordinateur peut être vue comme un tableau unidimensionnel de très grande taille :
- une case de ce tableau permet de stocker un octet (= 8 bits).
- chaque case est numérotée (implicitement, par son numéro d'ordre).
  le numéro de cette case est appelée son **adresse mémoire**.
- lorsqu'un objet de type `T` est stocké dans la mémoire, les `sizeof(T)` octets qu'il occupe sont toujours stockés :
  - ◇ dans des cases consécutives.
  - ◇ à une adresse multiple du `sizeof(T)`

  ceci s'appelle l'**alignement mémoire**.

  **Exemple :**

  une variable de type double, utilise 8 cases consécutives dans la mémoire, et le numéro de la case où elle est stockée est nécessairement multiple de 8.

## 1.2 Mémoire d'un programme

La mémoire d'un programme est typiquement découpé en segments :
- le **segment de code** (ou texte) qui contient le code compilé du programme (en lecture seule),
- le **segment bss** (pour Block Started by Symbol) ou non initialisé contenant les variables globales et les variables statiques qui ne sont pas initialisés dans le code.
  valeur par défaut à 0 (règle du standard : pas de coût = initialisé à 0).
- le **segment de données** (ou initialisé) contenant les variables globales et les variables statiques qui sont initialisés,
- la **pile d'appel** (call stack) qui stocke les variables locales et la pile d'appel des fonctions (voir ci-après).

- le **heap** où sont stockés les variables allouées dynamiquement (voir ci-après).

## 1.3 Heap et stack

**Stack :** zone mémoire spécifiquement utilisée pour toutes les variables locales et la pile d'appel des fonctions (rappel : toute variable locale est dans l'environnement local d'une fonction).

L'allocation mémoire dans le stack est relativement rapide, et les variables locales sont généralement placées dans la cache primaire ou les registres.

La quantité de mémoire nécessaire alloué lors de l'appel d'une fonction (= emplacement de stockage de chaque variable) est connue à la compilation.

La mémoire alloué dans le stack reste accessible tant qu'elle n'est pas dépilée du stack :
- toutes les variables dans l'environnement local d'une fonction est désallouée au retour de la fonction,
- l'espace local utilisé par une variable locale peut être recyclé dès qu'elle n'est plus utilisée.

Sa taille est **fixe** mais configurable à la compilation et **limitée** (quelques centaines de Ko, voir quelques Mo).
- Tout ce qui consomme beaucoup de mémoire dans le stack n'est pas une bonne idée (passage par valeur, tableaux statiques de grande taille, appels récursifs trop profond).
- dépassement de la capacité du stack = erreur `stack overflow`.

le **heap** (ou **free store**) : zone mémoire dans laquelle est allouée la mémoire dynamique.
- La mémoire allouée est généralement calculée au cours de l'exécution du code.
- La mémoire allouée dans le heap reste allouée jusqu'à ce qu'elle soit spécifiquement desallouée (sinon = fuite mémoire) ou que le programme se termine.
- L'allocation de la mémoire dans le heap est relativement lente (gestionnaire allocation/desallocation + verrou système pour multithreading).
- On accède à la mémoire dynamique à travers un pointeur : déréférencer un pointeur est plus lent que d'accéder directement à une variable.
- Elle est limitée par la mémoire physique disponible sur le système (= grand espace de stockage). La présence d'un swap (mémoire virtuelle sur le disque) permet de repousser la limite de la RAM, au coût d'accès disque.
- En cas de requête provoquant un dépassement, la mémoire n'est pas allouée (le cas doit être géré au niveau du code).

Pour le C++, notons les nuances suivantes :
- une zone de mémoire spécifique (**const data area**) contenant l'ensemble des constantes (= déclarées avec le modificateur `const`) dont la valeur est connue à la compilation.
  **Particularité :**
  - ◇ zone de mémoire en lecture seule,
  - ◇ le résultat de la modification de cette zone mémoire n'est pas définie,
  - ◇ ne rien supposer sur le stockage sous-jacent dans cette zone car le compilateur peut l'optimiser.
- le norme C++ indique que :
  - ◇ le **Free Store** stocke les allocations/desallocations faites avec `new`/`delete`.
  - ◇ le **Heap** stocke les allocations/desallocations faites avec `malloc`/`free`.

sans spécifier s'il s'agit d'une seule et même zone. Néanmoins, même si cela était le cas, il est possible d'allouer des objets C++ dans le Heap (voir allocation en place).

## 1.4 Retour sur les classes de stockage

Les objets qui ont une durée de stockage :

- **automatique** sont stockées dans le stack,
- **statique** (variable statique ou globale) sont dans le segment de données (si initialisées) et dans le segment bss (sinon),
- **par thread** sont stockées dans le stack. Dans le cas d'un programme multithread, il y a un stack particulier par thread.
- **dynamique** sont stockées dans le heap.

La connaissance de la durée de stockage permet donc de savoir exactement où la variable est stockée, et comment elle est gérée lors de l'exécution du programme.

## 1.5 Stockage mémoire

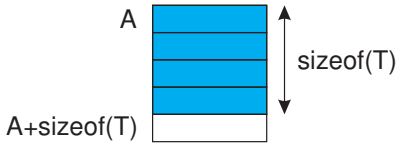
Nous reprenons ce que nous avions déjà dit en introduction.

Toute variable est caractérisée par :

- son nom = le symbole $S$ utilisé pour la désigner dans son contexte.
- son adresse $A$ = le numéro de la case mémoire à partir de laquelle elle est stockée.
- son type $T$ qui permet d'indiquer :
  - ◇ le nombre d'octets nécessaires au stockage d'une variable de ce type.
  - ◇ la façon d'interpréter la zone de mémoire qui stocke la variable afin de construire la valeur de la variable.

**Note :** en C/C++, la fonction `sizeof(T)` permet de retourner cette taille (en octets).

Donc, le symbole $S$ occupe les cases mémoire : $\{A, \ldots, A + \texttt{sizeof}(T) - 1\}$




A noter que :

- seul le contexte (variable locale/allocation dynamique) détermine si la mémoire utilisée sera dans le heap ou le stack.
- l'utilisateur ne choisit **jamais** où l'allocation sera effectuée. Elle est déterminée automatiquement par l'allocateur interne de l'application.
  **Conséquence :** il est donc absurde d'affecter une valeur numérique autre que le pointeur null ou une adresse mémoire qui pointe sur une zone mémoire allouée.

# 2 Pointeur et référence

## 2.1 Principes

**Définition :** un pointeur sur un type `T` (noté `T*`) est une variable dont la valeur est une adresse mémoire à laquelle se trouve une valeur de type `T`.

**Important :** un pointeur ne stocke qu'une adresse, et elle doit toujours contenir l'adresse d'une zone mémoire allouée (= pointer sur une case mémoire allouée).

**Opérateurs :**
- `&` : sur une variable, permet d'obtenir l'adresse de cette variable (à savoir, l'endroit où sa valeur est stockée dans la mémoire,
- `*` : pour un pointeur, permet d'obtenir la valeur stockée à l'adresse sur laquelle il pointe.

En conséquence, si `T` n'est pas un type pointeur,

| Déclaration | type de `&a` | type de `a` | type de `*a` |
|---|---|---|---|
| `T a;` | `T*` | `T` | pas de sens |
| `T *a;` | `T**` | `T*` | `T` |

## 2.2 Règles de manipulation des pointeurs

**Pointeur nul :** un pointeur est dit nul s'il pointe sur rien.

**Règles :**
- un pointeur doit toujours être initialisé à sa création.
  - ◇ en C++, cela est presque toujours possible (= le déclarer à l'endroit où l'on connaît sa valeur).
  - ◇ en C, si sa valeur n'est pas connue à l'initialisation, lui donner la valeur `NULL`.
- un pointeur doit toujours pointer sur une zone mémoire valide ou être nul.
- lorsque la zone mémoire sur laquelle pointe le pointeur ne doit plus être utilisée ou est devenue invalide, le pointeur doit être mis à nul.

**Définition d'un pointeur nul :**
- en C ou C++, utiliser `NULL` (= 0L)
  **Exemple :** `int *a = NULL;`
- en $C^{++}_{11}$, utiliser `nullptr`
  **Exemple :** `int *a = nullptr;`

**Note :** lors d'une évaluation, oubli d'initialisation d'un pointeur = 0 à la question.

### a) `void*` et casting

Un valeur d'un pointeur est juste une adresse mémoire.

**Nouveau type :** `void*` = pointeur non typé.
- permet de déclarer un pointeur non typé.
- l'opérateur `*` n'a pas de sens sur un `void*`.

Comment affecter un `void*` ?

- un type `T1*` est incompatible avec un type `T2*` si `T1` ≠ `T2`.
- un **cast** est un opérateur qui force la conversion d'un type dans un autre. La conversion d'une variable `u1` de type `T1` en une variable `u2` de type `T2`. En `C`, il s'effectue avec `u2 = (T2)u1`.

**Exemple :**
```
int  a = 3;
void *b = (void *)&a; // conv int* -> void*
int  *c = (int *)b;   // conv void* -> int*
```

**Attention :** assurez-vous de la cohérence de ce qui est écrit.

Dans l'exemple précédent, écrire `float *d = (float*)b` demande d'interpréter la représentation binaire de l'entier 3 comme celle d'un flottant.

**Conséquence :** le cast `C` est donc potentiellement dangereux. Il ne doit **jamais** être utilisé pour convertir des pointeurs (utiliser les casts `C++` spécifiques : `static_cast` ou `dynamic_cast`).

#### b) Problème de `NULL`

**Attention :** l'utilisation `NULL` en `C++` peut poser des problème :

- l'entier 0 ou `NULL` représentent le pointeur nul dans un contexte où il peut être interprété comme tel (ne sont pas proprement des types pointeur).
- les conversions `long`→`int`, `long`→`bool`, `0L`→`void*` sont considérées comme aussi bonnes les unes que les autres.

**Exemple :** si une fonction `f` a 3 surcharges `f(int)`, `f(bool)` et `f(void*)` :

- `f(0)` appelle `f(int)` jamais `f(void)`.
- `f(NULL)` peut ne pas compiler, sinon appelle `f(int)`.
  **Conséquence :** il est nécessaire d'éviter les surcharges d'une fonction ayant un argument de type pointeur avec une fonction avec un argument de type entier.

En $C^{++}_{11}$,

- utiliser `nullptr` comme pointeur nul partout où le type utilisé est bien un pointeur (noter que dans le cas précédent `f(nullptr)` appelle bien `f(void*)`).
- Le type associé à un pointeur de type nul est `std::nullptr`.
  peut être utilisé pour surcharger les fonctions passant un pointeur nul.
  **Exemple :** `f(int*)`, `f(float*)`, `f(nullptr_t)`.

## 2.3 Référence

**Définition :** une référence de type `T` (noté `T&`) est une façon de donner un autre nom à une variable déjà existante.

**Conséquence :** une variable et une référence à cette variable ont la même adresse mémoire.

**Utilisation des références :**

- **passage d'une variable par référence** : en passant la référence, le paramètre est un autre nom pour la variable passée en paramètre, permettant ainsi sa modification.

**Exemple :**

```
...
int   v = 3;
fun(v);
...
```

```
void fun(int &a) {
    ...
    a = 2;
}
```

- **retour d'une référence** : permet de retourner une référence passée en paramètre.

**Exemple :**

```
int  u=2, v=3;
max(u,v) = 7;
```

```
int &max(int &a, int &b) {
    return (a < b ? b : a);
}
```

## 2.4 Utilisations des références

**Conséquence :**

Le passage par référence est le nouveau mode de passage de paramètres à privilégier.

En C, un paramètre est passé par pointeur (seule alternative de la copie) afin de le modifier ou si le paramètre est une structure/objet long à copier.

Le passage par référence permet ainsi un accès direct sans copie.

Le modificateur `const` permet de passer par référence en lecture seule.

**Notes :**

- les références sont uniquement utilisables en C++,
- une référence & peut uniquement faire référence à une variable qui peut être nommée, et non à une constante numérique ou à une expression temporaire.
  **Exemple :** `int &v = 4` ou `4+u` n'est pas possible).
- le modificateur `const` lève cette restriction.
  **Exemple :** `const int &v =4+u` est possible.
  `const int&` signifie référence vers un entier non mutable (et non référence constante vers un entier).
  c'est, avant le $C^{++}_{11}$, le seul l'outil disponible permettant de faire référence à un objet temporaire (i.e. d'obtenir l'adresse).

### a) Paramètres et `const`

Lorsqu'un pointeur sur un tableau dynamique est passé en paramètre d'une fonction, le comportement est le même que pour un tableau statique : seul une copie de l'adresse du pointeur est passée.

Par défaut, il est donc possible de modifier le contenu du tableau.

`const` permet de modifier le comportement d'un paramètre. Pour un type pointeur `T*` :

- `T*` : pointeur et valeurs pointées modifiable
- `const T*` : pointeur modifiable, valeurs pointées constantes
- `T* const` : pointeur constant, valeurs pointées modifiables.
- `const T* const` : pointeur et valeurs pointées constants
- `const T&` : référence constante

Pour résumer :

| paramètre | lire v | modifier v | lire v[i] | modifier v[i] |
|---|---|---|---|---|
| `int *v` | oui | oui | oui | oui |
| `const int* v` | oui | oui | oui | non |
| `int* const v` | oui | non | oui | oui |
| `const int* const v` | oui | non | oui | non |
| `const int& v` | oui | non | n/a | n/a |

# 3 Allocation dynamique d'un objet

**Rappel :**

- l'allocation dynamique est le procédé permettant d'allouer de la mémoire (dans le heap) dont la durée de vie n'est pas associée à une variable locale.
- en C, avec `size_t *malloc(size_t)` où `size_t` est la taille de la mémoire à allouer, et retourne un pointeur sur la mémoire allouée.
  **Exemple :**
```
int *v = (int*)malloc(sizeof(int));
```

En C++, l'allocation dynamique d'un objet de type `T` s'effectue avec l'opérateur `new` de la manière suivante :

- `new T` ou `new T()` pour allouer un bloc de taille `sizeof(T)` initialisé avec le constructeur par défaut.
  **Rappel :** pour les BITs, le constructeur par défaut n'initialise rien.
- `new T(`*`args`*`)` pour allouer un bloc de taille `sizeof(T)` initialisé avec le constructeur `T(`*`args`*`)`.

et retourne un pointeur de type `T*` vers le bloc mémoire alloué.

**Exemple :**
```
int *u = new int;      // alloue un entier par défaut
int *v = new int(4);   // alloue un entier initialisé à 4
```

**Remarques :**

- La mémoire est allouée sous forme d'un seul bloc contigu de mémoire.
- `new` retourne `NULL`/`nullptr` lorsque l'allocation a échouée,
  et lance l'exception `std::bad_alloc` (voir chapitre sur la gestion d'exception).
  Ce cas d'erreur doit être traité après chaque allocation (en traitant l'exception ou en testant la valeur retournée).
- L'échec de l'allocation signifie qu'il n'y a pas suffisamment de mémoire libre ou qu'il n'est pas possible d'allouer un bloc contigu de cette taille dans la mémoire (cas de la fragmentation mémoire).
- La mémoire allouée dynamiquement **doit** être désallouée explicitement,
  sinon elle reste allouée jusqu'à la fin de l'exécution du processus associé à l'application.

La désallocation d'un bloc mémoire dynamiquement s'effectue par l'appel explicite de l'opérateur `delete` sur un pointeur au début de ce bloc.

**Exemple :**
```
T *u = new T;
...
delete u; // appelle aussi le destructeur u.~T()
```

**Remarques :**

- tout bloc alloué dynamiquement ne peut être désalloué qu'en totalité.
- l'appel à `delete` :
  - ◇ appelle le destructeur de l'objet,
  - ◇ **puis** désalloue la mémoire dynamique.
- si le pointeur passé à `delete` ne pointe pas sur un bloc alloué dynamiquement (ou nul), l'appel à `delete` engendre en une erreur de segmentation.
  même effet si on tente de desallouer un bloc déjà désalloué.

**Que se passe-t-il une fois un bloc désalloué ?**

le pointeur pointe toujours sur le bloc, sans modification de sa valeur.

il reste possible d'accèder aux données, tant que le bloc n'a pas été réalloué puis réécrit.

**Conséquences :**

- la valeur du pointeur vers le début d'un bloc alloué dynamiquement doit être conservé afin de permettre la désallocation.
- penser à mettre le pointeur à nul.
  on peut utiliser une macro ou un template afin de désallouer et mettre le pointeur à nul.
  **Exemple :**

```
#define SAFE_DELETE(x) { if (x) { delete x; x=NULL; } }
...
SAFE_DELETE(u);
```

```
template <class T> void safe_delete(T* &x)
    { if (x) { delete x; x=nullptr;} }
...
safe_delete(u);
```

# 4 Tableau

**Rappel :** un tableau est un ensemble d'objets de même type indexés par leurs numéros d'ordre.

En mémoire, un tableau de $n$ objets de type `T` est représenté de la manière suivante :

- l'ensemble des objets est placé dans un seul bloc contigu de taille $n \times$ `sizeof(T)`,
- le tableau est représenté par un pointeur `p` de type `T*` sur le début du bloc (= *i.e.* sur le premier élément du bloc),
- les éléments du tableau sont numérotés de 0 à $n-1$,
- l'accès au `i`$^{\text{éme}}$ élément du tableau s'effectue en se décalant de `i` $\times$ `sizeof(T)` octets depuis le début du bloc (*i.e.* de `p`),
- l'adresse du $i^{\text{éme}}$ élément du tableau se calcule comme `p` + `i` $\times$ `sizeof(T)`,
- l'arithmétique des pointeurs fait que pour un pointeur de type `T`, `p` + `i` correspond à l'adresse mémoire `p` + `i` $\times$ `sizeof(T)`,
- l'écriture `p[i]` permet d'accèder au `i`$^{\text{éme}}$ élément du tableau (en lecture ou en écriture) et est strictement équivalente à `*(p+i)`.

Donc un pointeur de type `T*` est homéomorphe à tableau (il contient au moins un élément).

## 4.1 Tableau statique

Un tableau statique est un tableau :

- déclaré comme une variable locale sous la forme : `T v[N];`
  où `T` est le type du tableau, `N` est la taille du tableau et `v` la variable locale associée au tableau.
- dont la taille `N` peut être explicitement calculé par le compilateur.

Un tableau statique se comporte comme une variable locale, à savoir :

- la tableau est alloué dans le stack au lieu de la déclaration de sa variable locale associée, et libéré en fin de portée de cette variable,
- le constructeur du type `T` est appelé sur chaque élément du tableau lors de sa déclaration,
- idem pour le destructeur en fin de portée.
- si `T` est un BIT, les éléments du tableau ne sont pas initialisés.

**Exemple :**

```
const int  n=100;
float b[n];      // éléments de b non initialisés
struct complex { float r,i; complex() :r(0.f),i(0.f) {}; };
complex z[4]; // éléments de z tous initialisés à 0
```

**Remarques :**

- Un tableau statique est homéomorphe à un pointeur, à savoir que lorsque l'on manipule un tableau statique, on manipule en fait un pointeur.
  Un tableau statique peut être passé en paramètre d'une fonction à la place d'un pointeur.
- Si l'argument d'une fonction a la forme d'un tableau statique (exemple : `void fun(int a[4])`), alors les déclarations `void fun(int a[])` et `void fun(int *a)` sont équivalentes. donc, si l'argument d'une fonction a la forme d'un tableau statique, seul le pointeur sera passé (i.e. aucune donnée ne sera copiée), et il ne sera tenu aucun compte de la taille déclarée.
- la fonction `sizeof` ne donne la taille réelle du tableau statique que dans le contexte dans lequel il est déclaré.

  **Exemple :**
  ```
  void fun(int y[5]) { /* sizeof(y) = 4  */
      int x[10];       /* sizeof(x) = 40 */ }
  ```
- Noter que pour un tableau statique `p`, `p` et `&p` sont tous les deux l'adresse du premier élément du tableau.
- Si un tableau statique est déclaré comme champs d'une structure ou d'une classe, alors il est stocké en intégralité dans la structure.

  **Exemple :**
  ```
  struct InTab { int x; int y[10]; };
  // sizeof(InTab) = 11 * sizeof(int)
  ```

### a) Initialisations

**Exemple :**

```
int   a[4] = {1,2,3,4};
int   b[2][2] = {{1,2},{3,4}};
```

**Remarques :**

- l'initialisation n'est possible que sur des tableaux de taille constante.

- on peut initialiser partiellement le tableau (*i.e.* ne donner que les premières valeurs du tableau).
- cette écriture n'est valable **qu'au moment de la déclaration du tableau**.
- les seules autres façons d'initialiser un tableau est, soit d'écrire la valeur élément par élément (boucle `for`), soit d'utiliser une initialisation mémoire par bloc (voir plus loin).
- les valeurs d'un tableau doivent avoir été écrites au moins une fois avant d'être lues.

### b) Boucle sur un tableau statique

Dans le cas particulier d'un tableau statique, celui-ci peut être traité comme un conteneur car son type `T[n]` contient non seulement le type des données stockées dans le conteneur, mais également son nombre d'éléments.

Un `range-for` est alors possible ($C_{11}^{++}$).

**syntaxe :** `for( T x : conteneur ) { ... }`

**Signification :** parcourir l'ensemble des éléments `x` du conteneur.

**Important :** si le conteneur contient des objets de type `T`, utiliser de préférence des références (sinon copies potentiellement coûteuses) :

- `T &x` si le contenu du conteneur dans être modifié pendant la boucle,
- `const T &x` si le conteneur est invariant par la boucle.

**Exemple :**

```
// tableau statique
int array[4] = {1, 2, 3, 4};
// doublement des valeurs du tableau
for(int &x : array) { x *= 2; }
// affichage des valeurs du tableau
for(const int &x : array) {
    std::cout << x << "\n";   }
```

### c) Limites des tableaux statiques

Un tableau statique a des limites importantes :

- sa taille est fixe : doit être une expression numérique, `const` (fixé à la compilation) ou `constexpr` ($C_{11}^{++}$, calculable à la compilation).
- sa taille est limitée par le compilateur à la taille du stack (gcc=7.4Mb, gcc cygwin=2MB, VS=1MB).
- l'allocation dans le stack peut conduire à sa saturation, mais :
  - ◇ la taille du stack peut être modifiée dans les options du compilateur,
  - ◇ le stack doit rester petit pour être efficace : c'est la réutilisation permanente de la mémoire dans le stack qui place la mémoire associée en permanence dans le cache du processeur.
- il est associée à une variable locale dont la portée est celle de la variable.
  **Exemple :** une fonction ne peut pas renvoyer le pointeur d'un tableau statique déclaré dans le code de cette fonction (différent de Java)
- on ne peut pas écrire un code dont, par exemple, le nombre de vecteurs et matrices dépend de paramètres connus à l'exécution.

Donc, les tableaux statiques sont à réserver :

- aux tableaux de petites tailles dont la taille est obligatoirement connue à l'avance.

- dont la portée est limitée.

Dans tous les autres cas, il est nécessaire d'utiliser un tableau dynamique.

# 5 Tableaux dynamiques

Un **tableau dynamique** est un tableau dont :

- la taille peut être calculée lors de l'exécution (en C et en C++).
- l'allocation et la desallocation sont explicitement demandées, et à la charge de l'utilisateur.
- la mémoire est réservée dans le **heap**, et n'a de limite que la mémoire disponible restante sur le système (à la fragmentation près).

C'est la manière usuelle de gérer la mémoire dans un code C/C++.

## 5.1 Allocation en C

L'allocation d'un bloc mémoire s'effectue par l'appel de la fonction `malloc` :

- à laquelle on passe en paramètre la quantité de mémoire à allouer (de type `size_t` ≡ `unsigned int`).
- et qui retourne un pointeur sur le début du bloc alloué ou `NULL` si l'allocation a échouée.

```
size_t *malloc(size_t)
```

La taille passée en paramètre est le nombre d'octets à allouer = N*sizeof(T) pour un tableau de `N` éléments de type `T`.

La mémoire allouée est réservée et non initialisée.

Il est nécessaire de :

- de caster explicitement le type de retour du `malloc` en `(T*)` = conversion du type de pointeur.
- inclure `stdlib.h` au début du code.

**Note :** si `size`=0, retourne un pointeur NULL ou un pointeur valide vers une zone de taille nulle (dépend de l'implémentation).

La libération mémoire s'effectue comme dans le cas de l'allocation dynamique standard avec `free`.

## 5.2 Allocation en C++

En C++, l'allocation dynamique d'un tableau s'effectue avec l'opérateur :

```
T* new T[size_t]
```

où `T` est le type du tableau à allouer, et `size_t` la taille passée en paramètre est le nombre d'éléments du tableau à allouer.

**Remarques :**

- un tableau alloué dynamiquement a toutes les propriétés d'un tableau (allocation contigüe, ...),
- une fois la mémoire allouée, le constructeur **par défaut** est ensuite appelé sur chacun des éléments du tableau.

- ◇ s'il n'y en a pas de défini, la compilation échoue.
- ◇ il n'a pas de possibilité avec cet opérateur de construire un tableau avec un autre constructeur que le constructeur par défaut.
- ◇ **attention :** à ce que fait le constructeur par défaut ne fasse pas d'initialisations inutiles.

**Note :** si `size_t` = 0, retourne un pointeur valide vers une zone de taille nulle,

## 5.3 Exemple

**Exemple :**

```
int *v = new int[3];
struct complex { float x,y; complex() : x(0.f),y(0.f) {}; };
complex *z = new complex[3];
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| v | int* | 0x18E0 |
| z | complex* | 0x18E4 |

**Mémoire :**

| | | | | | |
|---|---|---|---|---|---|
| 0x18E0 | 0xAD28 | 0xAD28 | ? | 0xB240 | 0+i0 |
| 0x18E4 | 0xB240 | 0xAD2C | ? | 0xB248 | 0+i0 |
| ... | ... | 0xAD30 | ? | 0xB250 | 0+i0 |

**Note :**

- pour `v`, la mémoire n'est pas initialisée car `int` est un BIT.
- pour `z`, chaque complexe du tableau alloué est initialisé avec le constructeur par défaut.

**Remarque :** on remarquera qu'ici, il y a dans la mémoire :

- une variable locale stocke l'adresse du bloc allouée.
- un bloc de mémoire contigu réservé pour stocker les données.

## 5.4 Désallocation

La desallocation d'un bloc mémoire s'effectue par l'appel explicite de l'opérateur :

```
void delete [] void*
```

sur l'adresse du début du bloc alloué dynamiquement.

**Exemple:**

```
delete [] z;     // où z tableau de complexes
```

**Notes :** comme pour la désallocation d'un objet

- Le destructeur est **d'abord** appelé sur chaque objet du tableau.
- Puis, le bloc est désalloué **en entier** (libération partielle impossible).
- Les valeurs du tableau **ne sont pas modifiées** par la désallocation.
- Il est donc conseillé de mettre le pointeur desalloué à `NULL` / `nullptr`. évite une réutilisation involontaire du tableau après sa libération (ne l'empêche pas si plusieurs pointeurs différents pointent sur le tableau).

- Le passage d'une adresse invalide à la fonction de libération provoque un `segmentation fault` (idem si l'adresse passée est `NULL` / `nullptr`).

## 5.5 Accès

**Accès aux éléments :** comme pour un tableau statique avec l'opérateur `[]`.

Les précautions à prendre lors de l'accès à un tableau alloué dynamiquement :

- Comme pour les tableaux statiques, aucun accès ne doit aboutir à sortir de la zone mémoire allouée (même peine, même effet).
  Dans le cas de la mémoire dynamique, les débordements ont :
  - ◇ moins de chance de provoquer une erreur de segmentation,
  - ◇ plus de chance de provoquer des effets de bord.
- Toujours conserver l'adresse (du début du bloc) renvoyée par l'opérateur d'allocation.
  Elle est nécessaire pour désallouer le bloc alloué.
- Ne jamais tenter d'accéder à un bloc qui a été desalloué.
  N'étant plus réservé, il peut être réalloué et réécrit.

## 5.6 Initialisations

Pour un BIT, utiliser si possible les opérations mémoires suivantes beaucoup plus rapide qu'une boucle `for`.

- **à zéro :** `memset(u,0,n*sizeof(T))`.
  pour initialiser à 0 les `n` premiers `T` du tableau `u`.
  **remarque :** 0 est codé par une suite de 0 pour tous les types.
  **note :** ne jamais utiliser cette fonction pour initialiser à autre chose que 0, sauf `sizeof(T)` = 1.
- **par copie :** en copiant un tableau déjà existant. `memcpy(u,v,n*sizeof(T))` : pour copier les `n` premiers `T` du tableau `v` dans ceux de `u`.
- inclure `string.h` pour utiliser `memset` et `memcpy`.

Pour un objet,

- c'est le constructeur par défaut qui s'occupe des initialisations (et le destructeur des destructions).
- ne **jamais** utiliser de raz/copie mémoire sur un objet dont le constructeur/assignation par copie/déplacement est défini (car alors la bonne façon d'effectuer une copie **n'est pas** une copie mémoire).
- dans tous les cas, ne **jamais** utiliser de raz/copie mémoire sur un objet qui possède une méthode virtuelle ou fait de l'héritage virtuel (VPTR, ...)

## 5.7 Différence C et C++

Pourquoi des gestions mémoires différentes entre C et C++ ?

Les fonctions de gestion mémoire :

- en C, les fonctions ne font que réserver et libérer la mémoire.
- en C++,

- ◇ la fonction d'allocation réserve la mémoire **et** exécute une fonction d'initialisation automatique (le constructeur).
- ◇ le fonction de desallocation exécute une fonction de nettoyage automatique (le destructeur) **et** libère la mémoire.

**Conséquence :**

- si les constructeur/destructeur ne sont pas définis par défaut **et** qu'un constructeur par défaut peut être défini par le compilateur, `new`/`delete` fonctionnent comme `malloc`/`free`
- ne **jamais** mélanger les allocations : un `new` ne doit pas être libéré par un `free` ; inversement, `malloc` ne doit pas être libéré par un `delete`.
  Provoque des erreurs mémoires car ce n'est pas le même gestionnaire de mêmoire qui est utilisé.

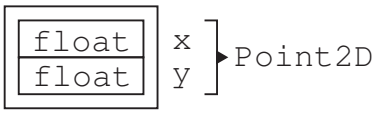
# 6 Conception des structures

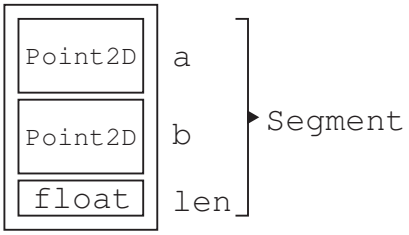
Les structures de base se construisent typiquement de trois manières (non exclusives) :

**Façon 1 : Agrégation des données**

grouper des types (élémentaires ou pas) qui forment ensemble un nouvel objet complexe.

**Exemple :** `Point2D`, `Segment`, ...




```
struct Point2D {
   float x,y;
}
```

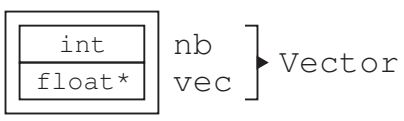
```
struct Segment {
   Point2D a,b;
   float   len;
}
```

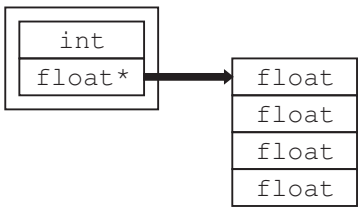
**Façon 2 : utilisation du contenu d'un tableau dynamique**

La taille d'une structure étant fixe, l'ajout d'un tableau dynamique dans une structure s'effectue en ajoutant un pointeur dans la structure dont le rôle est de pointer vers le tableau.

**Exemple :** un vecteur, une matrice, ...

**Cas du vecteur :**




```
struct Vector {
   int   nb;
   float *vec;
};
```

On remarquera que les données du tableau sont stockée à l'extérieur de la structure, et doivent être

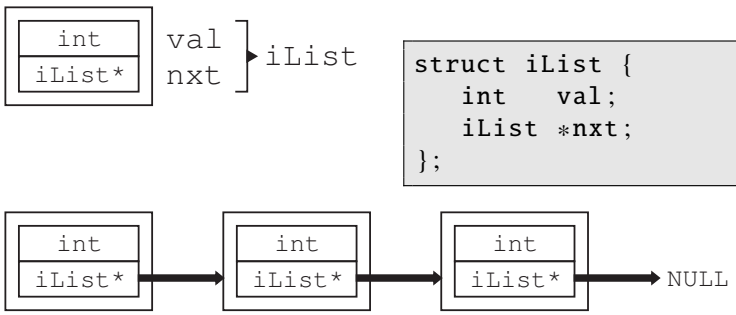
allouées en plus de la structure.

Il ne suffit donc pas de copier la structure pour dupliquer le vecteur (le constructeur et l'assignation par copie sont précisément faits pour ce cas).

**Façon 3 : construire des liens entre agrégats** (pointeur de la structure vers une autre structure)

Chaque structure représente un objet simple d'une structure plus complexe. Ce sont les liens entre les structures qui construisent la structure.

**Exemple :** une liste chainée (structure = un nœud, pointeur = lien entre éléments)




Si `iList` représente un nœud de cette liste, la structure de données complète est représentée par l'ensemble des nœuds reliées entre eux par les pointeurs.

A noter que la définition de la structure est récursive.

L'objet symbolique représenté par la structure se décompose donc en trois composantes :

- une partie "statique" incluse dans la structure, et spécifique à la structure.
- une partie "dynamique" externe à la structure mais liée à elle (et uniquement à elle) par des pointeurs.
- une partie "méta-structure" à travers les liens de la structure avec d'autres structures dont le tout représente un objet symbolique plus vaste.

La cohérence de chaque instance de l'objet représenté par une structure doit être assurée par chaque fonction qui manipule l'instance.

Les pointeurs dans un objet peuvent ainsi représenter plusieurs sémantiques différentes :

- **composition :** si le pointeur pointe vers des données privées (i.e. dont la durée de vie est la même que celle de l'objet.
  ces données privées peuvent elles-mêmes utiliser des pointeurs (exemples des listes chainées).
- **agrégation :** si le pointeur pointe vers un composants externe (i.e. dont la durée de vie est propre au composant).
- **navigabilité :** assurée par un pointeur (par un tableau de pointeurs de taille `N` pour une cardinalité `N`).

Les pointeurs peuvent être du type d'une classe parente permettant le polymorphisme des objets agrégés ou vers lesquels il est possible de naviguer.

# 7 Pointeur intelligent

Les pointeurs intelligents sont des patrons de classe permettant de :

- une libération automatique des ressources associées au pointeur en fin de portée,
  associe le pointeur à un objet qui est alloué dans son constructeur, et desalloué dans son destructeur.
- représenter les sémantiques les plus courantes des pointeurs,
  par sémantique, on entend les usages les plus courants d'un pointeur.
- une gestion de la mémoire sans fuite mémoire, même en cas d'exception,
- une gestion du multithreading à condition que les constructeurs appropriés soient utilisés.

En conséquence, lorsqu'un pointeur doit être utilisé, le pointeur intelligent correspondant à la sémantique recherchée devraient être utilisé à la place.

Il existe trois types de pointeur intelligent en $C_{11}^{++}$ :

- `unique_ptr` : personnification de la sémantique de propriété exclusive,
  correspond à une zone mémoire pointée par un pointeur unique.
- `shared_pointer` : personnification d'un objet auquel plusieurs pointeurs font références, et dont la durée de vie dépend du nombre de pointeurs qui y font référence.
  correspond à une zone mémoire partagée par plusieurs pointeurs.
- `weak_ptr` : pointeur intelligent qui a une référence non propriétaire sur un objet partagé par un `shared_pointer`.
  correspond à un pointeur faible sur un pointeur partagé (ne possède pas la ressource, et doit vérifier si l'accès à la ressource est possible).

**Remarque :** l'ensemble des pointeurs intelligents sont définis dans `memory` (*i.e.* faire `#incude <memory>`).

## 7.1 `unique_ptr`

Personnification de la sémantique de propriété exclusive :

- possède la ressource sur laquelle il pointe,
- **déplacement :** déplace la propriété du pointeur source vers le pointeur de destination (la source est ensuite mise à null, donc ne doit pas être const),
- **copie :** non autorisée (sinon deux pointeurs vers la même ressource).
- **destruction :** détruit la ressource si le pointeur est détruit ou s'il est assigné à une autre ressource.
- un `const unique_ptr` a une vie limitée à la portée dans laquelle il est déclarée.

**Exemple :**

```
int main() {
    std::unique_ptr<A> p1(new A);  // p1 possède A
    if (p1) p1->bar();
    {   // propriété transférée à p2
        std::unique_ptr<A> p2(std::move(p1));
        f(*p2);
        // propriété retournée tout p1
        p1 = std::move(p2);
    }
    if (p1) p1->bar();
}   // fin de portée p1: destruction
```

**Caractéristiques :**

- la taille d'un `unique_ptr` est celle d'un pointeur standard.
- **modificateurs :**
  - ◇ `release` : retourne l'objet géré et relâche la propriété,
  - ◇ `reset` ou `operator=` : remplace l'objet géré
  - ◇ `swap` : échange l'objet géré
  - ◇ utiliser `std::move` pour tranférer la propriété.
- **observateurs :**
  - ◇ `get` : retourne un pointeur vers l'objet géré
  - ◇ `operator bool` : vérifie si le pointeur est affecté
- **gestion d'exception :** garantie la suppression de l'objet sur sortie normale ou suite à une exception.

**Notes :**

- `std::make_unique` donne un template rendant la construction d'un `unique_ptr` plus facile ($C_{15}^{++}$, dispo. dans VS2015)

```
// unique_prt sur un A (avec constructeur par défaut)
std::unique_ptr<A> a1 = std::make_unique<A>();
// unique_prt sur un A (avec constructeur spécialisé)
std::unique_ptr<A> a2 = std::make_unique<A>(0, 1, 2);
```

- un destructeur spécifique peut être passé au constructeur, mais fait passer la taille `unique_ptr` d'un word à deux words.

**Cas des tableaux :**

- utiliser le constructeur : `std::unique_ptr<T[]>` ou `std::make_unique<T[]>`,
- accès aux éléments : utiliser `operator[]`.

**Exemple 1 :**

```
std::unique_ptr<A[]> p1(new A[50]);
std::unique_ptr<A[]> p2 = std::make_unique<A[]>(50);
p1[0]=4;
```

**Exemple 2 :**

```
class A {
private : std::unique_ptr<int[]> x, y;
public :
  A(std::size_t xSize, std::size_t ySize):
        x(std::make_unique<int[]>(xSize)),
        y(std::make_unique<int[]>(ySize)) {}
  ~A() {}  // utilise le destructeur de unique_prt
  // ...
};
```

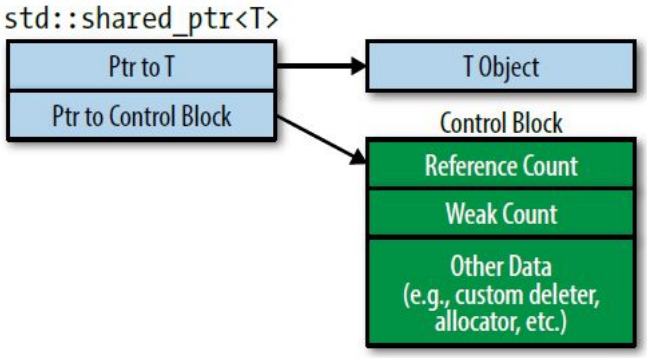
## 7.2 shared_ptr

Personnification d'un objet auquel plusieurs pointeurs font références, et dont la durée de vie dépend du nombre de pointeurs qui y font référence.

- l'objet n'est possédé par aucun pointeur qui y fait référence,
- l'objet est automatiquement détruit lorsqu'il n'y a plus aucun pointeur qui pointe vers lui (destruction déterministe sans garbage collector) à travers un compteur de référence,
- un constructeur incrémente le compteur, un destructeur le décrémente. Note que l'incrémentation et la décrémentation des références doit être fait de manière atomique (=support multi-thread), ce qui est coûteux.
- **copie par assignation :** sp1=sp2 (fait pointer sp1 vers l'objet de sp2 - donc incrémente le compteur de l'objet référencé, mais s'il fait lui-même référence à un autre objet, le compteur de cet objet est décrémenté.
- **déplacement (move) :** sp1 -> sp2, ne change pas le compteur de référence de l'objet pointé par sp1, mais décrémente celui de sp2 s'il n'est pas nul. En conséquence, une move est plus rapide qu'une copie.
- un version spécialisée de `std::swap` existe et permet d'échanger deux pointeurs.

**Caractéristique :**

- la taille d'un `shared_pointer` est le double d'un pointeur standard. L'utilisation d'un destructeur spécifique ne change par la taille d'un `shared_pointer` (voir la structure ci-dessous)
- le bloc de contrôle est créé et initialisé par le premier pointeur partagé créé sur l'objet (rappel : deux threads peuvent demander en même temps la création du pointeur partagé),
- même chose si un `shared_pointer` est créé à partir d'un `unique_ptr`.




**Attention :**

- `std::shared_ptr<T>(new T(args))` effectue (au moins) deux allocations (une pour le bloc de contrôle et une pour l'objet), ce qui poser des problèmes en terme de gestion d'exception et/ou de thread-safety.
- toujours utiliser `std::make_shared` pour construire un `shared_ptr<T>` en une seule allocation (plus efficace, thread-safe, gestion d'exception).
- la gestion d'un objet `std::shared_ptr<T>` est garantie d'être thread-safe, mais pas l'objet possédé lui-même.

**Exemple :**

```
std::shared_ptr<int>  *a = std::make_shared<int>();
std::vector<std::shared_ptr <std::string>> all;
all.emplace_back(std::make_shared <std::string>("un"));
all.emplace_back(std::make_shared <std::string>("deux"));
all.emplace_back(std::make_shared <std::string>("trois"));
std::vector<std::shared_ptr <std::string>>
                    some(all.begin(), all.begin() + 2);
for(auto &x:all)
   std::cout << *x << "=1" << x.use_count() << std::endl;
// sortie: one=2 two=2 three=1
```

**Remarque :**

un `shared_pointer` a un surcoût par rapport à un `unique_ptr` : toujours préférer `unique_ptr` sauf si la possession partagée est nécessaire.

## 7.3 `weak_ptr`

Un `weak_ptr` est un pointeur intelligent qui a une référence non propriétaire (faible) sur un objet partagé par un `shared_pointer`.

Plus précisément,

- à la construction, le `weak_ptr` est initialisé avec un `shared_pointer` (peut lancer une exception `std::bad_weak_ptr` n'est pas valide).
- quand l'on souhaite accéder à un objet seulement s'il existe, et qu'il peut être effacé à n'importe quel moment par un autre thread, `weak_ptr` permet de suivre l'objet :
  - ◇ si l'objet existe, alors il est converti en `shared_pointer` pour assurer une propriété temporaire,
  - ◇ sinon, l'objet doit être considéré comme expiré.

**Méthodes :**

- `expired` permet de vérifier si l'objet a déjà expiré,
- `lock` acquiert un verrou sur l'objet référencé (retourne un `shared_pointer`
- `swap` pour permuter deux `weak_ptr`s.

**Utilisation :**

1. déclarer un `weak_ptr` initialisé avec un `shared_pointer`,

2. l'appel à la méthode `lock()` sur le `weak_ptr` renvoie un `shared_pointer` propriétaire si l'objet existe encore, et un `shared_pointer` nul sinon.

   la propriété n'est conservée que jusqu'à la fin de la portée de l'objet `shared_pointer` acquis.

**Exemple :**

```
std::shared_ptr<int> sptr = std::make_shared<int>(10);
std::weak_ptr<int> weak1 = sptr;
sptr.reset( new int(5) );
std::weak_ptr<int> weak2 = sptr;
// ici: auto = std::shared_ptr<int>
if (auto tmp1 = weak1.lock()) { /* valide */ }
else { /* expiré */ }
if (auto tmp2 = weak2.lock()) { /* valide */ }
else { /* expiré */ }
```

`tmp1`/`tmp2` expirent à la fin de la conditionnelle.

# 8 Idiomes

## 8.1 Idiome copy-and-swap

Pour une classe qui doit gérer une ressource (par exemple un tableau de type `T`), une implémentation naïve de la classe est la suivante :

```
class Array {
private: size_t mSize;
         T*     mArray;
public:
  Array(std::size_t size = 0)
    : mSize(size), mArray(mSize?new T[mSize]():nullptr) {}
  Array(const iArray &other) : mSize(other.mSize),
    mArray(mSize ? new T[mSize] : nullptr),
    { std::copy(other.mArray, other.mArray + mSize, mArray); }
  Array& operator=(const iArray &p) {
    if (this != &other) {  //(1)
      delete [] mArray;    //(2)
      mArray = nullptr;    //(2)
      mSize = p.mSize;
      mArray = mSize ? new T[mSize] : nullptr;
      std::copy(p.mArray, p.mArray + mSize, mArray);
    }
    return *this;
  }
  ~Array() { delete [] mArray; }
};
```

L'implémentation ci-dessus pose les problèmes suivants :

- si `T` est BIT, le constructeur par copie ne lance pas d'exception (= ne peut pas échouer). Sinon la construction par copie peut échouer.
- (1) permet d'éviter l'auto-assignation ou de détruire le tableau que l'on essaye de copier. dans la plupart des cas, ce test est inutile (et donc ralentit le code).
- (2) le `new` après peut échouer, et en conséquence, l'objet sera incohérent (les données sont perdues et la taille est fausse).

En conséquence, on modifie l'implémentation de l'assignation par copie comme :

```
Array& Array::operator=(const iArray &p) {
  if (this != &other) {
    size_t newSize = p.mSize;
    T*     newArray = newSize ? new int[newSize] : nullptr;
    std::copy(p.mArray, p.mArray + mSize, mArray);
    delete [] mArray;
    mSize = p.mSize;
    mArray = newArray;
  }
  return *this;
}
```

Le code est encore plus long, et l'on voit que le code duplique du code déjà écrit dans le constructeur par copie.

Une solution consiste à écrire le code suivant :

```
class Array { ...
  friend void swap(Array& first, Array& second) {
        std::swap(first.mSize,  second.mSize);
        std::swap(first.mArray, second.mArray);
  }
  Array& operator=(Array other) { // (1)
    swap(*this, other);           // (2)
    return *this;
  }
};
```

1. définir une fonction `swap` qui permute le contenu de deux `Array`.
2. définir l'assignation par copie de façon à ce qu'elle utilise le constructeur par copie et l'opérateur `swap`.
   Le passage de `Array` s'effectue **par copie** dans l'objet local `other`. Puis, `other` est permuté avec `*this`, et libéré en fin de portée.

**Avantage de l'approche :** l'objet est toujours laissé dans un état cohérent même si la construction par copie ou l'allocation échoue. Il résiste donc aux exceptions possibles.

En $C_{11}^{++}$, si l'objet passé en paramètre est un objet temporaire, on va effectuer une copie de cet objet temporaire plutôt que de le réutiliser directement, et perdre ainsi une possibilité d'optimisation.

Mais, si l'on définit le constructeur par déplacement :

```
Array::Array(Array&& other) : Array() {
   swap(*this, other);
}
```

Alors, dans ce cas, lorsque la copie par assignation est lancée avec un `Array` temporaire :

- le passage de `other` s'effectue **par copie**, mais si l'objet passé est temporaire, c'est le constructeur par déplacement qui sera appelé.
  en conséquence, l'objet local `other` construit contient l'objet temporaire.
- le `swap` permute les valeurs entre l'objet local et l'objet courant `*this`,
- puis, libération de l'objet temporaire.

Les deux codes précédents (swap + operator= avec passage par copie + constructeur par déplacement) constituent l'idiome copy-and-swap.

**Note :** `swap` ne fait qu'échanger les valeurs des champs (en les copiant), la "copie" de la ressource est obtenue en copie la valeur du pointeur.

## 8.2 Idiome `pImpl`

Idiome qu'il est important de connaître (**pImpl** = **P**ointer of **Impl**ementation).

**Intérêt :**

- si l'implémentation de la classe transférée demande un temps de compilation important.
- permet d'occulter le fonctionnement d'une classe pour distribuer une bibliothèque.

**Exemple :**

```
//header file: cat.h
class Cat {
private:
  // externe
  class CatImpl;
  // Handle
  CatImpl *impl_;
public:
  Cat(const char *);
// Constr.
  ~Cat(); // Destr.
  Purr();
};
```

```
//CPP file:
#include "cat.h"
class Cat::CatImpl {
  Purr();
  ...
};
// forward (implémentation cachée)
Cat::Cat(const char *c) {
    impl_ = new CatImpl(c); }
Cat::~Cat() { delete impl_; }
Cat::Purr() { impl_->Purr(); }
// implémentation locale
CatImpl::Purr() { ... }
```

### a) Implémentation par objet et déplaçable

En $C_{11}^{++}$, on peut utiliser les pointeurs intelligents pour faciliter l'écriture. Tout d'abord en version, sans partage de l'implémentation, déplaçable mais non copiable.

```
// header file: cat.h
class Cat {
private:
  class CatImpl;
  unique_ptr<CatImpl> impl_;
public:
  Cat(const char *c);
  ~Cat();
  Cat(Cat&&);
  Cat& operator=(Cat &&);
  Purr();
};
```

Les propriétés de `unique_ptr` pourraient être utilisées (= déplacement par défaut du champs) afin que le compilateur génère les codes du constructeur et de l'assignation par déplacement.

Mais, afin que le compilateur génère le code l'opérateur d'assignation par déplacement, il doit savoir

détruire l'objet pointé par `impl_`, mais dans le fichier d'entête, `CatImpl` est un type incomplet.

Ce qui conduit à la définition suivante :

```
// source file: cat.cpp

// implémentation de Cat::CatImpl
class Cat::CatImpl { ... };

// définition des méthodes de Cat
// l'utilisation de =default devient possible
// car CatImpl est maintenant un type complet
Cat::Cat(const char *c):cat_(make_unique<CatImpl>(c)) {};
Cat::~Cat() = default;
Cat::Cat(Cat&&) = default;
Cat::Cat& operator=(Cat &&) = default;
Cat::Purr() { impl_->Purr(); }
```

Le compilateur peut maintenant générer correctement les codes par défaut des destructeurs et du constructeur et de l'assignation par déplacement.

### b) Implémentation par objet, déplaçable et copiable

Si l'on veut version, sans partage de l'implémentation, déplaçable et copiable. On ajoute à la définition de la classe :

```
//header file: cat.h
class Cat {
    // comme avant
    Cat(const Cat &c);
    Cat& operator=(const Cat &c);
};
```

```
// source file: cat.cpp
// comme avant
Cat::Cat(const Cat &c) : impl_(new CatImpl(*c.impl_)) {};
Cat& Cat::operator=(const Cat &c) {
   if (this != &c) impl_.reset(new CatImpl(*c.impl_));
   return *this;
};
```

### c) Implémentation partagée, déplaçable et copiable

Dans le cas où l'on souhaite une implémentation partagée, le problème est plus simple car le destructeur de l'implémentation n'a pas besoin d'être invoquée (utilise directement celui du `shared_ptr`).

```
//header file: cat.h
class Cat {
private:
  class CatImpl;
  shared_ptr<CatImpl> impl_;
public:
  Cat(const char *c);
  // déclarations non nécessaires
  // - constructeur et assignation par déplacement
  // - destructeur non nécessaire
  // assurés par l'implémentation du shared_ptr
  Purr();
};
```

L'implémentation par défaut suffit.

# 9 Allocation en place

On rappelle que le fonctionnement habituel lors de la déclaration/allocation dynamique d'un objet (ou d'un tableau d'objets) de type `T` consiste à :

- réserver la mémoire nécessaire pour stocker un objet de type `T`,
- exécuter le constructeur de `T` pour construire l'objet à l'emplacement réservé.

Or, il existe de nombreux cas où ce fonctionnement est problématique, par exemple lorsque :

- la mémoire doit être allouée, mais que le constructeur à appeler pour initialiser l'objet n'est pas encore connu (par exemple, dans une application temps réel),
- un objet contient un tableau d'objets de type `T` dont on connait la taille, mais pas encore les valeurs (ou gérer par exemple un pile de type `T` de taille maximale fixe préallouée),
- on écrit un allocateur (= gestionnaire devant gérer les allocations/désallocations dans un pool mémoire),
- lorsque la conception exige de construire un objet à un endroit précis dans la mémoire (cas d'objets système).

Pour résoudre ce mode de fonctionnement en C++, nous avons besoin de plusieurs outils :

- un outil d'allocation/desallocation mémoire "indépendant" du constructeur/destructeur :
  - ◇ un allocateur (similaire au `malloc` du `C`) capable d'allouer de la mémoire sans exécuter de constructeur,
  - ◇ un desallocateur (similaire au `free` du `C`) capable de libérer de la mémoire sans exécuter de destructeur.

  outils de gestion mémoire seuls sans construction/destruction.
- un constructeur/destructeur "indépendant" de l'allocation/desallocation :
  - ◇ un **constructeur en place** qui permet de construire un objet à un emplacement mémoire spécifié et préalablement alloué.
    **rappel :** le constructeur n'alloue jamais l'objet lui-même.

◇ un appel explicite du destructeur pour exécuter la libération de la mémoire interne de l'objet.
**rappel :** le destructeur ne désalloue jamais l'objet lui-même.

outils de construction/destruction d'objets seuls sans allouer/desallouer de la mémoire.

## 9.1 Gestion mémoire non typée

Outil d'allocation/desallocation mémoire "indépendant" du constructeur/destructeur :

- `void* operator new(size_t count)` pour allouer un bloc de `count` octets (même fonctionnement qu'un `new` classique).
- `void operator delete(void* ptr)` pour désallouer un bloc alloué.

**Exemple :**

```
// allocation
void *data  = ::operator new(nb_elts * elts_size);
```

```
// desallocation
::operator delete(data);
```

**Notes :**

- Utiliser `T* reinterpret_cast<const T*>(void*)` pour caster les emplacements mémoire `void` en type `T` souhaité.
- Les pointeurs renvoyés par ces fonctions fonctionnent pour tout alignement $\leq$ `alignof(std::max_align_t)`.
- Il est également possible d'utiliser les appels `C` `malloc`/`free` dans un code C++.
- Existe aussi en version `new[]` et `delete[]`.

Allocation d'un objet sur un emplacement mémoire déjà alloué (**construction en place**) :

- la construction d'un objet de type `Type` à l'adresse mémoire `prt` dans un bloc déjà alloué peut être (pour un objet de type `T`) effectué avec : `T* new(ptr) T(args)`.
- la destruction d'un objet sans libérer la mémoire occupée celui-ci est effectué par un appel explicite au destructeur.
  Si `t` est un pointeur sur un objet de type `T`, ceci s'effectue avec `t->~T()`.

**Exemple :**

```
// allocation: place pour 256 éléments de type A
// (aucun constructeur appelé)
A* mem = reinterpret_cast<A*>(::operator new(256*sizeof(A)));
// appel du constructeur A(x) sur le 16ème élément
A *a16 = new(&mem[15]) A(x);
...
// appel du destructeur sur ce même élément
a16->~A();
// il est possible de réutiliser cet emplacement
...
// l'emplacement mémoire reste alloué tant que l'on
// ne libère pas le bloc avec:
::operator delete(mem);
```

L'approche est un peu différente si des alignements spécifiques sont requis (exemple : alignement

SSE, ligne de cache ou page de la mémoire virtuelle) :

- en C, utiliser `void *aligned_alloc(size_t alignment,size_t size)` (libération avec `free`), où `size` est un multiple de `alignment`.
- en $C_{11}^{++}$, il est possible de définir un type qui respecte les règles d'alignement souhaité avec `std::aligned_storage`.
  Utiliser alors les fonctions d'allocation standard sur un objet de ce type.

**Exemple :**

```
// allocation
void *data = (void *)aligned_alloc(alignment,blocksize);
// desallocation standard avec free(mem)
```

```
// définition du type T avec l'alignement souhaité
typename std::aligned_storage<sizeof(T),alignof(T)>::type *data;
// allocation/desallocation standard avec ce type.
```

**Note :** La fonction `size_t alignof(Type)` permet de retourner l'alignement nécessité par un type `Type`.

# 10 Operateur new/delete

`new`/`delete` sont également des opérateurs de classe. A savoir, pour une classe `T`, la surcharge de l'opérateur :

- `void * operator new (size_t size)` permet d'allouer dynamiquement la mémoire d'un objet de type `T` (`size=sizeof(T)`)
- `void operator delete (void* ptr)` permet de libérer l'emplacement mémoire alloué dynamiquement pour un objet de type `T`.

**Quel est l'intérêt d'un telle surcharge ?**

- surcharger ces opérateurs permet d'utiliser un allocateur particulier afin de gérer les **allocations dynamiques** dans la classe surchargée,
  permet par exemple d'utiliser un zone de mémoire préallouée et de recycler les objets désallouées.
- les allocation dans la stack (variables locales) ne sont pas affectées par cette surcharge.

**Remarques :**

- utiliser l'opérateur de base `::new` et `::delete` dans les surcharges, sinon appel récursif (appel de l'opérateur en cours de définition).
- Attention, ces opérateurs sont hérités, d'où l'importance de surcharger ces opérateurs si on hérite d'une classe qui les a surchargé, et de faire un `assert` dans le `new` sur la taille à allouer afin de s'assurer que l'allocateur utilisé par défaut reste adéquat en cas de surcharge.

**Exemple:**

```
// surcharge inutile
struct A {
   int a;
   A(int x) : a(x) {}
   void * operator new (size_t size) {
      assert(size == sizeof(A));
      return ::operator new(sizeof(A));
   }
   void operator delete (void* ptr) {
      if (mem) ::operator delete(ptr);
   }
};

void exec() {
   A  x;        // pas d'appel à la surcharge
   A  *a = new A(4);  // appel à A::new
   delete a;      //
}
```

L'intérêt de cette approche est qu'elle permet d'intégrer la gestion mémoire de l'objet dans la définition de la classe, et de la rendre transparente à l'utilisateur.

Pour le cas où il est souhaitable d'utiliser des gestions mémoire différentes sur un même d'objet, voir la gestion d'un allocateur comme paramètre d'un patron de classe (c'est ce qui est fait dans la STL).

Les opérateurs globaux `::new`/`::delete` peuvent également être surchargés.

**Conséquences :** toutes les allocations dynamiques utiliseront ces opérateurs, et la surcharge devient alors la méthode d'allocation dynamique par défaut.

**Utilisations possibles :**

- utilisation d'allocateurs spécifiques sur la totalité de l'application,
- instrumentation des allocateurs pour trace ou débugging.

**Exemple :**

```
#define new new(__FILE__, __LINE__)
TrackerClass  MemoryTracker;   // classe à écrire
void * operator new (size_t size, char const * file, int line) {
    void * p = malloc (size);
    MemoryTracker.Add(p, file, line);
    return p;
}
void operator delete (void * p) {
    MemoryTracker.Remove (p);
    free (p);
}
// MemoryTracker.dump() par exemple pour dump mémoire
```

**Remarque :** dans ce cas, il n'est donc évidemment pas possible de faire appel à `new`/`delete`. On utilise alors les fonctions bas-niveau `malloc`/`free` du `C`.

L'équivalent de la méthode `realloc` n'existe pas en `C++`.

Sur le fond, cette méthode peut sembler une bonne idée. Elle permet :
- d'augmenter la taille du tableau sans le réallouer,
- d'éviter d'avoir à recopier le contenu actuel du tableau.

En pratique :
- si on prend des motifs d'allocation de taille aléatoire, les réallocations s'avèrent rares. autrement dit, les allocations/desallocations, faites par un allocateur que l'on ne maitrise pas, doivent finir par engendrer un motif de désallocation que permet régulièrement une réallocation plutôt qu'une allocation/copie.
- fausse bonne idée : ne garantit pas que le copie ne se fera pas
  - ◇ si la taille finale du tableau peut être évaluée, le faire.
  - ◇ si la taille d'un tableau peut beaucoup augmenter s'orienter plutôt vers des structure de données de type `deque` (cf STL), où l'augmentation de la taille du tableau s'effectue toujours sans copie,
  - ◇ sinon, un tableau n'est peut-être pas la structure appropriée,
  - ◇ n'économise pas de penser à un gestionnaire de mémoire spécialisé

# 11 Allocateurs

**Définition :** un allocateur est un objet :
- destiné à allouer et libérer de la mémoire,
- auquel la gestion de la mémoire est déléguée.

En C++, un allocateur doit implémenter essentiellement les 4 méthodes suivantes :
- `allocate<T>` : alloue de la place de stockage pour n objets de type `T` (sans constructeur),
- `deallocate<T>` : désalloue une place de stockage précédemment allouée.
- `construct<T>` : construction en place d'un objet dans un espace déjà alloué.
- `destroy<T>` : destruction en place d'un objet (sans désallouer l'espace qu'il occupe).

L'allocateur par défaut (`std::allocator`) revient à utiliser `new`/`delete` d'une manière thread-safe.

Or, une bonne stratégie de gestion mémoire est toujours :
- **compacité :** limiter le nombre d'allocation/desallocation.
- **localité :** allouer à proximité les objets utilisés en même temps.

Mais, si l'on construit un conteneur comme une liste chainée, l'allocateur standard n'assure ni la localité ni la compacité des allocations.

Dans ce type de cas, il est donc préférable de disposer d'une allocation par "pool" de mémoire, à savoir :
- l'allocateur `A` alloue un bloc mémoire (contigu) permettant de stocker *n* objets de type `T` (= pool de mémoire de `A`),
- si un allocateur `A` est associé à conteneur de `T`, alors toute allocation/libération mémoire du conteneur s'effectuera dans le pool de mémoire détenu par `A`,
- lorsque le pool est plein, l'implémentation de l'allocateur peut prévoir d'accroitre la taille de son pool (en allouant un nouveau bloc) où de renvoyer `std::bad_alloc`.

Dans le cas d'une liste chainée, on s'assure donc que :
- les éléments de la liste sont localisés dans le pool mémoire détenu par l'allocateur,

- si l'insertion/suppression dans le conteneur est local à un thread, les verrous systèmes ne sont pas nécessaires.

Le gain de temps peut donc être VRAIMENT considérable.

Une implémentation fiable et rapide d'un allocator n'allouant que des objets de même taille et compatible avec la STL peut être trouvée dans `boost` :

- `fast_pool_allocator` : allocation `T` par `T` (exemple : `list`),
- `pool_allocator` : allocation de paquets de `T` (exemple : `array`)

**Attention :** l'implémentation est statique (singleton) fait l'allocateur commun pour tous les types de même `sizeof` et non un conteneur en particulier.

**Exemple :**

```
{
   std::vector<int, boost::pool_allocator<int> > v;
   for (int i = 0; i < 10000; ++i) v.push_back(13);

}  // v désalloué, mais pool mémoire restant alloué

// libération du pool de mémoire
boost::singleton_pool<boost::pool_allocator_tag,
                sizeof(int)>::release_memory();
```

Voir la documentation de `boost` pour les détails.

`boost::pool` ou `boost::object_pool` fournit l'implémentation d'un pool de mémoire utilisable dans les autres cas.

**Exemple de performance :**

```
/*répétition*/ T* p = (T*)malloc(sizeof(T)); free(p);
```

```
/*répétition*/ T* p = new T; delete p;
```

```
boost::object_pool<T> pool;
/*répétition*/ data* p = pool.malloc(); pool.free(p);
```

```
boost::object_pool<T>* pool = new boost::object_pool<T>;
/*répétition*/ T* p = pool->malloc(); pool->free(p);
delete pool;
```

```
boost::object_pool<T> pool;
/*répétition*/ T* p = pool.construct(); pool.destroy(p);
```

Mesurée sur 10M d'objets : VS2015 (VC 14.0), boost 1.59

| Test | VC `malloc` `free` | VC `new` `delete` | Boost `malloc` `free` | Boost* `malloc` `free` | Boost `new` `delete` |
|---|---|---|---|---|---|
| Temps | 1.654s | 1.837s | 0.059s | 0.101s | 0.092s |
| Accélération | ×1 | ×1 | ×27.8 | ×18.2 | ×17.9 |

cas Boost* : attention aux indirections !

# Conclusion

Dans ce chapitre, nous avons vu :

- La façon dont la mémoire est structuré sur un ordinateur,
- Les différences d'utilisation entre pointeur et référence,
- L'allocation dynamique d'objet ou de tableau en C++,
- La conception de structures contenant des pointeurs,
- Les pointeurs intelligents qui permettent de gérer automatiquement les desallocations des ressources mémoire en respectant la sémantique du pointeur (unique, partagé, faiblement partagé),
- Les deux idiomes courants copy-and-swap et pImpl,
- L'allocation en place qui permet de gérer de façon séparée l'allocation mémoire d'un objet et de sa construction/destruction,
- la surcharge des opérateurs `new`/`delete` afin de définir des gestions mémoires particulières sur des classes d'objets.
- les allocateurs qui permettent des définir des classes de gestions mémoire particulières, avec l'exemple des allocateurs de `boost`,
- les indirections multiples afin de gérer les tableaux multi-dimensionnels (annexe 2),
- un rappel de l'utilisation de base d'un deboggueur (annexe 3).

Il est à noter qu'il est absolument essentiel d'apprendre à utiliser les pointeurs intelligents.

# 12 Annexe 1 : erreurs courantes de manipulation des pointeurs

Cette section contient des exemples élémentaires pour ceux toujours inconfortables avec la notion de pointeur. Elle reprend les opérateurs de base, et explique avec des exemples concrets l'effet exact de ces opérateurs dans la mémoire de l'ordinateur.

Veuillez noter que l'ensemble de ces exemples ne considèrent pas le fonctionnement interne du processeur (passages par les caches et registres) ni les situations de concurrence (multi-threading), mais seulement le résultat habituel et concret des l'opérateurs mémoires dans ce cadre.

Par ailleurs, dans les exemples qui suivent, les adresses mémoires sont notées en hexadécimal sous la forme `0x1234`. Ces adresses sont également 16bit par souci de concision, au lieu des 32 ou 64bit habituels.

## 12.1 Table des symboles

La table des symboles est une table qui stocke l'ensemble des variables locales allouées. Elle contient essentiellement, pour chaque symbole, son nom, son type et son adresse mémoire.

**Exemple :**

**Code :**

```
int   a=241;
float b=3.21f;
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| a | int | 0xAC00 |
| b | float | 0xAC04 |

**Mémoire :**

| | |
|---|---|
| | ⋮ |
| 0xAC00 | 241 |
| 0xAC04 | 3.21f |
| | ⋮ |

Cette table est implicitement construite lors de la compilation du code, et permet lors du déboguage de connaître à partir du symbole la valeur et l'adresse de la variable.

Noter que cette table n'existe plus pour un code optimisée pour lequel les optimisations mémoires peuvent faire disparaître certaines variables, et réutiliser les registres.

Elle permet également de rendre explicite la notion de pointeur. Nous l'utiliserons souvent dans cette leçon.

## 12.2 Pointeur simple

### a) Principes

**Définition :** un pointeur sur un type `T` est une variable dont la valeur est une adresse mémoire à laquelle se trouve une valeur de type `T`.

**Nouveau type :** le type d'un pointeur sur une case mémoire contenant un type `T` s'écrit `T*`.

**Exemple :** soit *a* un entier égal à 241. Soit *b* un pointeur sur l'entier *a*. Donc, *b* est de type `int*` et sa valeur est le numéro de la case mémoire qui stocke *a*.

| nom | type | adresse |
|---|---|---|
| *a* | int | 0xAC00 |
| *b* | int* | 0xAC04 |

| | |
|---|---|
| 0xAC00 | 241 |
| 0xAC04 | 0xAC00 |

**Important :** un pointeur ne stocke qu'une adresse, et elle doit toujours contenir l'adresse d'une zone mémoire allouée (= pointer sur une case mémoire allouée).

### b) Opérateurs

Les opérateurs de base pour manipuler les pointeurs sont les suivants :

- **opérateur &**
  pour toute variable, permet d'obtenir l'adresse d'une variable (à savoir, l'endroit où sa valeur est stockée dans la mémoire).
  **Exemple :**
  ```
  int a = 3;
  // b contient l'adresse de a
  int *b = &a;
  ```
- **opérateur ***
  pour un pointeur, permet d'obtenir la valeur stockée à l'adresse sur laquelle il pointe.
  **Exemple :**
  ```
  // *b = valeur pointée par b
  int  c = *b + 1;
  ```

On a alors deux cas :

**cas 1 :** si a est une variable de type T qui n'est pas un pointeur (types élémentaires, structures, objets, ...)

- &a est l'adresse de cette variable ; son type est T*.
- *a n'a pas de sens.

**Exemple :**

**Code :**

```
int  u=5;
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| u | int | 0xBC80 |

**Mémoire :**

| | |
|---|---|
| 0xBC80 | 5 |

alors :

- u a pour valeur 5, son type est int.
- &u a pour valeur 0xBC80, son type est int*.
- *u n'a pas de sens.

**cas 2 :** si a est une variable de type T*

- &a est l'adresse de ce pointeur (i.e. l'adresse de la case mémoire qui stocke l'adresse) ; son type est T**. On verra plus tard à quoi peut servir un double pointeur.
- *a est la valeur contenue dans la case mémoire dont l'adresse est a ; son type est T.

**Exemple :**

**Code :**

```
int   u = 5;
int   *v = &u;
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| u | int | 0xBC80 |
| v | int* | 0xBC84 |

**Mémoire :**

| | |
|---|---|
| 0xBC80 | 5 |
| 0xBC84 | 0xBC80 |

alors :

- v a pour valeur 0xBC80, son type est int*.
- &v a pour valeur 0xBC84, son type est int**.
- *v a pour valeur 5, son type est int.

**Exemple :**

**Code :**

```
int  a = 3;
int  *b = &a;
int  *c = b;
int  **d = &b;
a = a + 1;
*b = *b + 1;
*c = *c + 1;
**d = **d + 1;
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| a | int | 0xBC80 |
| b | int* | 0xBC84 |
| c | int* | 0xBC88 |
| d | int** | 0xBC8C |

**Mémoire :**

| | ligne 4 | ligne 5 | ligne 6 | ligne 7 | ligne 8 |
|---|---|---|---|---|---|
| 0xBC80 | 3 | 4 | 5 | 6 | 7 |
| 0xBC84 | 0xBC80 | 0xBC80 | 0xBC80 | 0xBC80 | 0xBC80 |
| 0xBC88 | 0xBC80 | 0xBC80 | 0xBC80 | 0xBC80 | 0xBC80 |
| 0xBC8C | 0xBC84 | 0xBC84 | 0xBC84 | 0xBC84 | 0xBC84 |

**Valeurs pointées :**

| | ligne 4 | ligne 5 | ligne 6 | ligne 7 | ligne 8 |
|---|---|---|---|---|---|
| *b | 3 | 4 | 5 | 6 | 7 |
| *c | 3 | 4 | 5 | 6 | 7 |
| *d | 0xBC80 | 0xBC80 | 0xBC80 | 0xBC80 | 0xBC80 |
| **d | 3 | 4 | 5 | 6 | 7 |

## 12.3 Références

Une référence (ou alias) est une façon de donner un autre nom à un même symbole.

**Nouveau type :** le type d'une référence à un symbole de type `T` s'écrit `T&` (se dit également référence à une lvalue).

**Exemple :**

**Code :**

```
int   u = 5;
int   &v = u;
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| u | int | 0xBC80 |
| v | int | 0xBC80 |

**Mémoire :**

| 0xBC80 | 5 |
|---|---|

La table des symboles montre clairement que `u` et `v` font références à la même case mémoire (et donc sont deux noms identiques pour la même valeur).

**Notes :**

- les références sont uniquement utilisables en C++,
- une référence & peut uniquement faire référence à une variable qui peut être nommée, et non à une constante numérique ou à une expression temporaire (exemple : `int &v = 4` ou `4+u` n'est pas possible).
- le modificateur `const` (pour faire une référence constante) lève cette restriction (exemple : `const int &v =4+u` est possible).

## 12.4 Règles à respecter

Les règles de bases suivantes doivent toujours être respectées :

- un pointeur doit pointer sur de la mémoire allouée pour pouvoir être utilisé.
- l'accès aux éléments d'un tableau doit se faire dans la limite de ses bornes.
- toute valeur lue dans un tableau doit avoir été au préalable écrite au moins une fois.
- tout tableau dynamique alloué doit être désalloué.
- conserver toujours un pointeur sur le début du tableau,
  si la mémoire n'est pas libérée dans la même fonction où elle est allouée, celui-ci doit être transmis, sinon il est impossible de libérer la mémoire allouée.

**Fréquences et allocateur :**

- L'allocation mémoire est une opération (système) lente :
  - ◇ ne jamais faire d'allocation dans des boucles.
  - ◇ limiter le nombre d'allocations.
    ne pas passer son temps à réserver/libérer.
    ne pas faire d'allocation pendant les parties intensives du code (allouer avant l'espace nécessaire, libérer ensuite).
  - ◇ réserver par bloc

  L'impact peut être conséquent (30% déjà observé).
- Allocateur : réservation par blocs
  - ◇ il est préférable d'allouer plusieurs blocs en une seule fois, et d'utiliser plusieurs pointeurs sur le bloc alloué.
  - ◇ la généralisation de l'allocation par bloc conduit à la notion d'allocateur (gestionnaire de plusieurs tableaux dans un seul bloc mémoire).
- Il est préférable d'allouer ensemble les blocs utilisés ensembles.

Le non-respect peut engendrer de la fragmentation mémoire.

## 12.5 Erreurs de base

Pour être utilisé, tout pointeur doit pointer sur une zone allouée.

**Exemple 1 :** utilisation d'un pointeur invalide

```
int *u;
*u  = 4:     // erreur (non initialisé)
int *v = NULL;
*v = 4;      // erreur (pointeur NULL)
v = 0x12345678;
*v = 12;     // erreur (mauvaise adresse)
```

**Exemple 2 :** utilisation d'un pointeur sur une zone desallouée.

```
int *v = new int[2];
*v = 1:       // ok;
delete [] v;      // faire v=NULL
*v = *v + 1;  // erreur (mais ne plante pas)
```

**Variation :** l'allocation statique est limitée à la portée de la variable. Ne jamais conserver un pointeur vers une zone mémoire locale liberée.

**Exemple 3 :** utilisation d'un pointeur invalide

```
int *v = NULL;
{
    int a=3;
    v = &a;
}   // a libéré ici
*v = 6; // erreur
```

Même erreur si v pointe sur une variable locale dans une fonction.

**Exemple 4 :** retour d'un pointeur vers un tableau statique

```
int *AllocVector4(void) {
    int v[4]; // variable locale
    return v;   // pointeur vers mémoire locale
}
// erreur: v libéré au retour de la fonction
```

**Solution pour l'exemple 4 :** faire une allocation dynamique

```
int *AllocVector4(void) {
    return new int[4];
}
```

**L'adresse d'un bloc alloué dynamiquement doit être conservée pour pouvoir être libéré.**

**Exemple 5 :** perte de l'adresse d'un tableau

```
int *v=NULL;
v = new int[4]; // zone mémoire 1
v = new int[10];  // zone mémoire 2
// impossible de libérer ici la zone mémoire 1
...
delete [] v;  // libération zone mémoire 2
```

**Exemple 6 :** ne jamais renvoyer la référence d'un pointeur alloué déréférencé

**Exemple :**

```
A& fun() {
    A *a = new A();
    return *a;
}
```

Ce genre de construction pose deux problèmes :

- il n'est pas naturel de libérer un objet déréférencé (*i.e.* de faire `delete &a`).
- si cette fonction est chaînée avec d'autres fonctions, alors l'objet sera presque certainement perdu.
  **Exemple :** `fun4(a,b, fun2(c, fun()));`
  ce qui sera le cas si `fun2` est un opérateur.

## 12.6 Erreur courante

**Attention** aux objets qui contiennent des pointeurs vers des ressources

- les pointeurs d'une structure doivent toujours être initialisés (au moins à `nullptr`).
- lors de la copie d'une telle structure, la copie avec l'opérateur = ne permet d'obtenir le résultat recherché.
- lors de la désallocation d'une structure, penser à libérer la partie dynamique de la structure (à savoir, la mémoire dynamique allouée utilisée seulement par la structure).

**Exemple :**

Pour la structure suivante :

```
struct Vector { int nb; float *vec; };
```

Soit la fonction d'initialisation :

```
Vector InitVector(int n) {
   Vector V = { n, new float[n] };
   memset(V.vec,0,n*sizeof(float));
   return V;
}
```

On considère maintenant le code suivant :

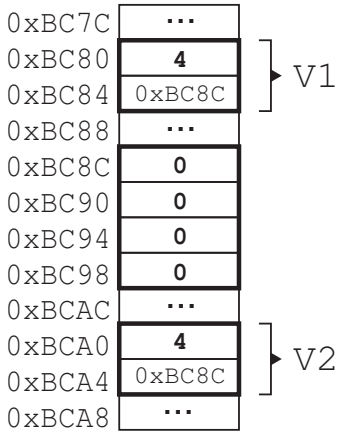
```
Vector V1 = InitVector(3);
Vector V2 = V1;
```

Pourquoi `V1` et `V2` ne sont pas des vecteurs indépendants ?

Les vecteurs ne sont pas indépendants au sens suivant : les données du vecteur `V2` utilisent exactement la même table que le vecteur `V1`.

En conséquence, si le contenu du vecteur `V1` est modifié, alors le contenu du vecteur `V2` l'est aussi.

**Raison :** l'opérateur = ne copie que les champs, et non les données.




Il faudrait écrire une fonction de copie explicite :

```
Vector CopyVector(const Vector &Vi) {
   Vector Vo = {Vi.n, new float[Vi.n]};
   memcpy(Vo.vec,Vi.vec,Vi.n * sizeof(float));
   return Vo;
}
```

Considérons le code suivant :

```
void fun1() {
   Vector  V=InitVector(3);
   ...
   // fin de portée de V
}
```

Si rien n'est fait, la mémoire dynamique de la structure n'est pas libérée (et donc perdue).

**Conséquence :** elle doit être libérée explicitement. On propose à cet effet la fonction suivante.

```
void DeleteVector(Vector &V) {
   // libération partie dynamique du vector
   delete [] V.v;
}
```

Le code précédent doit donc être corrigé sous la forme :

```
void fun1() {
   Vector  V=InitVector(3);
   ...
   DeleteVector(V);
}
```

# 13 Annexe 2 : niveaux d'indirection

Une indirection est l'accès à une case mémoire à travers un pointeur.

**Par l'exemple :**

```
int   a=4;
int   *b = &a;     // une indirection
int   **c = &b;    // deux indirections
int   ***d = &c;   // trois indirections
```

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| a | int | 0x18E0 |
| b | int* | 0x18E4 |
| c | int** | 0x18E8 |
| d | int*** | 0x18EC |

**Mémoire :**

| | |
|---|---|
| 0x18E0 | 4 |
| 0x18E4 | 0x18E0 |
| 0x18E8 | 0x18E4 |
| 0x18EC | 0x18E8 |

**Accès à la valeur pointée :**

| | & | valeur | * | ** | *** | **** |
|---|---|---|---|---|---|---|
| a | 0x18E0 | 4 | n/a | n/a | n/a | n/a |
| b | 0x18E4 | 0x18E0 | 4 | n/a | n/a | n/a |
| c | 0x18E8 | 0x18E4 | 0x18E0 | 4 | n/a | n/a |
| d | 0x18EC | 0x18E8 | 0x18E4 | 0x18E0 | 4 | n/a |

D'accord mais à quoi cela sert ?

## 13.1 Principe

**Par l'exemple :**

- un tableau d'entiers (par exemple, un vecteur) est de type `int*`.
- un tableau de vecteurs (par exemple, une matrice) est de type `int**`.
- une tableau de matrice est de type `int***`.
- ...

Sens de l'opérateur `[]` sur ces tableaux :

- un tableau d'entiers (`int*`)
  `T[i]` = `i`$^{\text{ème}}$ valeur du tableau (`int`).
- un tableau de vecteurs (`int**`)
  `T[i]` = `i`$^{\text{ème}}$ vecteur (`int*`).
  `T[i][j]` = `j`$^{\text{ème}}$ valeur du `i`$^{\text{ème}}$ vecteur (`int`).
- un tableau de matrices (`int***`)
  `T[i]` = `i`$^{\text{ème}}$ matrice (`int**`).
  `T[i][j]` = `j`$^{\text{ème}}$ vecteur de la `i`$^{\text{ème}}$ matrice (`int*`).
  `T[i][j][k]` = `k`$^{\text{ème}}$ valeur du `j`$^{\text{ème}}$ vecteur de la `i`$^{\text{ème}}$ matrice (`int`).

Ce sens sera précisé par la suite.

**Exemple 1 :** Comment faire pour allouer un tableau de vecteurs ?

De quoi a-t-on besoin pour construire un tableau de `n` vecteurs de taille `p` ?

- un tableau de `n` vecteurs est donc un tableau de `n` `int*`.
- pour stocker les données de chaque vecteur, on a besoin de `p` `int`. Donc, au total, on a besoin de `n` $\times$ `p` `int`.
- chaque vecteur doit pointer vers l'endroit où sont stockés ses données.

Pour un tableau `t` de vecteurs d'entiers (de 2 vecteurs de 3 entiers), stockant la matrice $\left(\begin{smallmatrix}1&2&3\\4&5&6\end{smallmatrix}\right)$, le stockage est le suivant :

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| t | int** | 0x1810 |

**Mémoire :**

| | | |
|---|---|---|
| 0x1810 | 0x1944 | matrice t |
| ... | ... | |
| 0x1944 | 0x1960 | vecteur t[0] |
| 0x1948 | 0x19B4 | vecteur t[1] |
| ... | ... | |
| 0x1960 | 1 | valeur t[0][0] |
| 0x1964 | 2 | valeur t[0][1] |
| 0x1968 | 3 | valeur t[0][2] |
| ... | ... | |
| 0x19B4 | 4 | valeur t[1][0] |
| 0x19B8 | 5 | valeur t[1][1] |
| 0x19BC | 6 | valeur t[1][2] |

Un code permettant d'allouer la matrice précédente est le suivant :

```
const int n=2, p=3;
int **t = NULL;
// allocation du tableau de n vecteurs
t = new int*[n];
if (t==NULL) { /* alloc error */ }
// allocation de chacun des n vecteurs
for(int i=0;i<n;i++) {
    t[i] = new int[p];
    if (t==NULL) { /* alloc error */ }
}
```

On libère la mémoire allouée avec :

```
for(int i=0;i<n;i++) delete [] t[i];
delete [] t;
```

**Problème :** cette méthode d'allocation viole au moins une des règles énoncées pour l'allocation mémoire (allocation dans une boucle, fragmentation mémoire)

**Solution :** On alloue les données dans un seul vecteur de taille `n` $\times$ `p`.

```
const int n=2, p=3;
int **t = NULL;
// allocation du tableau de n vecteurs
t = new int*[n];
if (t==NULL) { /* alloc error */ }
// allocation de n*p données
t[0] = new int[n*p];
if (t[0]==NULL) { /* alloc error */ }
// pointer les autres vecteurs sur les données
for(int i=1;i<n;i++) t[i] = t[i-1] + p;
```

On rappelle l'algorithmique des pointeurs, ajouter `p` à un pointeur de type `T` incrémente son adresse de `p*sizeof(T)`.

Libération de la mémoire allouée :

```
delete [] t[0];    // libération données
delete [] t;       // libération pointeurs de ligne
```

La mémoire allouée par la seconde méthode d'allocation pour un tableau `t` de vecteurs d'entiers

(de 2 vecteurs de 3 entiers), stockant la matrice $\left(\begin{smallmatrix}1&2&3\\4&5&6\end{smallmatrix}\right)$ est le suivant :

**Table des symboles :**

| nom | type | adresse |
|---|---|---|
| t | int** | 0x1810 |

**Mémoire :**

| | | |
|---|---|---|
| 0x1810 | 0x1944 | matrice t |
| ... | ... | |
| 0x1944 | 0x1960 | vecteur t[0] |
| 0x1948 | 0x196C | vecteur t[1] |
| ... | ... | |
| 0x1960 | 1 | valeur t[0][0] |
| 0x1964 | 2 | valeur t[0][1] |
| 0x1968 | 3 | valeur t[0][2] |
| 0x196C | 4 | valeur t[1][0] |
| 0x1970 | 5 | valeur t[1][1] |
| 0x1974 | 6 | valeur t[1][2] |

**Interprétation de l'opérateur `[]`**

soit un tableau `t` de type `int**` (donc un tableau de vecteurs).

- `t` = adresse du tableau de pointeurs de ligne (= pointe sur le premier pointeur de ligne)
- `t+i` = adresse du `i`ème pointeur de ligne.
- `*(t+i)` = valeur pointée par le `i`ème pointeur de ligne (= dans le tableau de données, adresse du premier élément de la `i`ème ligne).
- `*(t+i)+j` = adresse du $j^{\text{ème}}$ élément de la `i`ème ligne.
- `*(*(t+i)+j)` = valeur du $j^{\text{ème}}$ élément de la `i`ème ligne.

**Exemple :** en reprenant la matrice $2 \times 3$ précédente :

| Expression | Valeur |
|---|---|
| t | 0x1944 |
| t+1 | 0x1948 |
| *(t+1) | 0x196C |
| *(t+1)+2 | 0x1974 |
| *(*(t+1)+2) | 6 |

| | |
|---|---|
| 0x1810 | 0x1944 |
| ... | ... |
| 0x1944 | 0x1960 |
| 0x1948 | 0x196C |
| ... | ... |
| 0x1960 | 1 |
| 0x1964 | 2 |
| 0x1968 | 3 |
| 0x196C | 4 |
| 0x1970 | 5 |
| 0x1974 | 6 |

## 13.2 Applications

Le principe est le même pour les structures régulières de dimension $p$.

Par la suite, on notera `T`$*^i$ pour un pointeur de type `T` avec $i$ indirections, et `t`$[0]^i$ pour l'accès à l'élément 0.

**Exemple :** `T`$*^4$ = `T` $* * * *$.

Allocation d'un objet stockant des données de type $T$ de taille $n_1 \times n_2 \times \cdots \times n_p$ :

- un vecteur $t_p$ de $n_1$ `T`$*^{p-1}$.

- un vecteur $t_{p-1}$ de $n_1 \times n_2$ T$*^{p-2}$.
  le $t_p[i]$ pointant sur $t_{p-1} + i.n_2$, pour $i \in \{0, \ldots, n_1 - 1\}$
- un vecteur $t_{p-2}$ de $n_1 \times n_2 \times n_3$ T$*^{p-3}$.
  le $t_{p-1}[i]$ pointant sur $t_{p-2} + i.n_3$, pour $i \in \{0, \ldots, n_1 \times n_2 - 1\}$
- ...
- un vecteur $t_1$ de $n_1 \times \cdots \times n_p$ T.
  le $t_2[i]$ pointant sur $t_1 + i.n_1$, pour $i \in \left\{0, \ldots, n_1 \times \cdots \times n_{p-1} - 1\right\}$

La libération se fait dans le sens inverse : $t_1$, puis $t_2$, ..., $t_p$.

## 13.3 Alternative aux indirections

L'alternative aux indirections pour stocker une structure de dimension $p$ consiste à :

- n'allouer que les données nécessaires au stockage sous forme d'un seul table (mono-dimensionnel)
  les lignes d'une matrice sont stockées les unes après les autres, ...
  les matrices d'un tableau de matrices sont stockées les unes après les autres ...
- calculer pour les coordonnées souhaitées $(i_1, i_2, \ldots, i_p)$ le décalage dans le tableau mono-dimensionnel afin d'atteindre la donnée souhaitée.

**Exemples :**

$M_1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ stocké dans $T_1 =$ | 1 | 2 | 3 | 4 | 5 | 6 |

$M_1(i, j) = T_1[i * 3 + j]$

$M_2 = \left\{ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \right\}$ stocké dans $T_2 =$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

$M_2(i, j, k) = T_2[i * 4 + j * 2 + i]$

Une indirection est alors remplacée par une multiplication et une addition.

## 13.4 Limites des indirections

Pour un stockage d'un tableau de dimension $p$ de taille $n_1 \times n_2 \times \cdots \times n_p$. Pour simplifier les calculs, on supposera que $\forall i, n_i = n$.

- **avec des indirections :**
  Données = $n^p$, Pointeur = $1 + n + n^2 + \cdots + n^{p-1} = \frac{n^p - 1}{n-1}$
  Accès = $p$ indirections (= $p$ additions + $p$ déréférencements).
- **avec un tableau unidimensionnel :**
  Données = $n^p$, Pointeur = 1
  Accès = $p$ multiplications + $p$ additions + 1 déréférencement.

Autrement dit,

- le stockage avec des indirections peut se révéler prohibitif.
- le stockage unidimensionnel peut induire un coût de calcul important de l'adresse, mais qui peut être réduit par le compilateur (partage entre les différents accès pour des accès proches).

## 13.5 Cas statique

**Rappel :** un tableau multidimensionnel statique est un tableau dont la taille est fixe et déterminée à la déclaration.

**Exemple :** `int a[4][4];`

**Règles :**

- Tous les tableaux multidimensionnels sont gérés comme des tableaux unidimensionnels.
- Tous les calculs des indices à partir de la taille du tableau sont placés en dur dans le code pour fin d'optimisation.
- Le type `T[N][P]` n'est compatible par défaut qu'avec des types `T[M][P]` où `N` et `M` peuvent être différents.
  Les types `T*` et `T**` ne sont pas compatibles.
  sauf à forcer explicitement la conversion (casting).

**Initialisation :** `int a[2][3] = {{2,4,6},{1,3,5}};`

## 13.6 Règles et erreurs

Les règles de bases sont les mêmes que pour un tableau dynamique avec un seul niveau d'indirection.

Néanmoins, on insistera sur les points suivants :

- l'allocation mémoire étant une opération lente, le nombre d'allocations doit être limité.
- si les bornes des indices ne sont pas respectés, on risque :
  - ◇ soit d'écrire à l'extérieur de la zone allouée (débordement)
  - ◇ soit d'écrire ailleurs dans la zone allouée. Par exemple, pour une matrice $M$ de taille $4 \times 4$, l'écriture dans $M[1][7]$ ($2^{\text{ème}}$ ligne, $8^{\text{ème}}$ colonne) écrit en fait dans $M[2][3]$ ($1 \times 4 + 7 = 2 \times 4 + 3$).
- pour les tableaux de grande taille, faire en sorte qu'ils ne soient initialisés qu'une seule fois.
  **Exemple :** initialiser systématiquement un tableau à 0 peut sembler une bonne idée, mais les initialisations inutiles peuvent faire perdre un temps très important.

# 14 Annexe 3 : Débogueur

Un débogueur est un programme permettant :

- d'arrêter l'exécution en n'importe quel point du code.
- d'afficher les valeurs des symboles du code, ou de contenu de la mémoire.
- d'afficher et se déplacer dans la pile d'appel des fonctions pour consulter l'environnement d'exécution de n'importe quelle fonction dans la pile.
- d'exécuter le code pas à pas.

C'est donc un outil permettant d'analyser l'état d'un programme à un moment quelconque de son exécution.

**Remarque :** c'est un outil d'aide complémentaire aux traces et souvent nécessaire pour analyser, comprendre et trouver rapidement l'origine de bogues.

Le débogueur le plus répandu et toujours disponible sur une plateforme Unix s'appelle `gdb` (acronyme

de **G**NU **D**é**B**ogueur).

## 14.1 Utilisation de base

Il s'agit du cas où le programme se termine sur un `segmentation fault`. On suppose dans que l'exécutable construit se nomme `toto`.

L'utilisation de base (à savoir, l'analyse d'un code dont l'exécution conduit à un plantage) est la suivante :

1. compiler l'ensemble des `.c`/`.cpp` avec le flag `-g`
   ceci permet d'inclure les symboles aux fichiers compilés.
   **Comment faire :** ajouter `-g` au `CPPFLAGS`.
2. lancer `gdb` sur l'exécutable. A la suite de cette commande, on se trouve à l'invite de commande de `gdb`.
   **Comment faire :** `gdb ./toto`
3. lancer l'exécutable lui même avec `run` (si le programme a des arguments, les passer derrière `run`).
   **Comment faire : (gdb)** `run`
4. lorsque le programme s'arrête sur l'erreur, on reprend la main à l'invite de commande `gdb`. On peut maintenant taper des commandes sous `gdb` afin d'analyser l'état du programme au moment de l'erreur (voir ci-après).
5. sortir de `gdb`.
   **Comment faire : (gdb)** `quit`

## 14.2 Commandes

### a) Points d'arrêt

Un point d'arrêt est une façon de provoquer l'interruption de l'exécution d'un code à l'endroit où une condition est vérifiée.

Il y a deux types de point d'arrêt :

- **point d'arrêt du code :** placé sur une ligne du code, ou sur une fonction. Déclenché lorsque la ligne est atteinte ou lorsque la fonction est appelée.
- **point d'arrêt mémoire :** placé sur une adresse mémoire. Déclenché lorsque la valeur stockée à cette adresse mémoire est modifiée.

Sous gdb, les commandes pour manipuler les points d'arrêt (PA) sont les suivantes :

- `break fun` : place un PA au début de la fonction `fun`.
- `break num` : place un PA à la ligne `num` du code.
- `break *adr` : place un PA mémoire sur l'adresse `adr`.
- `info breakpoints` : affiche l'ensemble des PAs (avec leurs ids)
- `disable id` : désactive le PA numéro `id`.
- `enable id` : active le PA numéro `id`.
- `delete id` : supprime le PA numéro `id`.
- `clear fun` : efface tous les PAs dans la fonction `fun`.

- `delete` : efface tous les PAs.

**Note :**

`file:num` signifie la ligne `num` du fichier `num`.
`file:fun` signifie la fonction `fun` du fichier `file`.

### b) Commandes d'exécution

Lors de l'exécution manuelle, deux modes sont possibles :

- mode `step` : exécution ligne-à-ligne (LAL) de la fonction courante. Les appels de sous-fonctions ne sont pas détaillés (exécution et retour immédiats).
- mode `next` : exécution pas-à-pas (PAP) de la fonction avec entrée et exécution LAL des fonctions appelées.

Les commandes d'exécutions sont les suivantes :

- `run` : lance l'exécution du programme. Ne s'arrête que sur un PA ou sur une erreur d'exécution.
- `step` : exécution LAL.
- `step nlines` : exécution LAL de `nlines` lignes.
- `next` : exécution PAP.
- `next nline` : exécution PAP de `nlines` lignes.
- `until line` : exécution jusqu'à ce que la ligne `line` soit atteinte.
- `finish` : fini l'exécution de la fonction courante, revient.
- `continue` : reprend l'exécution (jusqu'au prochain PA).
- `continue n` : reprend l'exécution en ignorant les n PAs suivants.
- `return val` : retour immédiat de la fonction avec la valeur `val`.
- `quit` : sortir de `gdb`.

### c) Pile d'appel des fonctions

Sous `gdb`, la pile d'appel des fonctions peut être visualisée et manipulée.

- elle représente l'état exact dans lequel le programme se trouve à ce moment.
- elle contient la pile d'appel, numéroté de `0` (main, le sommet de la pile) à `n` (profondeur actuelle de la pile).
- on appelle `frame` les informations relatives à un appel de fonction. Elles contiennent les paramètres d'appel et les variables locales de la fonction.
- déplacement dans la pile d'appel :
  monter = aller dans la fonction appelante.
  descendre = aller dans la fonction appelée.

Les commandes suivantes permettent de visualiser et manipuler la pile d'appel des fonctions :

- `where` : affiche la pile d'appel des fonctions.
- `frame n` : va au niveau `n` de la pile.
- `up` : va dans la fonction appelante (monte d'un niveau dans la pile).
- `up n` : remonte de `n` niveau dans la pile.
- `down` : va dans la fonction appelée (descend d'un niveau dans la pile).
- `down n` : descend de `n` niveau dans la pile.

#### d) Affichage

**Commandes d'affichage du code :**

- `list n` : affiche le code à la ligne `n`
- `list fun` : affiche le code de la fonction `fun`
- `list` : affiche les 10 lignes suivantes.

**Commandes des valeurs d'une variable :**

- `print var` : affiche la variable `var`. `p` est un raccourci de `print`.
  **Exemple :** `print a`
- `print /switch var` : idem en forçant l'affichage dans un format particulier, où `switch` est x=hexadécimal, d=entier, u=entier non signé, t=binaire, a=adresse, c=caractère, f=flottant, s=chaîne.
  **Exemple :** `print /x a` (idem en hexadécimal).
- `x /NFU addr` : affiche à partir de l'adresse `addr`, où N est le nombre d'éléments à afficher, F est le format (comme ci-dessus), U=force la nombre d'octets pour le format à b=1, h=2, w=4, g=8 octets.
  **Exemple :** `x /10dw p` (pour afficher 10 entiers à partir de l'adresse p).

## 14.3 Exemples

- vérifier qu'une fonction `tata` s'exécute correctement :
  1. mettre un PA sur la fonction (`break tata`).
  2. lancer l'exécution (`run`).
  3. continuer l'exécution du programme (`step`/`print`).
- pour un point d'arrêt sur une condition particulière. Écrire le test vérifiant cette condition, et placer le point d'arrêt dans le test.
- un élément $a[12]$ d'un tableau change de valeur sans que je la change explicitement (effet de bord).
  1. mettre un PA au moment où le tableau $a$ est alloué, puis lancer l'exécution.
  2. mettre un PA mémoire, sur $a + 12$ (`break *(a+12)`).
  3. continuer l'exécution du programme (`continue`).

## 14.4 Fichier core

Lorsqu'un programme **compilé en mode debug** provoque un accès mémoire invalide, alors le programme se termine avec le message suivant :

```
segmentation fault
core dumped.
```

`core dumped` signifie qu'un fichier `core` a été généré. Celui-ci contient la pile d'appel des fonctions ainsi que l'ensemble de la mémoire allouée par le programme au moment de sa terminaison.

Ce fichier peut être utilisé `gdb` afin de recharger l'environnement du programme (nommé ici `toto`) au moment de sa terminaison de la façon suivante :