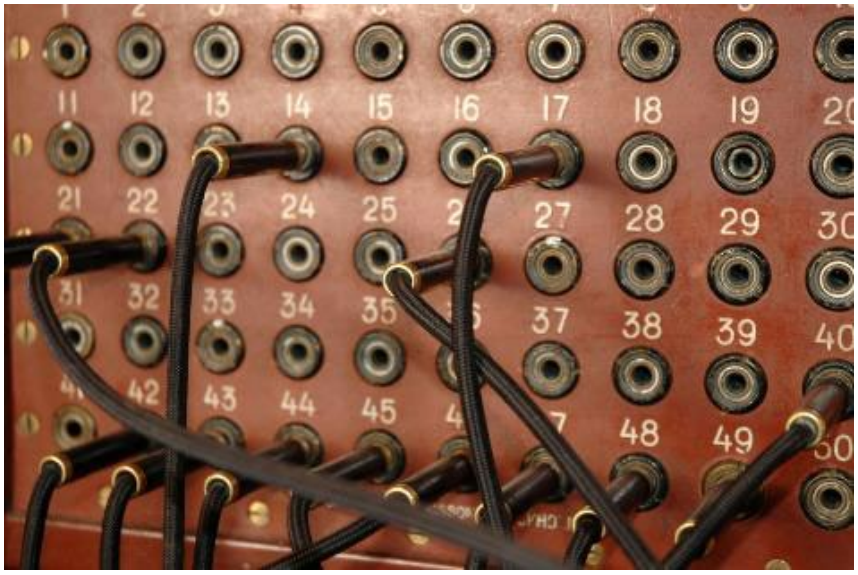


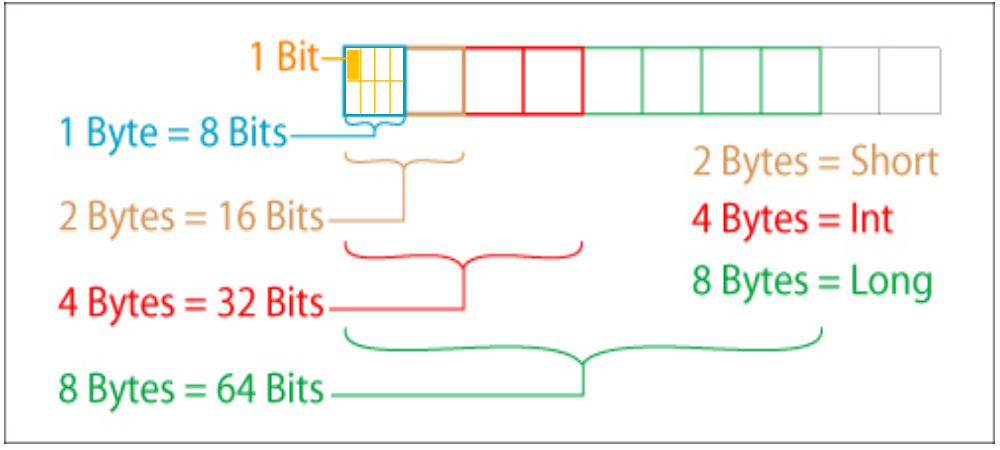
Chapitre 2
Pointeurs

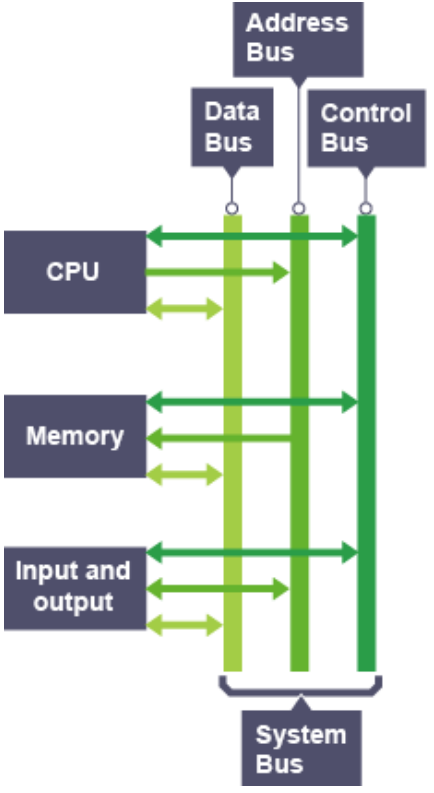
Plan du chapitre 2

1. Introduction

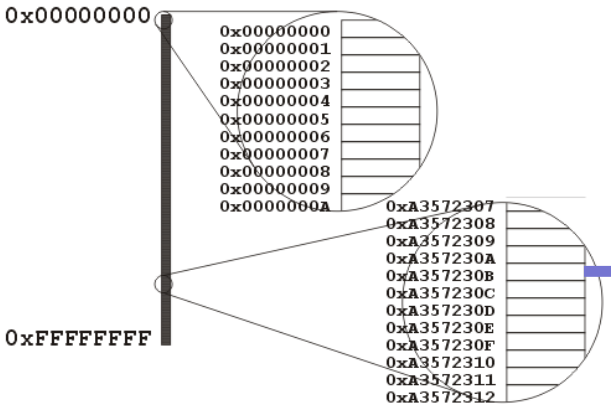
La mémoire d'un ordinateur

- Dans un ordinateur, la mémoire principale est divisée en octets (*bytes*)
- Chaque octet stocke huit bits d'information
- Chaque octet a une adresse unique



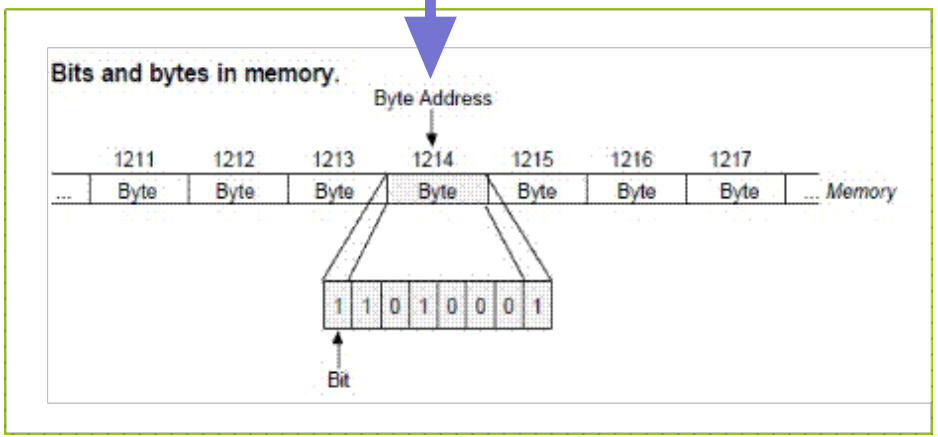


Autre vue de la mémoire



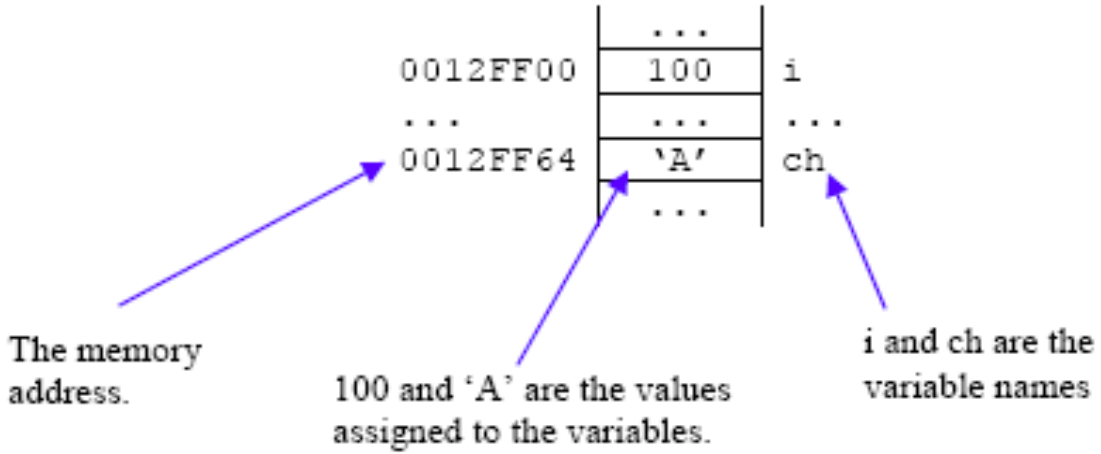
Adresses (0x0000.. à 0xFFFF..) et

emplacements (contenu = un octet)



Cas d'un langage de programmation

- Le contenu de chaque variable déclarée dans un programme occupe un ou plusieurs octets en mémoire
- Chaque variable est identifiée univoquement par son nom (identificateur)
- L'adresse d'une variable correspond à l'**adresse du premier octet** de la zone mémoire utilisée pour son stockage



Afficher l'adresse d'une variable en C

- Grâce à l'opérateur **&** et à une instruction de formatage spécifique de `printf`

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

int main(void) {
   char c = 'A';
   int  i = 100;
   printf("c vaut %c, adresse %p\n", c, (void*)&c);
   printf("i vaut %d, adresse %p\n", i, (void*)&i);
   return EXIT_SUCCESS;
}
```

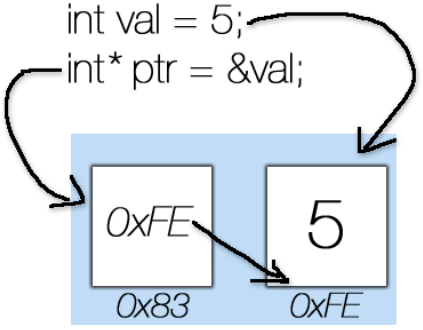
```
c vaut A, adresse 000000000061FE1F
i vaut 100, adresse 000000000061FE18
```

- %p : affiche une adresse (en hexadécimal)
- `(void*)` : convertit l'adresse d'un entier en une adresse générique
  (pas strictement nécessaire ici)

2. Pointeurs : définition

Définition d'un pointeur

- On peut stocker une adresse dans une variable
  - cette variable est d'un type spécifique, adapté au stockage des adresses : le **type pointeur**
  - en C, on doit également préciser le type de données stockées à l'adresse en question : `char`, `int`, etc.
- Lorsqu'on stocke l'adresse d'une variable `val` dans une variable de type pointeur `ptr`, on dit que « `ptr` pointe *vers* `val` » ou que « `ptr` pointe *sur* `val` »



Le type des pointeurs

- On doit indiquer vers quel type d'objet pointe une variable pointeur, en déclarant le pointeur ainsi : `type*`
  - un pointeur peut pointer vers des zones mémoire qui ne contiennent pas des variables (p. ex. des fonctions)
- En C, un pointeur ne peut pointer que vers des objets d'un type précis
  - on peut alors changer à volonté la référence du pointeur

```
int val = 13;
int* ptr1 = NULL;
int* ptr2 = NULL;
ptr1 = &val;
ptr2 = ptr1;
printf("ptr2 pointe vers %d", *ptr2);
```

```
ptr2 pointe vers 13
```

Déréférencement d'un pointeur

- Comment obtenir la valeur qui se trouve à l'adresse contenue dans un pointeur ?
  - que trouve-t-on à l'endroit de la mémoire vers lequel pointe une variable pointeur ?
  - que vaut l'objet référencé par le pointeur ?

```
int val = 13;
int* ptr1 = NULL;
int* ptr2 = NULL;
ptr1 = &val;
ptr2 = ptr1;
printf("ptr2 pointe vers %d", *ptr2);
```

Opérateur * :

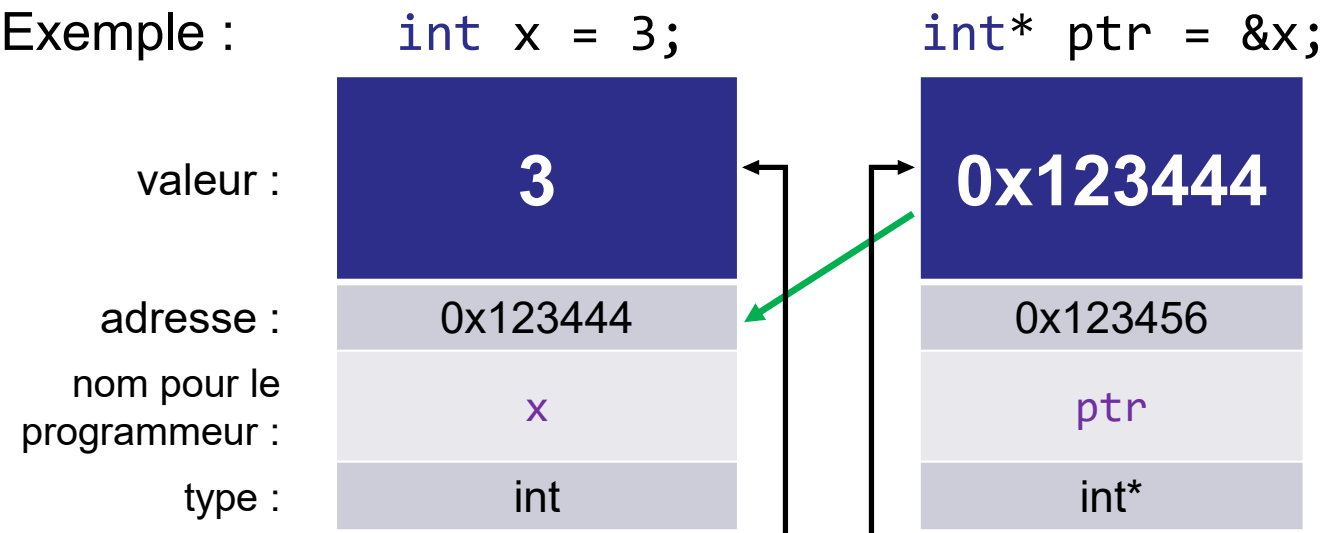
*ptr2 retourne la valeur pointée

Déclaration des pointeurs

- Utilisation de l'étoile (*)
  - déclaration du type du **pointeur** (= type de la **valeur pointée**)
  - préférer coller l'étoile au type : `int* ptr = NULL;`
    - C accepte aussi : `int *ptr` ou `int * ptr`
    - *Mais attention :* `int* p, q;` *déclare un pointeur et un entier !*
- **Bonne pratique** : **initialiser un pointeur à NULL**
  - évite une valeur indéterminée et permet de tester le pointeur
  - NULL : macro qui signifie « adresse invalide », p. ex. 0
- `void*` désigne un type de pointeur *générique*
  - utile dans les déclarations (ou les `cast`), ou lorsqu'on ne connaît pas à l'avance le type exact (ex. : `memcpy`, `qsort`, etc.)
  - on peut convertir librement entre un autre type et `void*`

Rôle de const avec les pointeurs

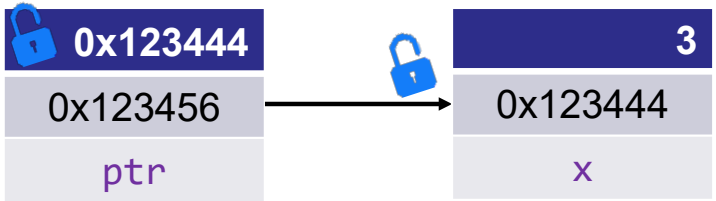
- Exemple :



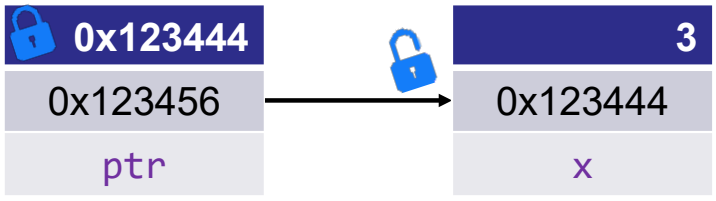
- Objet du 'const' dans `const int*` `const ptr` `= &x;`
  - le 1er indique que la valeur pointée ne peut changer, alors que le 2e se réfère à son adresse (contenue dans le pointeur)

Rôle de const avec les pointeurs

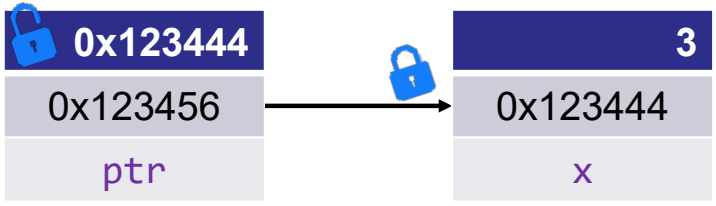
```
int  x   = 3;
int* ptr = &x;
```



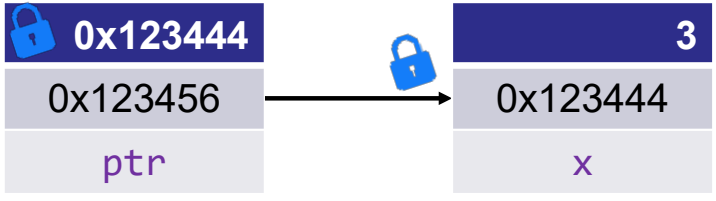

```
int* const ptr = &x;
```




```
const int* ptr = &x;
```




```
const int* const ptr = &x;
```



... et si l'on avait **const** int X = 3; ???

Erreurs fréquentes avec les pointeurs

- Le déréférencement (*) d'un pointeur qui pointe sur une variable (ou objet) n'existant plus, retourne une valeur indéterminée
  - ces erreurs sont souvent difficiles à localiser et corriger

```
int* ptr = NULL;
{
   int x = 3;
   ptr = &x;
}
printf("%d", *ptr);
```

- Le déréférencement d'un pointeur non initialisé ou NULL produit une **erreur de segmentation** (*segfault*)
  - plus facile à déboguer que le cas précédent ☺
  - peut être évitée en testant si le pointeur est NULL

```
int* ptr = NULL;        // toujours initialiser un
                        // pointeur à NULL ou avec l'adresse
                        // d'une autre variable
if (ptr == NULL) {...} // ne pas utiliser la valeur *ptr
```

3. Pointeurs et fonctions : le passage par adresse

Adresses comme arguments de fonctions

- En C++, il est possible de déclarer des paramètres d'entrée/sortie en utilisant le passage par référence (&)
  - les références étaient des « alias » des variables

- En C, les références n'existent pas … comment faire ?
- Nous pouvons passer à une fonction l'adresse de la zone mémoire à considérer
  - Le paramètre est donc un pointeur qui contient l'adresse de la variable
  - Le passage est donc par copie … mais une copie de l'adresse
  - Nous ne sommes pas limités par le nombre de paramètres

Comparer : valeur vs adresse

- La modification d'une variable passée par valeur ne dépasse pas la portée de la fonction
- Le passage d'une adresse permet de la dépasser, car la fonction modifiera le contenu stocké à cette adresse

```
void inc(int val) {
  val = val + 2;
}

int i = 5;
inc(i);
printf("%d\n", i);
```

5

```
void inc(int* val) {
  *val = *val + 2;
}

int i = 5;
inc(&i);
printf("%d\n", i);
```

7

Exemple avec `scanf`

- Pour pouvoir fournir les valeurs de variables lues au clavier, **`scanf`** doit connaître leurs adresses

```
int  i   = 1;
int* ptr = &i;       // ptr : variable pointeur contenant
                     // l'adresse de i (p pointe sur i)

scanf("%d", &i);     // correct
scanf("%d", ptr);    // correct

scanf("%d", &ptr);   // incorrect
                     // l'utilisateur ne peut pas
                     // entrer une adresse au clavier !
```

Arguments de fonctions avec `const`

- On utilise un passage par adresse lorsque la valeur sera modifiée par une fonction
- Mais on peut aussi utiliser une adresse pour éviter une copie de la variable, si elle est coûteuse
  - *comme pour le passage des objets par référence en C++*
- Si une fonction ne modifie pas une valeur pointée par la variable pointeur (i.e. l'objet dont on passe l'adresse), il est recommandé de déclarer cet argument comme `const`
- Si la valeur pointée est une constante, le `const` est nécessaire

```
void f(const int* x) {
   *x = 0;  // error: assignment of read-only location '*x'
}
```

Pointeurs comme valeurs de retour

- Une fonction peut retourner une adresse
  - i.e. un pointeur sur une autre variable
- Mais, attention !
  Si l'adresse est celle d'une variable locale à la fonction, alors son contenu ne sera plus disponible après le retour de la fonction ☹
- A noter que gcc, p.ex., ne nous prévient pas (pas de warning !).
  Pire encore, il se peut qu'à l'exécution le résultat affiché soit 7.

```
int* f(int* ptr) {
   …
   return ptr;
}
```

```
int* f(void) {
   int  i   = 7;
   int* ptr = &i; // variable locale
   return ptr;
}

int main(void) {
   int* ptr = f();
   printf("%d\n", *ptr); // !!
   return EXIT_SUCCESS;
}
```

4. Règles d'interprétation et d'écriture d'un déclarateur

Règles d'interprétation d'un déclarateur

- **Question** : Etant donné un déclarateur (D) écrit en C
  (p. ex. `int (*p)[10]`), comment l'interpréter ?

- **Réponse** : En utilisant les règles définies dans le tableau ci-dessous

Source: "*Langage C*", Claude Delannoy, Eyrolles 2003, p. 774 – 780

Règle	Affirmation avant application de la règle	Affirmation après application de la règle
Parenthèses	(D) est un T	D est un T
Pointeur	*[qualifieurs] D est un T	D est un pointeur (év qualifié par les qualifieurs) sur T
Tableau	D[n] est un T	D est un tableau de n T
Fonction	D(arg) est un T	D est une fonction renvoyant un T et dont les arguments sont d'un type défini par arg (si cette mention est présente)

N.B. *On applique toujours en priorité la première (selon l'ordre d'apparition dans le tableau) des quatre règles ci-dessus.*

Règles d'interprétation d'un déclarateur

- **Exemple** Interprétation de int (*p)[10]

1. Affirmation initiale: `(*p)[10]` est un `int`
2. Application de la règle tableau[1] : `(*p)` est un tableau de 10 `int`

   [1] la règle pointeur ne doit plus s'appliquer en premier, compte tenu des parenthèses
3. Application de la règle parenthèses : `*p` est un tableau de 10 `int`
4. Application de la règle pointeur : **p est un pointeur sur un tableau de 10 int**

Règles d'écriture d'un déclarateur

- **Question** : Comment écrire en C le déclarateur correspondant p. ex. à l'énoncé suivant : "p est un pointeur sur un tableau de 10 `int`" ?

- **Réponse** : En utilisant les règles définies dans le tableau ci-dessous

Source: "*Langage C*", Claude Delannoy, Eyrolles 2003, p. 774 – 780

Règle	Affirmation avant application de la règle	Affirmation après application de la règle
Pointeur	D est un pointeur (év qualifié par des qualifieurs) sur T	*[qualifieurs] D est un T (on peut placer des parenthèses autour de D, mais elles sont superflues)
Tableau	D est un tableau de n T (à noter que n n'est pas toujours spécifié)	(D)[n] est un T Si D ne commence pas par *, on peut supprimer les parenthèses autour de D
Fonction	D est une fonction renvoyant un T et dont les arguments sont d'un type défini par arg (si cette mention est présente, comme conseillé)	(D)(arg) est un T, si le type des arguments est spécifié (forme conseillée) (D)() est un T, si le type des arguments n'est pas spécifié (forme déconseillée) Dans les deux cas, si D ne commence pas par *, on peut supprimer les parenthèses autour de D

N.B. *Ici aucune priorité dans l'ordre des règles n'est à appliquer.*

Règles d'écriture d'un déclarateur

- **Exemple** Ecriture de "p est un pointeur sur un tableau de 10 `int`"

1. Application de la règle pointeur : `*p` est un tableau de 10 `int`
2. Application de la règle tableau : `(*p)[10]` est un `int`
   (ici, comme le contenu des parenthèses commence par *, on ne peut pas les supprimer)
3. D'où le résultat: `int (*p)[10]`

5. Pointeurs et tableaux : arithmétique des pointeurs

Un tableau : pointeur vers le 1er élément

- Exemple : tableau `tab` d'entiers

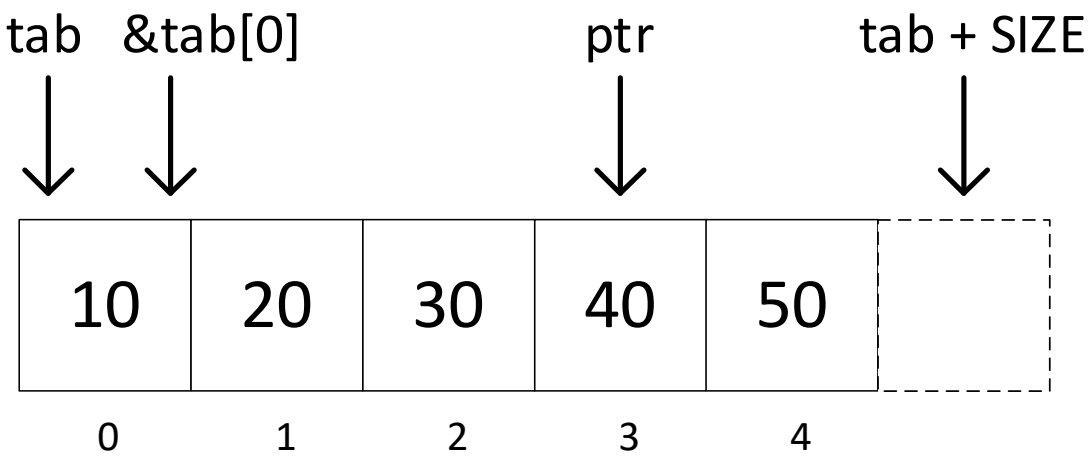
```
#define SIZE 10
int tab[SIZE];
```

- On accède aux éléments via `tab[0], tab[1], …, tab[SIZE-1]`
- On accède à l'adresse des éléments via `&tab[i]`
- En réalité, **`tab` vaut l'adresse du premier élément**
  - `tab` est équivalent à `&tab[0]`
- On pourra donc faire des additions et des soustractions sur des adresses (pointeurs) pour passer d'un élément d'un tableau à un autre (ce qui est très pratique)

Illustration

```
#define SIZE 5
int tab[SIZE] = {10, 20, 30, 40, 50};
int* ptr = &tab[3];
printf("%d", *ptr);
```

40



Pointeurs vers des tableaux : opérations

- Trois opérations sont possibles avec des pointeurs
  1. **Ajouter un entier à un pointeur** : si `ptr` contient l'adresse de l'élément `tab[i]`, alors `ptr + j` contient celle de l'élément `tab[i + j]`
     - c'est pour cela qu'on doit déclarer le type de données de `ptr` : pour savoir de combien d'octets avancer dans la mémoire
  2. **Soustraire un entier d'un pointeur** : même fonctionnement
  3. **Soustraire un pointeur d'un autre**, sur le même tableau : le résultat est leur écart, en nombre d'éléments du tableau

- **Attention !** Le comportement est indéfini (erroné) si :
  - on calcule avec des pointeurs qui ne pointent pas vers des tableaux
  - on soustrait des pointeurs entre des tableaux différents

Incrémentation des pointeurs (++)

- Si `tab` est un tableau et `p` pointe sur `tab[i]`, donc `p = &tab[i]`, alors **`tab[i++]` peut s'écrire `*p++`**
  - par exemple, `tab[i++] = 3` équivaut à `*p++ = 3`
- L'opérateur ++ est plus prioritaire que *

`*p++` ou `*(p++)`	Retourne la valeur pointée *p avant incrémentation, incrémente p ensuite (=> avance dans le tableau)
`(*p)++`	Retourne la valeur pointée *p avant incrémentation, incrémente *p ensuite (=> change la valeur pointée)
`*++p` ou `*(++p)`	Incrémente p d'abord (=> avance dans un tableau) retourne la valeur pointée *(p+1)
`++*p` ou `++(*p)`	Incrémente *p d'abord (=> change la valeur pointée), retourne la valeur incrémentée (*p)+1

Comparaison

- On peut tester l'égalité de deux pointeurs du même type : contiennent-ils la même adresse ?
  - `ptr1 == ptr2`, ou bien `ptr1 != ptr2`
- On peut en outre comparer la position de deux pointeurs vers le même tableau avec les inégalités : `<, <=, >, >=`

Equivalence pointeur / nom de tableau

- **`tab + i` est synonyme de `&tab[i]`**
  - adresse du i^ème élément du tableau tab
- **`*(tab + i)` équivaut à `tab[i]`**
  - le i^ème élément de tab

```
int tab[10];
*tab = 7;          // équiv. à tab[0] = 7
*(tab + 1) = 12; // équiv. à tab[1] = 12
```

- On peut donc écrire :

```
for (ptr = tab; ptr < tab + n; ++ptr) {
   somme += *ptr;
}
```

- On ne peut pas écrire :

```
while (*tab != 0) {tab++;}
// on ne peut pas changer l'adresse de tab !
```

Equivalence pointeur / nom de tableau

- On peut **appliquer l'opérateur [ ] à un pointeur sur un tableau**, comme si c'était un nom de tableau

```
int  tab[] = {1, 2, 0};
int* ptr   = &tab[0];
// ces écritures sont identiques
tab[2]     = 3;
*(ptr + 2) = 3;
ptr[2]     = 3;
```

- Depuis C99, on peut utiliser explicitement un pointeur comme nom de tableau
- **Attention !** `ptr` est le seul moyen d'accès au tableau raison du `const`

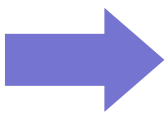
```
int  tab[] = {1, 2, 3};
int* ptr   = &tab[0];

// peut être remplacé par
int* const ptr = (int[]) {1, 2, 3};
```

Calcul d'adressage

- Les adresses en mémoire font référence à des zones d'un octet (byte), mais la plupart des types en C occupent plusieurs octets
- En utilisant le type d'un pointeur, le compilateur effectue un calcul pour incrémenter correctement son adresse
- Par exemple, pour avancer correctement sur le premier octet du 2e élément du tableau `tab`

```
int tab[10];
int* ptr = tab;
printf("%p\n", (void*)ptr++);
printf("%p\n", (void*)ptr);
```



```
000000000061FDF0
000000000061FDF4
```

Tableaux à plusieurs dimensions

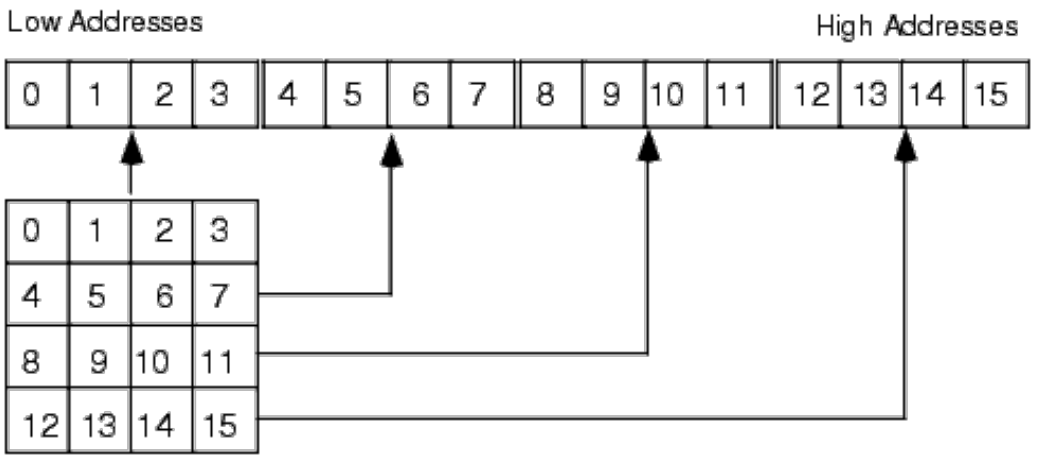
- Les pointeurs peuvent également pointer vers des éléments de tableaux multidimensionnels
  - Rappel : un tableau peut avoir une ou plusieurs dimensions
- La déclaration suivante crée un tableau bidimensionnel = une matrice →

```
int m[5][9];
```

  - m a 5 lignes et 9 colonnes, indexées à partir de 0
  - `m[i][j]` est l'élément de la rangée i, colonne j
  - `m[i]` désigne toute la rangée i de m
- Les matrices sont stockées dans la mémoire de manière séquentielle, ligne après ligne : d'abord 0, puis 1, etc.

Stockage en mémoire : exemple

```
#define SIZE 4
int m[SIZE][SIZE] = {{ 0, 1, 2, 3},
                     { 4, 5, 6, 7},
                     { 8, 9,10,11},
                     {12,13,14,15}};
```



Rôle des pointeurs

```c
int  m[NUM_ROWS][NUM_COLS];
int* ptr;

/* initialiser le tableau à 0 */
for (ptr = &m[0][0]; ptr <= &m[NUM_ROWS - 1][NUM_COLS - 1]; ++ptr) {
   *ptr = 0;
}  /* en réalité il vaut mieux utiliser 'memset' */
```

- **Important !**
  m vaut bien `&m[0][0]` en terme d'adresse mais n'est pas du même type
- m est du type `int(*)[NUM_COLS]` un tableau à une dimension, dont les éléments sont eux-mêmes des tableaux à une dimension (NUM_COLS)

Dimensions et type des pointeurs

- Lorsqu'un tableau a deux dimensions ou plus, le type du pointeur correspondant au nom du tableau dépend de toutes les dimensions, sauf la première
- Par exemple, si on déclare : `int tab[m][n], (*ptr)[n];`
  alors il est acceptable d'écrire `ptr = tab;`
- **Attention** Si on déclare `int tab[m][n], (*ptr)[m];`
  et écrit `ptr = tab;` alors le code compile (pas de vérification),
  mais si m ≠ n, il aura un comportement indéfini

Tableaux de dimensions variables

- Comment indiquer dans les paramètres d'une fonction un tableau dont les dimensions sont inconnues à la compilation ? (par exemple, en 2D)

- Deux possibilités
  1. Considérer le tableau comme ayant un seul indice, et expliciter les calculs de pointeurs correspondant aux valeurs des indices réels 2D (ou plus)
  2. Créer dynamiquement des tableaux de pointeurs sur les lignes d'un tableau à 2 indices, ce qui permettra de retrouver le formalisme tableau (p.ex. `tab[i][j]`)

1. Tableau 2D considéré comme 1D

- On passe à la fonction l'adresse du (début du) tableau et ses deux dimensions (n et p) :

```
void f(int* adr, size_t n, size_t p)
```

- Si on souhaite lui passer un tableau `t` déclaré comme `int t[2][3];` alors un appel comme `f(t, 2, 3)` ne compile pas, car `t` n'est pas de type `int*` , mais `int(*)[3]` (pointeur sur des blocs de trois `int`)
  - **solution** : caster le paramètre effectif `t => f((int*)t, 2, 3);`
- Accès à un élément du tableau dans la fonction f
  - `adr + i*p` pointe sur le 1er élément de la ième ligne
  - **pour accéder à `t[i][j]` dans f** : `*(adr + i*p + j)`

2. Création de pointeurs sur les lignes

- On passe toujours l'adresse et les dimensions

```
void f(int* adr, size_t n, size_t p)
```

- A l'intérieur de la fonction, on crée un **tableau temporaire** (voir plus loin pour *malloc*) de pointeurs **sur les n lignes** du tableau d'origine, et on le remplit, comme ceci :

```
int** ad = (int**) malloc(n * sizeof(int*));
for (size_t i = 0; i < n; ++i)
    ad[i] = adr + i*p;
```

- L'élément (i, j) du tableau sera bien **`ad[i][j]`**
  - parce que cela vaut `(ad[i])[j]`, donc `*(ad[i] + j)` donc `*(adr + i*p + j)` comme avant

Pointeurs sur des pointeurs

- On peut avoir des tableaux de pointeurs, puisque les tableaux peuvent contenir des données de n'importe quel type
- Si un argument d'une fonction est un pointeur, on peut souhaiter que la fonction modifie sa valeur, donc on doit passer l'adresse de ce pointeur

```
bool next(int** ptr, const int* tab, size_t taille) {
   ++(*ptr); // ou ++*ptr;
   return *ptr >= tab && *ptr < tab + taille;
}

int  tab[] = {1, 2, 3, 4, 5, 6};
int* ptr   = tab + 2;

printf("%d\n", *ptr);    // => 3
if ( next(&ptr, tab, sizeof(tab) / sizeof(int)) ) {
   printf("%d\n", *ptr); // => 4
}
```

6. Pointeurs sur fonctions

Définition et usage

- En C, les pointeurs peuvent également contenir les adresses de fonctions, pas seulement de données
  - en effet, les fonctions se trouvent également en mémoire, donc chaque fonction a une adresse
  - le nom de chaque fonction est traduit par le compilateur en une adresse mémoire
- Comment faire pour passer le nom d'une fonction comme argument d'une autre fonction ?
- Comment faire pour définir une variable qui peut prendre comme valeur un nom de fonction ?

➢ Utiliser des **pointeurs sur des fonctions**

Utilisation avec des variables pointeurs

- On peut déclarer un pointeur sur une fonction. Exemple : **`int (*f)(int);`**
  - On peut ensuite stocker dans un tel pointeur les adresses de diverses fonctions

```
int carre(int n) {return n * n;}
int cube(int n)  {return n * n * n;}

int i = 3;
int (*f) (int) = &carre; // f = pointeur sur fonction int->int
printf("%d\n", (*f)(i)); // affiche 9
f = &cube;
printf("%d\n", (*f)(i)); // affiche 27
```

- Un nom de fonction est implicitement converti en un pointeur vers elle, donc `f = cube;` est acceptable

Passage d'une fonction en argument

- **Exemple** calculer l'intégrale d'une fonction quelconque
- On définit une fonction appelée `integrate` qui approxime l'intégrale d'une fonction f (mathématique) passée en argument : nécessairement via un pointeur

```
// f = pointeur sur fonction double->double
double integrate(double (*f)(double), double a, double b) {
   y_a = (*f)(a); // appel de f
   y_b = f(b);    // autre forme acceptable et surtout plus simple
   ...
}
...
result = integrate(sin, 0.0, PI / 2);
result = integrate(cos, 0.0, PI / 2);
```

Exemple avec `qsort` de `stdlib.h`

```
void qsort(void* base, size_t nmemb, size_t size,
           int (*compar)(const void*, const void*));
```

- **`qsort`** est une fonction générique de tri qui peut trier un tableau pourvu qu'on lui fournisse le critère de comparaison entre deux éléments, sous la forme d'un pointeur sur une fonction, noté `compar` dans la déclaration
- le critère de comparaison entre p et q doit retourner
  - une valeur < 0 (non spécifiée) si *p doit être considéré comme plus petit que *q selon le critère de tri (donc si *p doit venir avant *q dans le tableau trié)
  - une valeur > 0 dans le cas contraire (si *p vient après *q)
  - 0 si *p et *q sont considérés égaux

Dernier exemple

- Définir une série de commandes, par ex. pour un traitement de texte simple
  - tableau de pointeurs sur des fonctions

```
void (*file_cmd[]) (void) = {new_cmd,
                             open_cmd,
                             close_cmd,
                             close_all_cmd,
                             save_cmd,
                             save_as_cmd,
                             save_all_cmd,
                             print_cmd,
                             exit_cmd};
```

- Demander à l'utilisateur de choisir une commande en entrant un nombre n
- Accéder directement à la commande : `file_cmd[n]();`

7. Allocation dynamique de la mémoire

Allocation dynamique de la mémoire

- En C, les structures de données, notamment les tableaux, sont normalement de taille fixe
  - cela peut être un problème, car on est obligé de choisir leur taille lors de l'écriture d'un programme ☹
- Heureusement, C/C++ supporte l'**allocation dynamique de la mémoire**, c'est-à-dire la possibilité d'allouer de la place en mémoire pendant l'exécution d'un programme
  - on peut donc utiliser des structures de données qui se développent et/ou se réduisent selon nos besoins ☺

❖ C propose plusieurs fonctions d'allocation dynamique de la mémoire

Fonctions d'allocation dynamique

- La librairie <`stdlib.h`> déclare trois fonctions permettant d'allouer dynamiquement de la mémoire :
  - **`malloc`** alloue un bloc de mémoire, mais n'initialise pas l'espace mémoire
  - **`calloc`** alloue un bloc de mémoire et l'initialise à zéro (tous les bits à 0)
  - **`realloc`** redimensionne un bloc de mémoire précédemment alloué (sans initialiser)
- Ces fonctions retournent toutes un pointeur générique (`void*`)
  - ce pointeur contient l'adresse (de début) de la zone mémoire allouée
  - … ou NULL si l'allocation dynamique n'a pas pu se faire
  - ❖ Corollaire : stocker la valeur de retour dans un pointeur puis vérifier qu'il n'est pas NULL

malloc – sémantique et valeur de retour

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

- Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.
- The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.
- If *size* is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.
- **Return value**
  - On success, a pointer to the memory block allocated by the function.
    The type of this pointer is always *void**, which can be cast to the desired type of data pointer in order to be dereferenceable.
  - If the function failed to allocate the requested block of memory, a *null pointer* is returned.

Source : https://www.cplusplus.com/reference

malloc – exemple

- **Exemple** Allocation dynamique d'une zone mémoire permettant d'y stocker 10 entiers (de type `int`)

```
const size_t N = 10;
int* ptr = (int*)malloc(N * sizeof(int));
if (!ptr) {
   // échec de l'allocation dynamique ...
}
else {
   // réussite de l'allocation dynamique ...
}
```

- On peut tester les pointeurs comme les nombres :
  tout pointeur non NULL est vrai, et tout pointeur NULL est faux

```
if (ptr == NULL) // équivaut à if (!ptr)
if (ptr != NULL) // équivaut à if (ptr)
```

calloc – sémantique et valeur de retour

```
void* calloc(size_t num, size_t size);
```

- Allocates a block of memory for an array of *num* elements, each of them *size* bytes long, and initializes all its bits to zero.
- The effective result is the allocation of a zero-initialized memory block of (*num*size*) bytes.
- If *size* is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.
- **Return value**
  - On success, a pointer to the memory block allocated by the function.
    The type of this pointer is always *void**, which can be cast to the desired type of data pointer in order to be dereferenceable.
  - If the function failed to allocate the requested block of memory, a *null pointer* is returned.

Source : https://www.cplusplus.com/reference

realloc – sémantique et valeur de retour

```
void* realloc(void* ptr, size_t size);
```

- **ptr** Pointer to a memory block previously allocated with *malloc*, *calloc* or *realloc*. Alternatively, this can be a *null pointer*, in which case a new block is allocated (as if *malloc* was called).
- **size** New size for the memory block, in bytes.

- Changes the size of the memory block pointed to by *ptr*.
- The function may move the memory block to a new location (whose address is returned by the function).
- The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new size is larger, the value of the newly allocated portion is indeterminate.
- If *size* is zero, the return value depends on the particular library implementation: it may either be a *null pointer* or some other location that shall not be dereferenced.

Source : https://www.cplusplus.com/reference

realloc – sémantique et valeur de retour

- **Return value**
  - A pointer to the reallocated memory block, which may be either the same as *ptr* or a new location.
  - A *null pointer* indicates that the function failed to allocate storage, and thus the block pointed by *ptr* was not modified.

Source : https://www.cplusplus.com/reference

`realloc` – précautions à prendre

- **Attention !**
  - si la zone mémoire courante ne peut être agrandie, `realloc` va allouer une autre zone mémoire et y recopier le contenu courant
  - le pointeur retourné sera différent de `ptr`
  - Corollaire :
    Nécessaire de mettre à jour tous les pointeurs pointant sur l'ancienne zone mémoire !

Nécessité d'une fonction `free`

- `malloc` et les autres fonctions d'allocation de mémoire allouent des blocs dans une partie de la mémoire appelée **tas** (*memory heap*)
- des appels fréquents et/ou de grandes zones demandées peuvent épuiser la place disponible dans le tas
  - les appels suivants retourneront le pointeur NULL
- **Fuite de mémoire** (*memory leak*) : demandes successives d'allocation de zones, puis perte des pointeurs vers elles → zones inaccessibles et inutiles → tas épuisé
- Comment éviter les fuites ? **Appeler `free`**
  - certains langages le font automatiquement = ***garbage collector***
  - outils de détection pour C :
    valgrind, MemorySanitizer (LLVM), Dr. Memory

free – sémantique

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

- A block of memory previously allocated by a call to `malloc`, `calloc` or `realloc` is deallocated, making it available again for further allocations.
- If *ptr* does not point to a block of memory allocated with the above functions, it causes undefined behavior.
- If *ptr* is a null pointer, the function does nothing.
- Notice that this function does not change the value of *ptr* itself, hence it still points to the same (now invalid) location.

Source : https://www.cplusplus.com/reference

free – précautions à prendre

- **Attention !**

  (Rappel) Une fois la mémoire restituée via `free(ptr)`, *ptr* pointe toujours sur la zone mémoire initialement allouée (désormais invalide).

```
char* ptr = (char*)malloc(4);
free(ptr);
ptr[2] = 'a'; // erreur !
```

Solution : Faire `ptr = NULL;` immédiatement après avoir fait `free(ptr);`

```
free(ptr);
ptr = NULL;
ptr[2] = 'a'; // segfault : garde-fou
```

free – précautions à prendre

- **Attention !**

  Si plusieurs pointeurs pointent vers une zone libérée avec `free`, tous deviennent invalides (*dangling pointers*) après la libération

  - essayer de les déréférencer => résultat indéterminé !
  - modifier leurs valeurs => risques encore plus grands !

```
char* ptr1 = (char*)malloc(4);
char* ptr2 = NULL;
if (ptr1) {
   ptr2 = ptr1 + 3;
}
free(ptr1);  // ptr2 devient invalide aussi
ptr1 = NULL; // Mettre à NULL ptr1 et ptr2 !
ptr2 = NULL;
```

8. Manipulation de la mémoire

<string.h>

memcpy

```
void* memcpy(void* destination, const void* source, size_t num);
```

- copie *num* octets d'une zone commençant à *source* vers une zone commençant à *destination*
- retourne *destination*
- moyen le plus rapide (et portable) de transférer des données entre des zones mémoire
  - optimisé → les zones *source* et *destination* doivent être disjointes

```
#define SIZE 10
double tab[SIZE];
double* ptr = (double*)calloc(SIZE, sizeof(double));
memcpy(tab, ptr, SIZE * sizeof(double));
/* les éléments de tab sont maintenant tous à zéro */
```

memmove

```
void* memmove(void* destination, const void* source, size_t num);
```

- copie *num* octets d'une zone commençant à *source* vers une zone commençant à *destination*
- ici, contrairement à `memcpy`, les zones *source* et *destination* peuvent se chevaucher

memset

```
void* memset(void* ptr, int value, size_t num);
```

- met les *num* premiers octets de la zone mémoire pointée par *ptr* à la valeur *value* (interprétée comme un unsigned char)
- retourne *ptr*

```
#define N 10
double* ptr = (double*)malloc(N * sizeof(double));
memset(ptr, 0, N * sizeof(double));
/* tous les bits de la zone mémoire sont maintenant à zéro */
```

memcmp

```
int memcmp(const void* ptr1, const void* ptr2, size_t num);
```

- compare deux à deux les *num* premiers octets des deux zones mémoire pointées par *ptr1* et *ptr2*, et retourne :
  - 0 si les deux zones ont le même contenu
  - une valeur négative (non spécifiée) si le premier octet différent a une valeur plus petite dans la zone 1 que dans la zone 2
  - une valeur positive (non spécifiée) dans le cas inverse

N.B. Les octets sont interprétés comme étant du type unsigned char

memchr

```
void* memchr(const void* ptr, int c, size_t num);
```

- recherche dans les *num* premiers octets de la zone mémoire désignée par *ptr* la première occurrence de l'octet *c*
- les octets de la mémoire ainsi que *c* sont interprétés comme des unsigned char
- retourne un pointeur sur le premier octet égal à *c* qui a été trouvé, ou NULL si pas trouvé

9. Résumé

Résumé

- Le concept de pointeur est un concept-clé en C car omniprésent.
- Le concept de pointeur permet, notamment, d'écrire des fonctions prenant en paramètre un tableau multidimensionnel de taille quelconque.
- L'allocation dynamique de la mémoire permet d'allouer de la mémoire en cours d'exécution d'un programme, en fonction des besoins de ce dernier.
- C propose trois fonctions d'allocation dynamique de la mémoire : `malloc`, `calloc` et `realloc`.
- En C, c'est au développeur que revient la tâche de restituer la mémoire allouée dynamiquement qui n'est plus utilisée (pas de restitution automatique via un ramasse-miettes (*garbage collector*) comme p. ex. en Java).
- La fonction `free` permet de restituer la mémoire que l'on s'est préalablement allouée dynamiquement.

Résumé

- C++ propose aussi des opérateurs permettant d'allouer (`new`) et de restituer (`delete`) dynamiquement de la mémoire
- `malloc`, `calloc`, `realloc` et `free` peuvent aussi s'utiliser en C++
  … mais il est préférable d'utiliser `new` et `delete` car mieux adaptés à l'allocation / restitution dynamique d'objets (de type classe).
- C propose, via la librairie <string.h>, diverses fonctions de manipulation de la mémoire : `memcpy`, `memmove`, etc.