# Gestion de la mémoire





## 1. La structure de la mémoire

- La mémoire comporte 4 zones, 3 pour l'allocation et une pour l'exécution du programme.
- Chaque zone sert à mémoriser des types différents de variables.
    - Pile (Stack): pour les variables locales ou automatiques.
    - Tas (Heap): pour les variables allouées dynamiquement.
    - Zone statique: pour les variables statiques.
    - Zone exécutable : pour l'exécution du programme.

## 2. Les classes de stockage

- La syntaxe générale d'une définition ou d'une déclaration de variables est sous la forme :

```
classe modificateur type identificateur ;

static const int x;
```

- La classe de stockage détermine le mode de stockage, la durée de vie et la visibilité de la variable.
- Le C++ offre 2 catégories de classes de stockage :

```
Automatique => « auto », register

Statique => extern, static
```

### 2.1. « auto »

- Depuis « C++11 », le mot clé réservé « auto » a pris un autre sens.
- Il ne fait plus référence à la classe de stockage par défaut, « automatique ».
- Il est utilisé comme une inférence de type (rechercher automatiquement le type d'une variable).

### 2.2. « register »

- Les variables "registre" sont définies avec le mot clé « register ».
- Elles obéissent aux mêmes règles que les variables automatiques.
- Elles sont stockées plutôt dans un des registres du microprocesseur au lieu de la mémoire principale de l'ordinateur (RAM), si cela est possible. Ceci va permettre d'optimiser l'accès à la variable.
- Si l'allocation dans un registre n'est pas possible, la classe "auto" est utilisée à la place.
- Ce type d'allocation n'est valable que pour des types simples : char, int ou pointeur.
- Les variables allouées n'auront pas d'adresse en mémoire donc on ne peut pas affecter un pointeur à l'adresse d'une variable « register ».





### 2.3. « extern »

- Cette classe permet de renseigner sur l'existence d'un objet (variable, fonction, classe, etc.) qui est définie plus loin dans le fichier ou dans un autre module de l'application (un autre fichier par exemple).
- Cette déclaration ne crée pas de nouvel objet et, en particulier elle n'entraîne aucune allocation de mémoire pour quelque donnée que ce soit.
- La durée de vie des objets externes est permanente et elle est égale à celle du programme (le fichier où elle a été définie ainsi que les autres fichiers appartenant au programme).
- Les variables globales ont la classe « extern » lorsqu'elles sont définies sans aucune indication de classe de mémorisation.
- Une fonction est aussi de classe externe si on ne précise pas sa classe. Le mot clé « extern » est donc implicite.
- Par défaut, une variable « extern » est initialisée avec la valeur 0.
- Pour les tableaux, on peut omettre la taille de la première dimension.

### 2.4. « static »

- Les variables statiques sont définies avec le mot clé « static ».
- Elles sont stockées en zone statique.
- Elles ont la même portée que les variables "auto", mais leur durée de vie est permanente. Elles existent donc en mémoire pendant toute l'exécution du programme.
- Si une variable statique est initialisée, cette initialisation est faite au moment de la compilation (initialisée qu'une seule fois).
- Par défaut, une variable statique est initialisée automatiquement à 0.

**Pour une variable locale**

- La classe « static » permet de préserver la valeur de la variable locale entre les appels successifs à cette fonction.

- La variable reste locale à la fonction.

**Pour les variables ou fonctions globales**

- Le qualificatif « static » permet de définir un objet global privé à un fichier. Ceci permet de garantir un certain niveau d'encapsulation.

- L'attribut « static » informe le compilateur qu'il ne doit pas communiquer le nom de l'objet aux autres modules (autres fichiers par exemple).





### 2.5. Exemple

| 01mem.cpp | 01mem_main.cpp |
|---|---|
| `int tab[10];`<br>`static int v;`<br>`static char w;`<br><br>`int calcul (int a, int b) {`<br>`  tab[0]=a;`<br>`  tab[1]=b;`<br>`  return (fois5(a)+b);`<br>`}`<br>`static int fois5(int x) {`<br>`  return x*5;`<br>`}` | `extern int tab[10];`<br>`extern int calcul(int,int);`<br>`static int i, j;`<br>`static int fonc() {`<br>`  return 5;`<br>`}`<br>`int main() {`<br>`  i=j=3;`<br>`  cout << calcul(i,j) << endl;`<br>`  cout << tab[0] << ";"\`<br>`  << tab[1] << endl;`<br>`  return 0;`<br>`}` |
| Le tableau « tab » et la fonction « calcul(int,int) » sont déclarés comme variables globales publiques.<br>Les variables « v, w » et la fonction « fois5(int) » sont déclarées comme variables globales privées. | Le tableau « tab » est celui défini dans « fichier_1.cpp », de même que pour la fonction « calcul(int,int) ».<br>Les variables « i, j » ainsi que la fonction « fonc() » sont déclarées comme variables globales privées.<br>La méthode « main » est une variable globale publique. |

- Attention : lors de l'édition de liens, il ne faut pas oublier d'ajouter sur la ligne de commande le fichier dénommé « 01mem.o ».

## 3. Stockage des éléments en C++

- Comme nous l'avons expliqué précédemment, les éléments sont stockés en mémoire en fonction de leur classe de stockage.
- Les éléments peuvent avoir donc des durées de vie différentes.

### 3.1. Emplacement constant

```
const char *texte1 = "Bonjour";
char *texte2 = "Un autre bonjour";

texte1[1]= 'a'; // Erreur de syntaxe
texte2[1]= 'a'; // Peut provoquer une erreur d'exécution
```

- Emplacements constants : les données sont stockées en dehors de la zone de lecture/écriture du programme.

- Une modification sur ces données peut provoquer des erreurs indéfinies.

### 3.2. Tableau versus une chaîne de caractères

- Il y a une différence entre une chaîne de caractères et un tableau de caractères.

- Soit les déclarations suivantes d'un tableau dans la zone lecture/écriture du programme :





```
char tab1[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
char tab2[] = "Bonjour";
```

- Et les déclarations suivantes de pointeurs :

```
char *s1 = "Bonjour"; // s1 pointe le caractère 'B'.
char *s2 = "Bonjour"; // Fort probable que s2 pointe à la même
                         place
```

- Affectation à des variables « tableaux » :

```
*tab1 = 'Q'; *tab2 = 'A';
```

- Modification de la chaîne de caractères :

```
*s1 = 'Q'; *s2 = 'A'; // Peut générer une erreur de
                      // « segmentation » au moment
                      // de l'exécution.
```

## 4. Durée de vie

### 4.1. Durée de vie automatique

```
void f() {
    int j ; // La vie commence ici avec l'initialisation
} // La vie de « j » prend fin ici.
```

### 4.2. Durée de vie dynamique

```
char *p = new char[256] ; // La vie de « p » commence ici.
… … …
delete [] p ; // Elle prend fin ici.
```

- La signature de l'opérateur « delete » doit correspondre à celle de l'opérateur « new » utilisé lors de l'allocation de la mémoire.

### 4.3. Durée de vie statique

- Pour les variables globales statiques, elle commence au début du programme et se termine à sa fin.
- Pour les variables locales statiques, elle commence lors de la première rencontre et se termine à la fin du programme.





## 5. Description des opérateurs « new » et « delete »

- « malloc » et « free » servent à allouer de la mémoire à un élément donné et libérer l'espace mémoire alloué. Ces fonctions sont définies dans le langage « C ». Par extension, elles sont disponibles aussi dans le langage « C++ ».

- « new » et « delete » servent aussi à allouer de la mémoire à un élément donné et libérer l'espace alloué. Elles sont décrites uniquement dans le langage « C++ ».

- Allocation de la mémoire : « malloc » ou « new ».

- Libérer l'espace mémoire alloué : « free » ou « delete ».

- Quand on utilise « malloc » pour allouer de la mémoire à un élément donné, on utilise « free » pour libérer cette mémoire.

- Quand on utilise « new » pour allouer de la mémoire à un élément donné, on utilise « delete » pour libérer cette mémoire.

- On ne mélange pas les deux, par exemple, « malloc » avec « delete », « new » avec « free » etc.

```
void fonction(void){

    T x;

}
```

- Allocation de l'espace nécessaire pour contenir « x », une instance de « T ».
- Construction de l'instance « x » dans l'espace précédemment alloué.
- Destruction de l'instance « x ».
- Libération de l'espace mémoire alloué pour l'instance « x ».





### 5.1. L'opérateur « new »

**Signature :**

```
void* operator new(std::size_t);    // version simple

void* operator new[](std::size_t);  // version tableau
```

**Utilisation :**

```
int *z = new int;     // version simple

int *x = new int[4];  // version tableau
```

- Il est possible d'indiquer une valeur d'initialisation pour un seul élément alloué. Ce n'est pas le cas pour un tableau d'éléments.

```
int *pi = new int(5);
```

- Pour un tableau d'objets, il faudra prévoir un constructeur sans paramètres ou avec des valeurs par défaut.

**Interaction avec le constructeur :**

- Pour cet appel

```
T *ptr = new T ; // ptr sur une instance d'un objet du type T
```

- D'abord, il y a allocation en mémoire d'un espace suffisant pour contenir l'instance du type « T ».
- Si l'allocation réussit, il y aura construction de l'objet en mémoire en exécutant pour cela le contenu du constructeur.
- Finalement, l'adresse d'entrée de cette mémoire est stockée dans le pointeur « ptr ».





### 5.2. L'opérateur « delete »

**Signature :**

```
void  operator delete(void *);  // version simple
void  operator delete[](void*); // version tableau
```

**Utilisation :**

```
delete z;    // version simple
delete [] x; // version tableau
```

-

**Interaction avec le destructeur :**

- Pour cet appel :

```
T *ptr = new T ;

delete ptr;
```

- La destruction de l'élément se fait comme suit :

```
delete ptr;
```

- Si le pointeur « ptr » n'est pas égal à zéro, détruire l'instance de « T » se trouvant dans la zone mémoire adressée par le pointeur « ptr ».
- Libérer la mémoire adressée par le pointeur « ptr ».





### 5.3. Tableaux dynamiques

- En « C++ », le processus de création d'un objet dynamiquement est différent de la création d'un tableau d'objets. Les deux situations ne sont pas gérées de la même manière.
- On utilise les opérateurs « new » et « delete » pour la création dynamique des instances de la classe ou de types prédéfinis.
- On utilise les opérateurs « new[ ] » et « delete [ ] » pour la création ou de la destruction de tableaux dynamiques.

```
Test *un_test; // Un pointeur sur une instance
               // de la classe Test

un_test = new Test[N]; // La valeur N est connue
                       // et doit-être du type « int ».

// on effectue diverses tâches avec un_test

delete [] un_test;
```

- Les opérations d'allocation et de destruction se font comme suit :
    - Allocation de la mémoire pour un tableau de taille « N ».

- Appel du constructeur par défaut pour chaque instance dans le tableau, dans un ordre croissant. Ceci va permettre de créer un nombre « N » d'instances.
- Le pointeur renvoyé pointe vers le début de la mémoire allouée, c.-à-d., la 1re instance dans le tableau.
- À la fin du programme, le « delete [ ] » va faire appel au destructeur de chaque instance afin de la détruire. Cette destruction va se faire dans l'ordre inverse de la création.
- Finalement, le tableau est détruit.

- La création de tableaux dynamiques d'objets :

```
// Un pointeur de pointeurs
Test** un_test;

// On commence par allouer un tableau de pointeurs
un_test = new Test* [N]; // N un int dont la valeur est connue

// On alloue par la suite chaque élément
// On fait appel au constructeur par défaut
for (int index = 0; index < N; index++)
     un_test[index]  = new Test [M]; // M un int dont la valeur
                                     // est connue
```





- La destruction de tableaux dynamiques d'objets :

```
// On commence par faire appel au destructeur de chaque
// instance
for (int index = N; index < N; index++)
     delete[] un_test[index];

// On efface finalement l'espace alloué pour
// le tableau de pointeurs
delete[] un_test;
```

### 5.4. Bogues

- Accès à une zone non allouée!

```
int* a = new int[40];
a[45] = 10; // 45! La taille du tableau est 40.
```

- Oups! On a oublié de libérer l'espace mémoire!

```
int main() {

    int * p = new int; // Allocation
    delete p;          // On libère l'espace, OK.
```

```
        // Fuite de mémoire

        int * q = new int;
        // il manque le delete
    }
```

- La variable est locale ou globale?

```
    int* get_xyz(){

        int xyz; // Variable locale
        xyz = 22;

        return &xyz; // On retourne son adresse, oups!
    }

    int main() {

        int *zzz;

        zzz = get_xyz(); // Ouch!

        // traitement quelconque
        // plantage assuré …
    }
```





### 5.5. Déboguage

- Il existe plusieurs outils pour diagnostiquer l'utilisation de la mémoire[1].
- Le plus simple est l'utilisation des affichages en sortie dans les zones critiques pour examiner le fonctionnement du programme.
- On peut utiliser les outils de débogage. On peut citer par exemple gdb pour gcc ou l'utilitaire intégré dans Visual Studio. On peut demander par exemple à ces outils de surveiller les débordements de tableau.
- Les commandes strace (appel système) et ltrace (appels de fonctions) vous permettent d'examiner qui utilise quoi et pourquoi.
- Valgrind[2] est un profiler de code permettant de détecter des fuites de mémoire.
- AddressSanitizer[3] et ThreadSanitizer, deux utilitaires développés pour le compilateur clang, intégrés depuis dans gcc. Ils sont plus rapides que Valgrind.
- Dr memory[4], est une autre approche à valgrind. Il est disponible aussi sur Windows ce qui n'est pas le cas de valgrind.
- Il y a plusieurs produits commerciaux qui permettent le traçage de la mémoire. Le plus connu est « purify ».

1 https://en.wikipedia.org/wiki/Memory_debugger
2 http://www.valgrind.org/
3 http://clang.llvm.org/docs/index.html
4 http://www.drmemory.org/

```
valgrind ./11_mem

==8759== Memcheck, a memory error detector
==8759== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8759== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==8759== Command: ./11_mem
==8759==
==8759== Warning: set address range perms: large range [0x395f8040, 0x6f95f8040) (undefined)
==8759==
==8759== HEAP SUMMARY:
==8759==     in use at exit: 28,991,029,248 bytes in 1 blocks
==8759==   total heap usage: 2 allocs, 1 frees, 28,991,101,952 bytes allocated
==8759==
==8759== LEAK SUMMARY:
==8759==    definitely lost: 0 bytes in 0 blocks
==8759==    indirectly lost: 0 bytes in 0 blocks
==8759==      possibly lost: 28,991,029,248 bytes in 1 blocks
==8759==    still reachable: 0 bytes in 0 blocks
==8759==         suppressed: 0 bytes in 0 blocks
==8759== Rerun with --leak-check=full to see details of leaked memory
==8759==
==8759== For counts of detected and suppressed errors, rerun with: -v
==8759== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

### 5.6. Notions complémentaires

- Pointeurs intelligents (voir chapitre sur les STL).





### 5.7. Question de placement

- On constate pour le programme suivant que l'on perd du temps dans les processus de l'allocation et de la libération de l'espace alloué.

```
for( long i = 0; i < 10000; ++i){
    for ( long j = 0; j < 10000; ++j){
         Test *dp = new Test();
         // ...
         delete dp;
    }
}
```

- Comment créer des objets (appel du constructeur) sans allouer un nouvel espace et se contenter de ce que nous avons alloué précédemment ?

- On utilise pour cela la version de « new » avec placement.

```
void* operator new(std::size_t, void* p) noexcept;
void  operator delete(void* p, void*) noexcept;

void* operator new[](std::size_t, void* p) noexcept;
void  operator delete[](void* p, void*) noexcept;
```

- En utilisant l'opérateur « new » avec placement, l'exemple précédent devient comme suit :

```
char *cp = new char[sizeof(Test)];
for( long i = 0; i < 10000; ++i){
    for ( long j = 0; j < 10000; ++j){
         Test *dp = new(cp) Test();
         // ...
         dp->~Test();
    }
}
delete [] cp;
```

```
« delete dp » au lieu de « dp->~Test() » ?
```

- On ne peut pas faire un appel explicite au « delete » car il correspond à la destruction de l'espace mémoire alloué par « cp ».

- Pour cette raison que l'on a fait un appel manuel au destructeur de la classe « Test », permettant de détruire ainsi l'instance sans toucher à l'espace mémoire occupé par cette instance.





### 5.8. Question de placement dans le cas d'une exception levée

- Imaginer un instant que vous aviez utilisé la version de l'opérateur « new » avec placement et que ce dernier a levé une exception. Comment pourrions-nous libérer l'espace mémoire déjà alloué et éviter une fuite mémoire ?

```
try {
    A *ptr = new A();
}
catch(A_exception){
    // Le système libère l'espace alloué pour A
}
```

```
char *cp = new char[sizeof(B)];
try {
    B *ptr = new(cp) B();
}
catch(B_exception){
    // Le système ne libère pas l'espace alloué pour B
}
```

- Pour libérer l'espace alloué, le système cherche, un opérateur « delete » qui matche dans sa signature l'opérateur « new » correspondant.

```
void* operator new(std::size_t);                                                    1
void  operator delete(void *) noexcept;

void* operator new[](std::size_t);                                                  2
void  operator delete[](void*) noexcept;


void* operator new(std::size_t, const std::nothrow_t&) noexcept;                    3
void operator delete(void*, const std::nothrow_t&) noexcept;

void* operator new[](std::size_t, const std::nothrow_t&) noexcept;                  4
void operator delete[](void*, const std::nothrow_t&) noexcept;

void* operator new(std::size_t, void* p) noexcept;                                  5
void  operator delete(void* p, void*) noexcept;

void* operator new[](std::size_t, void* p) noexcept;                                6
void  operator delete[](void* p, void*) noexcept;
```

- Dans le premier exemple (la classe « A »), nous avons utilisé la version « 1 » de l'opérateur « new ». C'est l'opérateur « delete » associé à ce « new » qui sera utilisé pour libérer la mémoire allouée.

- Dans le second exemple, nous avons utilisé l'opérateur « new » de la version « 5 ». C'est l'opérateur « delete » associé qui sera appelé pour libérer l'espace mémoire. Il faudra redéfinir l'opérateur « delete » pour qu'il libère effectivement l'espace mémoire adressé par « cp ».





### 5.9. Surdéfinition des opérateurs « new » et « delete »

- Les opérateurs « new » et « delete » peuvent être surdéfinis à 2 niveaux :
    - Comme une méthode statique membre d'une classe.
    - Comme une fonction globale (on parle ici plutôt d'une redéfinition).
- Les méthodes membres seront responsables de l'allocation des instances pour une certaine classe.
- Les fonctions globales seront associées à toutes les autres allocations mémoires.
- Un exemple de surdéfinition des opérateurs « new » et « delete » est l'utilisation des placements.
- Quelques règles doivent être respectées lors de la surdéfinition des opérateurs « new » et « delete » :
    - L'opérateur « new » doit retourner le type « void * ».
    - Le premier argument de l'opérateur « new » doit être « size_t » et ne peut pas avoir une valeur par défaut.
    - Une méthode qui ne lève pas d'exception en cas d'une erreur d'allocation doit quand même retourner une valeur nulle. La méthode standard doit lever l'exception « bad_alloc ».
    - L'opérateur « delete » doit retourner un « void » et son premier argument doit avoir le type « void * ».
    - Vous ne pouvez pas invoquer les destructeurs non explicites.

- Vous ne pouvez pas redéfinir les opérateurs de placement associés à la librairie standard du C++.
- Votre programme doit offrir tout au plus une seule définition pour la surdéfinition (redéfinition) de l'opérateur « new » (resp. « delete ») au niveau global. Il n'est donc pas possible de déclarer votre fonction comme static ou bien dans un espace de nom.

- Les prototypes à utiliser en cas d'une redéfinition sont comme suit :

| | |
|---|---|
| `new Type` | `void* operator new(std::size_t);` |
| `new (nothrow) Type` | `void* operator new(std::size_t*, const std::nothrow_t&) noexcept;` |
| `new Type[taille]` | `void* operator new[](std::size_t);` |
| `new (nothrow) Type[taille]` | `void* operator new[](std::size_t*, const std::nothrow_t&) noexcept;` |
| `delete ptr` | `void  operator delete(void *) noexcept;` |
| `delete [] ptr` | `void  operator delete[](void* p) noexcept;` |
| `pas d'appel explicit` | `void operator delete(void*, const std::nothrow_t&) noexcept;` |
| `pas d'appel explicit` | `void operator delete[](void*, const std::nothrow_t&) noexcept;` |





- Les prototypes à utiliser en cas d'une surdéfinition sont comme suit :

| | |
|---|---|
| `new (a1,an) Type` | `void* operator new(std::size_t,typ1,typen);` |
| `new (a1,an) Type[taille]` | `void* operator new[](std::size_t*,type1,typen);` |

- Vous remarquez qu'il n'est pas possible de surdéfinir l'opérateur « delete », on peut tout au plus changer le comportement (redéfinir) des méthodes de base.

**La surdéfinition au niveau global :**

- La surdéfinition globale des opérateurs « new » et « delete » permet de masquer définitivement les opérateurs « new » et « delete » originaux (prévus par le système).

```
void* operator new(std::size_t sz) {
    return malloc(sz);
}

void operator delete(void* ptr) noexcept {
    free(ptr);
}
```

**La surdéfinition au niveau de la classe :**

- Il est possible de surdéfinir les opérateurs « new » et « delete » au niveau de la classe.
- La surdéfinition est très utilisée dans le cas de classes homogènes (Listes chaînées, arbres binaires, etc.).
- Ainsi ces opérateurs seront étroitement reliés au problème traité et peuvent agir de manière efficace.
- Cette technique, si elle est bien implantée, elle va permettre d'améliorer grandement les performances du problème traité.

```
class A {
public:
    void* operator new(std::size_t);
    void* operator new[](std::size_t);
    void operator delete(void*);
    void operator delete[](void*);
};
```

```
A *pa = new A();    // A::new(sizeof(A))
A *pb = new A[5];   // A::new(5*sizeof(A))
delete pa;          // A::delete(pa)
delete [] pb;       // A::delete[](pb)
```





- Le « new » (resp. « delete ») sera utilisé pour allouer (resp. libérer) l'espace mémoire alloué pour contenir une instance de « T », ou bien une instance qui dérive de « T ».

```
void* A::operator new(std::size_t s) {
    return ::new char[s]; // new global
}

void A::operator delete(void* p) {
    ::delete reinterpret_cast<A *> p; // delete global
}
```

- L'opérateur « new » peut avoir des arguments supplémentaires ayant n'importe quel type.
- Les opérateurs « new » et « delete » obéissent aux mêmes règles qu'une méthode membre d'un classe quand il s'agit par exemple de la portée de la méthode (resp. fonction).
- Une des conséquences de la surdéfinition à l'intérieur de la classe est que les opérateurs « new » et « delete » définis au niveau global, seront de facto masqués.

```
class A {
public:
    void* operator new(std::size_t,int);
};

A *ptr = new A(); // Erreur
```

```
Nous avons besoin d'un opérateur new à deux arguments.
En réalité, nous avons besoin de faire l'appel suivant sous
la forme d'un seul argument, le premier étant implicite :

A *ptr = new(5) A(); ⇔ A::operator new(sizeof(A),5)
```

- Si les opérateurs au niveau global sont masqués, on peut quand même les appeler en utilisant pour cela l'opérateur de résolution de portée (::) comme suit :

```
A *ptr = new(5) A(); ⇔ A::operator new(sizeof(A),5)

A *btr = ::new A(); ⇔ new(sizeof(A))
::delete btr;
```

**Surdéfinition de l'opérateur « delete »**

- Nous avons mentionné qu'il n'est pas possible de surdéfinir l'opérateur « delete ».
- Il n'est donc pas possible d'ajouter des arguments supplémentaires à la méthode « delete ».
- En effet, on connaît le contexte de la création d'un objet, mais pas le contexte de sa destruction.
- On peut quand même placer des « cookies » (des informations supplémentaires) au niveau du constructeur pour avoir une idée comment détruire l'objet ou bien on peut essayer d'en extraire de l'information à partir de l'adresse de l'objet à libérer.





```
void operator delete (void *temp){
    if (temp==0) return;
    if (temp>uneadresse) faire quelque chose;
    sinon faire autrechose;
}
```

### 5.10. Les opérateurs « new[] » et « delete[] »

- Il n'est possible de surdéfinir les opérateurs « new[] » et « delete [] » qu'à l'intérieur d'une classe.
- Il est possible de les redéfinir de manière globale ou locale à une classe.
- La surdéfinition ou la redéfinition supposent que nous savons comment calculer la taille du tableau.
- Si les opérateurs « new[] » et « delete[] » sont définis de manière globale, ils masquent la présence des opérateurs par défaut.

```
class A {
public:
    void* operator new[](std::size_t);
    void* operator new[](std::size_t,unsigned x);
    void operator delete[](void*);
    void operator delete[](void*,std::size_t index);
    void operator delete[](void*,std::size_t index,bool flag);
};
```

```
// A *ptr = new A[10];
void* A::operator new[](std::size_t z){
    ::new A[x/sizeof(A)]; // sinon ::new char[z];
}
// A *ptr = new(99) A[10];
void* A::operator new[](std::size_t z,unsigned x){
    unsigned n = z/sizeof(A);
    A* op = ::new A[x/sizeof(A)];
    for (int i=0;i<n;i++)
    op[i].valeur = x; // valeur membre de A
}
// A *ptr = new A[10];
// delete[] ptr;
void operator delete[](void* p){
    ::delete[] reinterpret_cast<A * >(p);
}
// Ici « size » contient de facto la taille du bloc mémoire à
// libérer. Il est donc important de choisir entre cette
// version et celle précédemment décrite.
void operator delete[](void*,std::size_t index){
    ::delete[] reinterpret_cast<A * >(p);
}
// delete(p,100,true);
void operator delete[](void*,std::size_t index,bool flag);
};
```

-