

Allocation dynamique en C++



1. Types d'allocation

Trois types d'allocation en C++



En C++, les variables peuvent être allouées de 3 manières

- **Statique** - globales et static
- **Automatique** - locales et attributs de classe
- **Dynamique**

Trois types d'allocation



```
#include <iostream>
using namespace std;

class C {
public:
    C() { cout << "--> Constructeur\n"; }
    ~C() { cout << "--> Destructeur\n"; }
};

void f() {
    cout << "    f() : début\n";
    cout << "    f() : fin\n";
}

int main() {
    cout << "main() : début\n";
    cout << "    f() : avant\n";
    f();
    cout << "    f() : après\n";
    cout << "main() : fin\n";
}
```

```
main() : début
f() : avant
f() : début
f() : fin
f() : après
main() : fin
```

- Les objets de type C affichent quand ils sont construits et détruits
- Illustrons ce qui se passe pour une variable ...
 - globale
 - statique de `f()`
 - locale de `f()`
 - allouée dynamiquement

Variable globale



```
C c; // variable globale

void f() {
    cout << "f() : début\n";
    cout << "f() : fin\n »;";
}

int main() {
    cout << "main() : début\n";
    cout << "f() : avant\n";
    f();
    cout << "f() : après\n";
    cout << "main() : fin\n";
}
```

--> Constructeur
main() : début
f() : avant
f() : début
f() : fin
f() : après
main() : fin
--> Destructeur

- Allouée et construite avant `main()`
- Détruite et libérée après `main()`

Variable locale statique



```
void f() {  
    cout << " f() : début\n";  
    static C c;  
    cout << " f() : fin\n";  
}  
  
int main() {  
    cout << "main() : début\n";  
    cout << " f() : avant\n";  
    f();  
    f();  
    cout << " f() : après\n";  
    cout << "main() : fin\n";  
}
```

```
main() : début  
f() : avant  
f() : début  
--> Constructeur  
f() : fin  
f() : début  
f() : fin  
f() : après  
main() : fin  
--> Destructeur
```

- Mémoire **allouée** au début du programme
- Variable **construite** au premier passage de exécution là où elle est déclarée
- Variable **détruite** et mémoire **libérée** après la sortie de `main()`

Variable locale statique (2)



```
void f(int i) {
    cout << " f(" << i << ") : début\n";
    if(i)
        static C c;
    cout << " f(" << i << ") : fin\n";
}

int main() {
    cout << "main() : début\n";
    cout << " f() : avant\n";
    f(0);
    f(1);
    f(2);
    cout << " f() : après\n";
    cout << "main() : fin\n";
}
```

main() : début
f() : avant
f(0) : début
f(0) : fin
f(1) : début
--> Constructeur
f(1) : fin
f(2) : début
f(2) : fin
f() : après
main() : fin
--> Destructeur

- Mémoire **allouée** au début du programme
- Variable **construite** au premier passage de exécution là où elle est déclarée
- Variable **détruite** et mémoire **libérée** après la sortie de **main()**

Variable locale automatique



```
void f() {  
    cout << "  f() : début\n";  
    C c;  
    cout << "  f() : fin\n";  
}  
  
int main() {  
    cout << "main() : début\n";  
    cout << "  f() : avant\n";  
    f();  
    cout << "  f() : après\n";  
    cout << "main() : fin\n";  
}
```

main() : début
f() : avant
 f() : début
 --> Constructeur
 f() : fin
 --> Destructeur
 f() : après
main() : fin

- Mémoire **allouée** en entrant dans le bloc où elle est déclarée
- Variable **construite** au passage de l'exécution là où elle est déclarée
- Variable **détruite** et mémoire **libérée** en sortant du bloc où elle est déclarée, i.e. à l'accolade fermante }

Variable locale automatique (2)



```
void f() {
    cout << " f() : début\n ";
    { C c; }
    cout << " f() : fin\n";
}

int main() {
    cout << "main() : début\n";
    cout << " f() : avant\n";
    f();
    cout << " f() : après\n";
    cout << "main() : fin\n";
}
```

main() : début
f() : avant
f() : début
--> Constructeur
--> Destructeur
f() : fin
f() : après
main() : fin

- Mémoire **allouée** en entrant dans le bloc où elle est déclarée
- Variable **construite** au passage de l'exécution là où elle est déclarée
- Variable **détruite** et mémoire **libérée** en sortant du bloc où elle est déclarée, i.e. à l'accolade fermante }

Allocation dynamique



```
C* f() {  
    cout << "  f() : début\n";  
    C* p = new C;  
    cout << "  f() : fin\n";  
    return p;  
}  
  
int main() {  
    cout << "main() : début\n";  
    cout << "  f() : avant\n";  
    C* q = f();  
    cout << "  f() : après\n";  
    delete q;  
    cout << "main() : fin\n";  
}
```

main() : début
f() : avant
f() : début
--> Constructeur
f() : fin
f() : après
--> Destructeur
main() : fin

- Mémoire **allouée** et variable **construite** au passage de l'exécution sur l'instruction **new**
- Variable **détruite** et mémoire **libérée** au passage sur l'instruction **delete**

Allocation dynamique (2)



```
C* f() {
    cout << "  f() : début\n";
    C* p = new C;
    cout << "  f() : fin\n";
    return p;
}

void g(C* p) {
    cout << "  g() : début\n";
    delete p;
    cout << "  g() : fin\n";
}

int main() {
    cout << "main() : début\n";
    C* q = f();
    g(q);
    cout << "main() : fin\n";
}
```

main() : début
f() : début
--> Constructeur
f() : fin
g() : début
--> Destructeur
g() : fin
main() : fin

- Mémoire **allouée** et variable **construite** au passage de l'exécution sur l'instruction **new**
- Variable **détruite** et mémoire **libérée** au passage sur l'instruction **delete**
- Eventuellement dans une toute autre fonction



new / delete

- L'allocation dynamique utilise les instructions **new** et **delete**, dont la syntaxe de base pour un type T est

```
T* ident = new T;
```

```
delete ident;
```

- L'identifiant *ident* est un **pointeur** de type T*.

Initialisation



- Pour initialiser les variables automatiques, nous avions 3 syntaxes

```
int i1(42);  
int i2{42};  
int i3 = 42;
```

- 2 syntaxes permettent **d'initialiser** le contenu de la variable dynamique créée

```
int* p1 = new int(42);  
int* p2 = new int{42};
```

Initialisation (2)



- En C++11 (ou plus), privilégiez l'initialisation par accolade

```
struct s {  
    int a;  
    int b;  
};  
  
s* p1 = new s{1,2};  
  
s* p2 = new s(1,2);
```

```
struct s {  
    int a;  
    int b;  
    s(int _a, int _b)  
        : a{_a}, b{_b} {}  
};  
  
s* p1 = new s(1,2);  
s* p2 = new s{1,2};
```



delete

- Il faut appeler **delete** pour chaque variable créée par **new**. Sinon, on a une fuite de mémoire.
- N'effacer un pointeur qu'une seule fois.

```
int* p = new int(42);
delete p;
delete p;
```

malloc: *** error for object 0x1004000d0:
pointer being freed was not allocated

- On a le droit d'**effacer le pointeur nul**. C'est une no-op.

2. Pointeurs (rappels)

Les pointeurs



T* p;

- p est un pointeur vers un élément de type T
- p contient l'**adresse mémoire** où est stockée la variable de type T.
- On **déréfère** le pointeur en le précédent du symbole *.
- *p s'utilise comme une variable de type T.
- On obtient l'adresse d'un variable en la précédent du symbole &

Les pointeurs (2)



- Si T est un type composé (**struct** ou **class**), les deux syntaxes suivantes sont synonymes

(*p) . m

p -> m

- On peut incrémenter / décrémenter un pointeur
- On peut ajouter / retirer un entier à un pointeur
- Le pointeur nul s'appelle **nullptr**

Influence du type pointé (1)



- Le pointeur stocke une adresse en mémoire
- Le type de pointeur influence la manière dont les données binaires sont **interprétées**
- p et q pointent sur la même adresse 0x7ff637c05930
- Les 16 bits commençant à cette adressant sont interprétés comme -1 ou 65535 selon le type

```
short *p = new short{-1};  
  
cout << "p : " << p << endl;  
cout << "*p : " << *p << endl;
```

p : 0x7ff637c05930
*p : -1

```
unsigned short* q =  
reinterpret_cast<unsigned short*>(p);
```

```
cout << "q : " << q << endl;  
cout << "*q : " << *q << endl;
```

q : 0x7ff637c05930
*q : 65535

Influence du type pointé (2)



- Les pointeurs disposent d'un `operator+(int d)` qui décale de `d` « *positions* » en mémoire
- Pour un pointeur `T*` `p`, l'adresse physique en mémoire de `p+d` est décalée de `sizeof(T)*d` par rapport à celle de `d`

```
short *p1 = nullptr; // nullptr = 0x0
cout << "short* : " << (p1+1) << endl;
```

short* : 0x2

```
int *p2 = nullptr;
cout << "int* : " << (p2+1) << endl;
```

int* : 0x4

```
double *p3 = nullptr;
cout << "double* : " << (p3+1) << endl;
```

double* : 0x8

```
string *p4 = nullptr;
cout << "string* : " << (p4+1) << endl;
```

string* : 0x18

Pointeurs constants



- On peut stocker un entier dans une variable ou un constante avec le mot clé **const**

```
int i0;  
int const i1;
```

- Pour un pointeur vers un entier, **const** peut s'appliquer
 - soit à l'adresse stockée dans le pointeur
 - soit à l'entier pointé
 - soit aux deux

```
int* p0;  
int const* p1;  
int* const p2;  
int const* const p3;
```



Effet de const

- **const** s'applique à ce qui le précède. Tout ce qui est constant doit être initialisé. Avec **int i;**

```
int* p0;  
p0 = &i;  
*p0 = i;
```

```
int const * p1;  
p1 = &i;  
*p1 = i;
```

```
int * const p2 = new int;  
p2 = &i;  
*p2 = i;
```

```
int const * const p3 = new int(42);  
p3 = &i;  
*p3 = i;
```

- Pour le type pointé, on peut placer le mot clé **const** avant ou après

3. Tableaux

Tableaux en C++



- Pour rappel, en C++ on utilise de préférence
 - La classe `std::array<T, n>` pour les petits tableaux de taille n connue à la compilation
 - La classe `std::vector<T>` pour les grands tableaux, ou pour ceux dont la taille n'est connue qu'à l'exécution, ou varie
- Mais il est aussi possible de travailler avec des **tableaux bruts** - i.e. blocs de mémoire stockant les éléments dans des emplacements consécutifs – et d'accéder aux éléments via des pointeurs.
- Ces tableaux bruts peuvent être alloués statiquement, automatiquement, dynamiquement



Tableaux « à la C »

- Pour déclarer un tableau brut de N éléments de type T, on écrit

T ident[N];

- N doit être connu à la compilation
 - Exemple :

Initialisation



- On peut initialiser un tableau avec un agrégat contenant au plus le nombre d'éléments du tableau (comme pour `std::array<T,n>`).
- S'il est plus court que le tableau, celui-ci est complété par des éléments initialisés par défaut
- Cela permet également de déclarer des tableaux constants

```
unsigned tab[5]{2, 5, 3, 1, 4}; // contient [2,5,3,1,4]

const int c_tab[4]{2,0,2,4};    // tableau constant, contient [2,0,2,4]

int tab2[4]{1,2};             // contient [1,2,0,0]

const double c_tab2[3]{};      // contient [0.,0.,0.]
```

Tableau et pointeur



- Un tableau brut se comporte essentiellement comme un pointeur constant vers le premier de ses éléments.

```
unsigned tab1[5]{2, 5, 3, 1, 4};  
unsigned * const p_tab1 = tab1;  
  
const int tab2[4]{2,0,2,4};  
const int * const p_tab2 = tab2;
```

- Mais, dans la fonction où il est déclaré, le compilateur peut nous donner son nombre d'éléments via l'opérateur `sizeof`, contrairement aux pointeurs

```
size_t N = sizeof tab / sizeof(int);  
// N vaut 5 (nombre d'éléments de tab)  
// sizeof tab vaut 20 si int est codé sur 32 bits  
  
size_t M = sizeof p_tab;  
// M vaut 8 sur un système 64 bits
```

Tableau en paramètre



- Passer un tableau en paramètre de fonction utilise une syntaxe différente des autres types de variable. Vous l'étudierez en PRG2
- En C++, le plus simple consiste à utiliser `std::span`
- A défaut, vous pouvez utiliser
 - un pointeur pour les données
 - un entier non-signé pour la taille du tableau
 - bref, les données stockées dans la structure d'un `std::span` ...



new[] / delete[]

- Il est aussi possible **d'allouer dynamiquement un tableau** de N variables de type T qui seront placées consécutivement en mémoire.

```
 $T^* \ ident = \ new \ T[N];$ 
```

```
 $\text{delete}[\ ] \ ident;$ 
```

Initialisation



- On peut initialiser un tableau créé dynamiquement avec les accolades

```
int* p1 = new int[6]{1,2,3,4,5,6};  
cout << p1[3] << endl; // affiche 4
```

```
int* p2 = new int[6]{};  
cout << p2[3] << endl; // affiche 0
```

operator[]



- On accède au contenu $i^{\text{ème}}$ élément du tableau soit via $*(ident+i)$, soit via $ident[i]$

```
int* p = new int[42];
```

```
p[5] = 24;
```

```
*(p+5) = 24;
```

```
delete[] p;
```

delete []



- Il faut appeler **delete[]** pour chaque tableau créé par **new[]**.
- Ne pas mélanger **new** et **delete[]**, ni **new[]** et **delete**.

4. ::operator new
et new(p)

Possible avec new et delete ...



- Listes simplement chainées

```
template <typename T>
struct MaillonSimple {
    T valeur;
    MaillonSimple* suivant;
};
```

- Listes doublement chainées

```
template <typename T>
struct MaillonDouble {
    T valeur;
    MaillonDouble* suivant;
    MaillonDouble* precedent;
};
```

- Arbres quelconques

```
template <typename T>
struct NoeudArbre {
    T valeur;
    NoeudArbre* parent;
    NoeudArbre* aine;
    NoeudArbre* puine;
};
```

- Arbres binaires

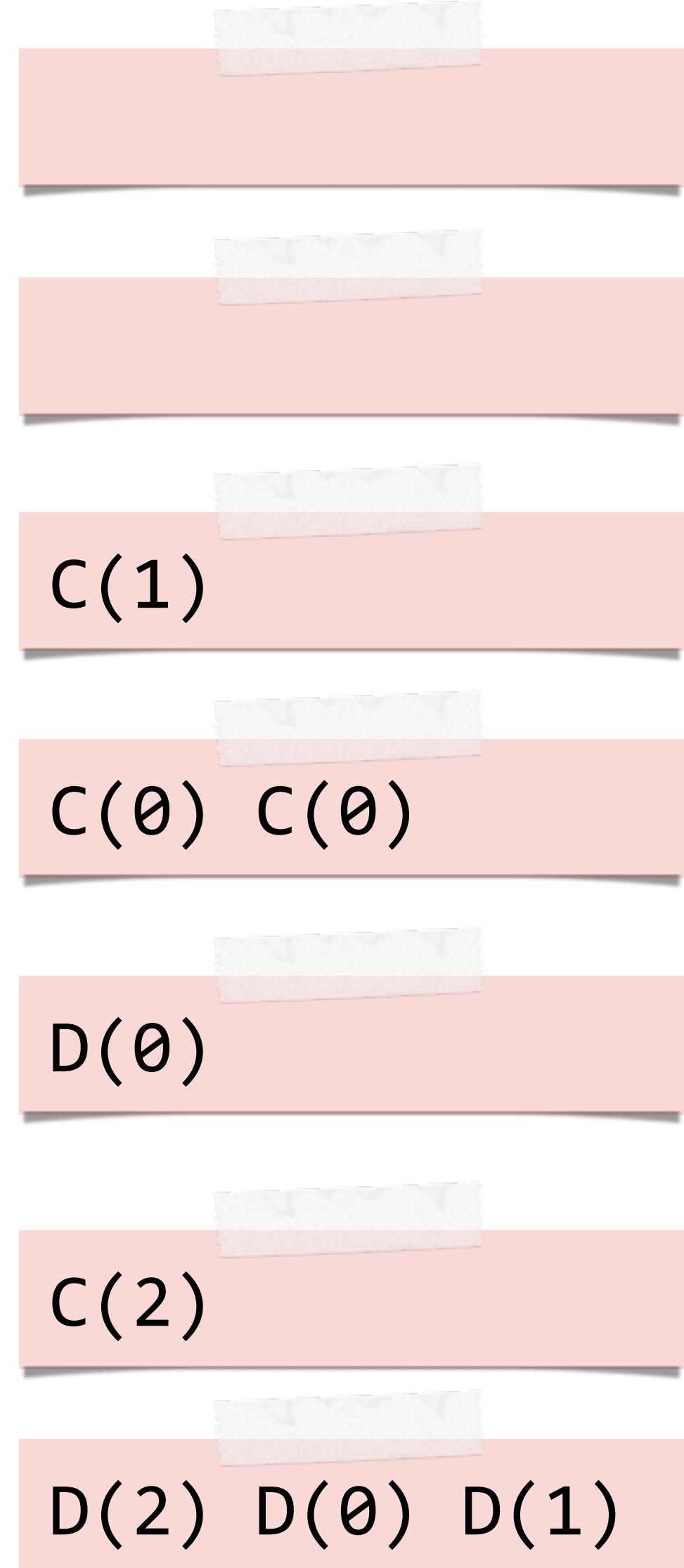
```
template <typename T>
struct NoeudArbreBinaire {
    T valeur;
    NoeudArbreBinaire* parent;
    NoeudArbreBinaire* gauche;
    NoeudArbreBinaire* droite;
};
```

Impossible avec new et delete ...

```
class C {
    int i;
public:
    C(int i = 0) : i(i) {
        cout << "C(" << i << ")";
    }
    ~C() {
        cout << "D(" << i << ")";
    }
};
```

- Dans std::vector, on distingue
 - allocation et construction
 - destruction et libération

```
int main() {
    vector<C> v;
    v.reserve(4);
    v.emplace_back(1);
    v.resize(3);
    v.resize(2);
    v.emplace_back(2);
}
```



Tout ce que fait new ...



Quand vous écrivez

```
T* p = new T{};
```

- **sizeof(T)** octets sont **alloués** en mémoire
- Si l'allocation échoue, une **std::bad_alloc()** exception est levée
- Sinon, un objet de type T est **construit** à l'emplacement mémoire alloué

Tout ce que fait new[] ...



Quand vous écrivez

```
T* p = new T[N];
```

- $N * \text{sizeof}(T)$ octets sont **alloués** en mémoire
- Si l'allocation échoue, une `std::bad_alloc()` exception est levée
- Sinon, N objets de type T sont **construits** à aux emplacements allant de p à $p+N-1$

Séparer allocation et construction



La ligne

```
T* p = new T{};
```

est équivalente aux deux lignes suivantes

```
void* p = ::operator new(sizeof(T)); // allocation mémoire
```

```
T* q = new(p) T{}; // construction objet de type T
```

Vous pouvez aussi utiliser la même variable pour les deux pointeurs. Eventuellement
caster explicitement les `void*` en `T*`

Ce que font `delete` et `delete[]`



Quand vous écrivez

```
delete p;           delete[] q;
```

- L'objet pointé par p est **détruit**; tous les objets dans le tableau q sont **détruits**
- La mémoire allouée à l'emplacement p ou pour le tableau q est **libérée**

Séparer destruction et libération



Pour un pointeur de type `T*`, la ligne

```
delete p;
```

est équivalente aux deux lignes suivantes

```
p->~T();           // destruction de l'objet de type T  
::operator delete(p); // libération de la mémoire
```

À partir de C++17, vous pouvez également remplacer `p->~T();` par

```
std::destroy_at(p);
```

5. Exceptions

Les exceptions levées par new



- Considérons l'instruction suivante

```
T* p = new T{};
```

- Pour rappel, elle est équivalente à ces 2 lignes

```
void* p = ::operator new(sizeof(T)); // allocation
```

```
T* q = new(p) T{}; // construction
```

- Elle peut donc lever 2 types d'exceptions
 - std::bad_alloc en cas de problème lors de l'allocation de mémoire
 - Toute exception levée par un constructeur de T est relayée

Eviter std::bad_alloc()



- On peut dire à ::operator new de ne pas lever d'exception

```
#include <new>
```

```
void* p = ::operator new(sizeof(T), std::nothrow);
```

- En cas de problème, elle retourne nullptr au lieu de lever bad_alloc

```
if(p == nullptr) { ... }
```

- On peut dire à l'instruction new d'utiliser cette version d'::operator new.

```
T* p = new (std::nothrow) T;
```

Eviter les exceptions de T()



- C'est impossible
- Au mieux vous pouvez les **catcher** et les traiter
- Le plus souvent, vous allez vous assurer de laisser votre classe en bon état (**garantie** faible ou forte) et **relayer** l'exception au niveau supérieur

Les exceptions levées par delete



- Aucune
- Tous les destructeurs sont déclarés **noexcept** par défaut et doivent être mis en œuvre ainsi
- Toute erreur lors d'un **delete** entraîne un crash brutal du programme non catchable.

Memory leaks...



Il y a 3 sources possibles de fuites

1. Affecter une autre valeur au seul (dernier?) pointeur contenant l'adresse de l'objet à détruire

```
int* p = new int(42);  
p = nullptr; // memory leak
```

2. Perdre ce pointeur en sortant de son scope

```
void f() {  
    int* p = new int(42);  
} // memory leak
```

Memory leaks...



3. Ne pas suivre le chemin d'exécution escompté en raison d'une exception levée

```
void f() {  
    int* p = new int(42);  
    g();      // peut lever  
    delete p; // ok si pas d'exception  
} // memory leak si g() lève
```

Comment éviter ces fuites ?

Comment éviter ces fuites?



- Toujours travailler avec **try et catch** ?

// *f peut nettoyer après g*

```
void f() {  
    auto p = new int(42);  
    try {  
        g(); // peut lever  
    } catch(...) {  
        // on nettoie  
    }  
    delete p;  
}
```

// *f ne peut pas nettoyer*
// *après g*

```
void f() {  
    auto p = new int(42);  
    try {  
        g(); // peut lever  
        delete p;  
    } catch(...) {  
        delete p;  
        throw;  
    }  
}
```

Comment éviter ces fuites?



Non ! Il vaut mieux utiliser le mécanisme d'exception pour qu'il appelle lui-même **delete**

- Une exception levée appelle les destructeurs des variables automatiques en sortant d'un scope
- Tous les **delete** nécessaires doivent donc être placés dans ces destructeurs
- Toute l'allocation dynamique doit être encapsulée dans des classes. Idéalement dans leur constructeur. Ce principe s'appelle **RAII** : Ressource Acquisition Is Initialisation

6. Allouer dans une classe

Motivation



<http://isocpp.github.io/CppCoreGuidelines/>

R.11: Avoid calling new and delete explicitly

Reason The pointer returned by new should belong to a resource handle (that can call delete). If the pointer returned by new is assigned to a plain/naked pointer, the object can be leaked.

Note In a large program, a naked delete (that is a delete in application code, rather than part of code devoted to resource management) is a likely bug: if you have N deletes, how can you be certain that you don't need N+1 or N-1? The bug may be latent: it may emerge only during maintenance. If you have a naked new, you probably need a naked delete somewhere, so you probably have a bug.



RandomString

- Pour illustrer ce concept, nous allons mettre en œuvre une classe RandomString
- Elle contient un tableau de N caractères aléatoires initialisés au constructeur

```
class RandomString {  
    size_t N;      // nombre de caractères  
    char* data;   // pointeur vers les données  
  
public:  
    RandomString(size_t n = 0) : N(n), data(nullptr) {  
        if (n != 0) {  
            data = new char[N];  
            for (size_t i = 0; i < N; ++i)  
                data[i] = 'A'+rand()%26;  
        }  
    }  
    ~RandomString() {  
        delete [] data;  
    }  
}
```

Copions ...



Considérons le code suivant.

```
{  
    RandomString rs1(10);  
    RandomString rs2 = rs1;  
} // La sortie de scope, c'est ici...
```

Que se passe-t-il en sortie de scope?

```
RandomString(41593,0x100088000) malloc:  
*** error for object 0x10060000:  
pointer being freed was not allocated
```



Pourquoi ?

- Ce qui se passe en détail...

```
{  
    RandomString rs1(10);      // constructeur(int)  
    RandomString rs2 = rs1;    // constructeur de copie  
} // destructeurs de rs1 et rs2
```

- Sans constructeur explicite, C++ en définit un implicitement équivalent à

```
RandomString(const RandomString& rs)  
    : data(rs.data) , N(rs.N) { }
```

- Et les destructeurs effacent deux fois le même data

```
~RandomString() { delete [] data; }
```

Copy constructor



- Chaque objet doit avoir sa propre copie des données, dont il est l'unique responsable
- Nous devons écrire notre propre constructeur de copie

```
RandomString(const RandomString& rs) {  
    data = new char[rs.N];  
    N = rs.N;  
    copy(rs.data, rs.data+N, data); // <algorithm>  
}
```



operator =

- Si on fournit un constructeur de copie, on doit aussi fournir un opérateur d'affectation : `RandomString& operator=(const RandomString&);`
- En plus de copier, il faut libérer les ressources précédentes ... et il ne suffit pas d'écrire

```
RandomString& operator=(const RandomString& rs) {  
    delete[] data;  
    data = new char[rs.N];  
    N = rs.N;  
    copy(rs.data, rs.data+N, data);  
    return *this;  
}
```

- Quid ...
 - si auto-affectation ?
 - si `this->N == rs.N` ?
 - si `new` lève une exception ?

Gérer l'auto-affectation



```
RandomString& operator=(const RandomString& rs) {  
    → if (this == &rs) return *this;  
  
    delete[] data;  
    data = new char[rs.N];  
    N = rs.N;  
  
    copy(rs.data, rs.data+N, data);  
  
    return *this;  
}
```

Ne pas réallouer si N identique



```
RandomString& operator=(const RandomString& rs) {  
    if (this == &rs) return *this;  
  
    if (rs.N != N) {  
        delete[] data;  
        data = new char[rs.N];  
        N = rs.N;  
    }  
  
    copy(rs.data, rs.data+N, data);  
  
    return *this;  
}
```

Garantie faible si new lève



```
RandomString& operator = (const RandomString& rs) {  
    if (this == &rs) return *this;  
  
    if (rs.N != N) {  
        delete[] data;  
        data = nullptr;  
        data = new char[rs.N];  
        N = rs.N;  
    }  
  
    copy(rs.data, rs.data+N, data);  
  
    return *this;  
}
```



Garantie forte si new lève



```
RandomString& operator = (const RandomString& rs) {  
    if (this == &rs) return *this;  
  
    if (rs.N != N) {  
        char* tmp = new char[rs.N];  
        // noexcept après cette ligne  
        delete[] data;  
        data = tmp;  
        N = rs.N;  
    }  
    copy(rs.data, rs.data+N, data);  
    return *this;  
}
```

copy & swap



- Si l'on ne voulait pas ré-utiliser `data` lorsque `rs.N == N`
- Il serait plus simple d'utiliser l'approche copy-and-swap
- En premier, définir une fonction `swap` échangeant efficacement (sans copier les données) le contenu de 2 `RandomString`

```
// méthode
void swap( RandomString& other) noexcept
{
    using std::swap;
    swap(this->N , other.N);
    swap(this->data , other.data);
}

// fonction annexe
void swap(RandomString& lhs,
           RandomString& rhs) noexcept
{
    lhs.swap(rhs);
}
```

copy & swap (2)



- Ensuite, ré-écrire l'affectation pour qu'elle copie dans une variable temporaire, puis swap cette variable temporaire avec *this
- Simple, et fournit une **garantie forte** en cas d'exception lors de la copie

```
RandomString& operator= (const RandomString& rs)
{
    if (this == &rs) return *this;
    RandomString tmp{rs}; // constructeur de copie
    //————— noexcept après cette ligne
    swap(tmp);           // swap
    return *this;
}
```