

Allocation dynamique en C++



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 p;
    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

Syntaxe de base



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'indentifiant *ident* est un **pointeur** de type T*.

new[] / delete[]



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

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

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



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 précédant 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 adresse 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;      int const * p1;      int * const p2 = new int;  
p0 = &i;        p1 = &i;          p2 = &i;  
*p0 = i;      *p1 = 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

```
const int * p1b;    const int * const p3b = new int(42);
```

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};
```

Initialisation (3)



- 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*^{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;
```

Le cas de zéro



- On a le droit d'allouer dynamiquement un tableau de taille nulle

```
int* p = new int[0];
delete[] p;
```

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

```
int* p = nullptr;
delete p; // ok
```



delete

- Il faut appeler **delete** pour chaque variable créée par **new**. Sinon, on a une fuite de mémoire.
- Il faut appeler **delete[]** pour chaque variable créée par **new[]**.
- Ne pas mélanger **new** et **delete[]**, ni **new[]** et **delete**.
- 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
```

**::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);  
  
}
```

C(1)

C(0) C(0)

D(0)

C(2)

D(2) D(0) D(1)

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);
```

Exceptions

Pourquoi s'y intéresser ?



- Comprendre l'interaction entre l'allocation dynamique et le mécanisme d'exceptions est essentiel (et unique) en C++
 - C n'a pas d'exceptions
 - Java, C#, ... n'ont pas d'instruction **delete**.
- Cette interaction est à deux directions
 - L'allocation dynamique (**new** ou **new[]**) génère des exceptions
 - Les exceptions affectent le flux d'exécution du programme entre **new** et **delete**.

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

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 qui contient un tableau de N caractères aléatoires initialisés au constructeur
- Tout d'abord, nous verrons comment copier un objet de cette classe
- Ensuite, nous verrons comment trier ces objets par ordre alphabétique
 - avec un tri à bulle (échange)
 - avec un tri par insertion (déplacement)



RandomString

```
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 0x100600000:  
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;  
}
```

Swap

Rappel : RandomString



```
class RandomString {  
    size_t N;      // nombre de caractères  
    char* data;   // pointeur vers les données  
  
public:  
    RandomString(size_t n = 0);  
    RandomString(const RandomString& rs);  
    RandomString& operator= (const RandomString& rs);  
    ~RandomString();  
};
```

Effectuons un tri à bulles...

```
int main() {
    vector<RandomString> v;

    for(int n : { 5, 3, 2, 5, 6, 5, 9 } )
        v.emplace_back(n);

    BubbleSort(v.begin(),v.end());

    for(const auto& rs : v)
        cout << rs << endl;
}
```





Pourquoi emplace_back ? (1)

```
class C {
public:
    C() { cout << "default\n"; }
    C(int i) { cout << "int\n"; }
    C(const C& c) { cout << "copy\n"; }
    C& operator = (const C& c) { cout << "operator=\n";
                                return *this; }
    ~C() { cout << "destructor\n"; }
};

int main() {
    size_t N = 2;
    vector<C> v(N); // N constructeurs
                      // par défaut
    for(int i = 0; i < N; ++i)
        v[i] = C(rand()); // affectations
}
```

default
default
int
operator=
destructor
int
operator=
destructor
destructor
destructor



Pourquoi emplace_back ? (2)

```
class C {
public:
    C() { cout << "default\n"; }
    C(int i) { cout << "int\n"; }
    C(const C& c) { cout << "copy\n"; }
    C& operator = (const C& c) { cout << "operator=\n";
                                  return *this; }
    ~C() { cout << "destructor\n"; }
};

int main() {
    size_t N = 2;
    vector<C> v;      // pas de construction
    v.reserve(N);     // pas de construction
    for (int i = 0; i < N; ++i)
        v.push_back(C(rand())); // construction du paramètre
                                // construction par copie
                                // destruction du paramètre
}
```

int
copy
destructor
int
copy
destructor
destructor
destructor



Pourquoi emplace_back ? (3)

```
class C {
public:
    C() { cout << "default\n"; }
    C(int i) { cout << "int\n"; }
    C(const C& c) { cout << "copy\n"; }
    C& operator = (const C& c) { cout << "operator=\n";
                                return *this; }
    ~C() { cout << "destructor\n"; }
};
```

int
int
destructor
destructor

```
int main() {
    size_t N = 2;
    vector<C> v;           // pas de construction
    v.reserve(N);          // pas de construction
    for (int i = 0; i < N; ++i)
        v.emplace_back(rand()); // construction en place
}
```

Opérateur d'affichage



```
ostream& operator<< (ostream& s,
                        const RandomString& r)
{
    for(auto p = r.data; p != r.data+r.N; ++p)
        s << *p;
    return s;
}
```

operator< : comparaison lexicographique



```
bool operator<(const RandomString& rhs) const noexcept
{
    auto& lhs = *this;

    size_t plusPetitN = std::min(lhs.N, rhs.N);

    for(size_t i = 0; i < plusPetitN; ++i) {
        if(lhs.data[i] < rhs.data[i]) return true;
        if(lhs.data[i] > rhs.data[i]) return false;
        // (lhs.data[i] == rhs.data[i]) continue;
    }

    return lhs.N < rhs.N;
}
```

Opérateurs de comparaison >, <=, >=



```
bool operator>(const RandomString& rhs) const noexcept {
    return rhs < *this;
}
```

```
bool operator<=(const RandomString& rhs) const noexcept {
    return not( rhs < *this );
}
```

```
bool operator>=(const RandomString& rhs) const noexcept {
    return not( *this < rhs );
}
```

Tri à bulles générique



```
template <typename Iterator>
void BubbleSort(Iterator first, Iterator last)
{
    size_t N = distance(first, last);
    for(size_t i = 1; i < N; ++i)
        for (auto j = first; j != prev(last, i); ++j)
            if (*next(j) < *j)
                swap(*j, *next(j));
}
```

Quelle est sa complexité?



```
template <typename Iterator>
void BubbleSort(Iterator first, Iterator last)
{
    if(first == last) return;
    size_t N = distance(first, last);
    for(size_t i = 1; i != N; ++i)
        for (auto j = first; j != prev(last, i); ++j)
            if (*next(j) < *j)
                swap(*j, *next(j));
}
```

- trier n RandomString
- de m caractères chacune
- $O(n^2)$ comparaisons
- $O(n^2)$ échanges
- quelle est la complexité d'un échange de m caractères ?



std::swap

```
template <typename T> void swap(T& lhs, T& rhs) {  
    T tmp = lhs;      // constructeur de copie  
    lhs = rhs;        // opérateur d'affectation  
    rhs = tmp;        // opérateur d'affectation  
}                      // destruction automatique de tmp
```

- allocation dynamique d'un tableau de m chars
- 3 copies de m chars
- destruction d'un tableau de m chars

 $O(m)$

swap(RandomString&) en O(1)



- Méthode

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

- Fonction

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



Bénéfices

- swap est défini noexcept
- Les complexités s'améliorent

swap	$O(m)$	$O(1)$
BubbleSort	$O(m \cdot n^2)$	$O(n^2)$
std::sort	$O(m \cdot n \cdot \log(n))$	$O(n \cdot \log(n))$

- En effet, toute manipulation des données dans std::sort s'effectue via **using std::swap; swap(a,b);**
- <http://en.cppreference.com/w/cpp/concept/Swappable>



copy & swap

```
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;
}
```

- Forme canonique de l'opérateur d'affectation.
- Fournit une garantie forte en cas d'exception lors de la copie
- A utiliser systématiquement, sauf si d'autres optimisations sont voulues / nécessaires

move

Rappel : RandomString



```
class RandomString {  
    size_t N;      // nombre de caractères  
    char* data;   // pointeur vers les données  
public:  
    RandomString(size_t n = 0);  
    RandomString(const RandomString& rs);  
    RandomString& operator= (const RandomString& rs);  
    ~RandomString();  
  
    bool operator<(const RandomString& rhs) const noexcept;  
};  
  
ostream& operator<< (ostream& s, const RandomString& r);  
void swap(RandomString& lhs, RandomString& rhs) noexcept;
```

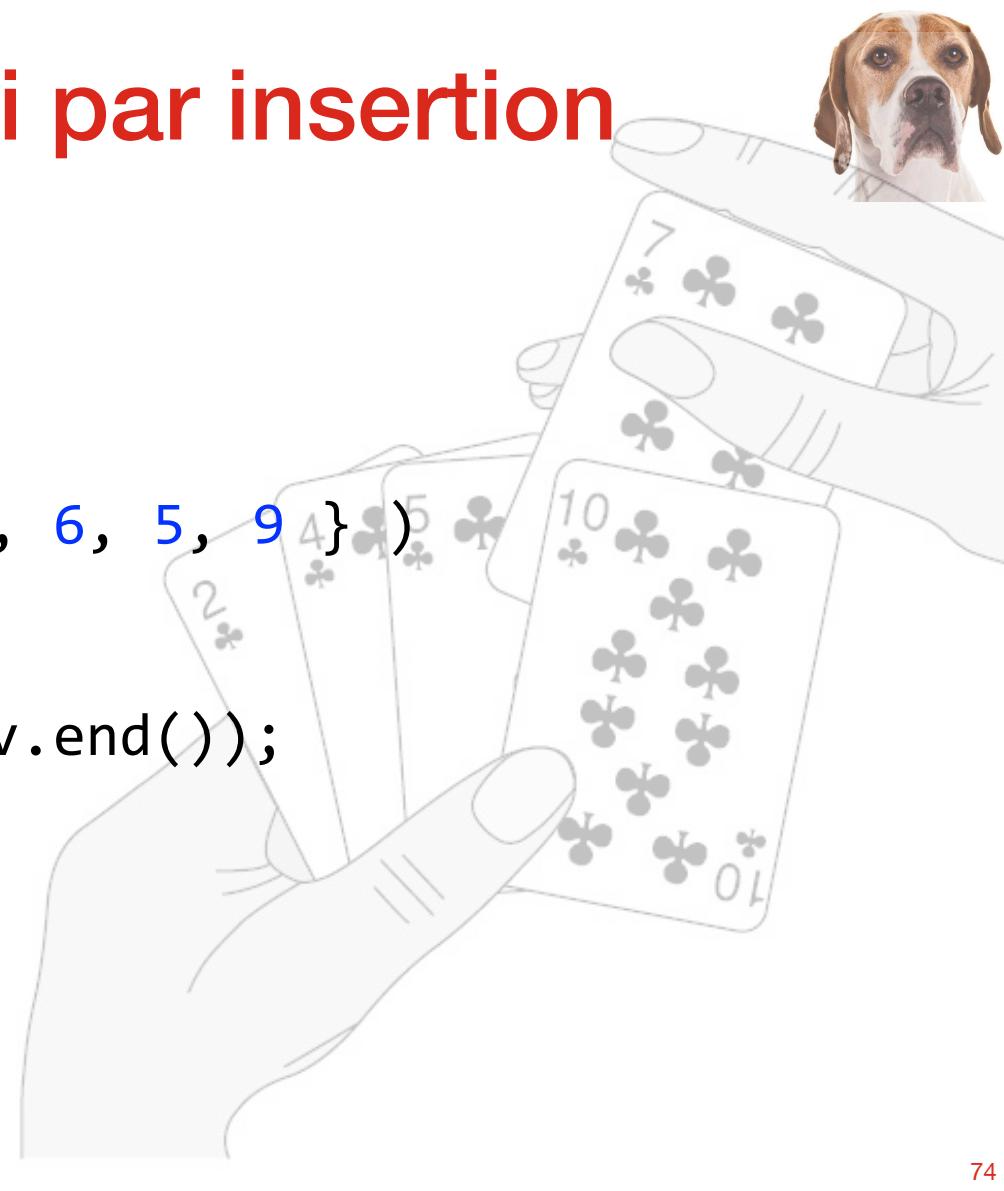
Effectuons un tri par insertion

```
int main() {
    vector<RandomString> v;

    for(int n : { 5, 3, 2, 5, 6, 5, 9 })
        v.emplace_back(n);

    InsertionSort(v.begin(),v.end());

    for(const auto& rs : v)
        cout << rs << endl;
}
```





Tri par insertion

```
template <typename Iterator> // bidirectionnal iterator
void InsertionSort(Iterator first, Iterator last)
{
    if (first == last) return;

    for (Iterator i = next(first); i != last; ++i)
    {
        auto tmp = *i;                      // constructeur de copie
        Iterator j = i;
        while (j != first and tmp < *prev(j))
        {
            *j = *prev(j);                  // operator=
            --j;
        }
        *j = tmp;                          // operator=
    }
}
```

Comment éviter ces copies?



- Avant C++11 ...
 - Réécrire l'algorithme avec swap
 - Trier des tableaux de pointeurs
- Depuis C++11 ...
 - utiliser la notion de déplacement avec `b = std::move(a);`
 - qui transfère les ressources de a à b, en laissant a dans un état indéterminé mais valide

Tri par insertion avec déplacement



```
template <typename Iterator> // bidirectionnal iterator
void InsertionSort(Iterator first, Iterator last)
{
    if (first == last) return;
    for (Iterator i = next(first); i != last; ++i)
    {
        auto tmp = std::move(*i);           // constructeur de déplacement
        Iterator j = i;
        while (j != first and tmp < *prev(j))
        {
            *j = std::move(*prev(j));     // affectation par déplacement
            --j;
        }
        *j = std::move(tmp);             // affectation par déplacement
    }
}
```

Que fait std::move?



- Regardons dans <utility>

```
static_cast<typename std::remove_reference<T>::type&&>(t)
```

- std::move caste son paramètre t de type T en *rvalue reference* T&&
 - Une expression en cours d'évaluation
 - Un valeur en retour de fonction
 - Le résultat d'un std::move...

Comment profiter de std::move ?



- Une rvalue reference n'utilisera plus ses ressources. On a le droit de les lui voler
- Pour cela, il faut écrire
 - Un constructeur de déplacement

```
RandomString(RandomString&& other);
```

- Un opérateur d'affectation par déplacement

```
RandomString& operator = (RandomString&& other);
```



Move constructor

- Prendre les ressources de other
- Annuler les ressources de other pour que son destructeur ne les casse pas.

```
RandomString(RandomString&& other) noexcept
    : N(other.N), data(other.data)
{
    other.data = nullptr;
    other.N = 0;
}
```

Move assignement operator



- Libérer les ressources de `*this`
- Prendre les ressources de `other`
- Annuler les ressources de `other`

```
RandomString& operator= (RandomString&& other) noexcept
{
    RandomString tmp = std::move(other);
    swap(tmp);
    return *this;
}                                     // tmp.~RandomString()
```

Bénéfices



- Construction et affectation par déplacement sont **noexcept**
- Les complexités s'améliorent

Construction & affectation	$O(m)$	$O(1)$
InsertionSort	$O(m.n^2)$	$O(n^2)$
std::stable_sort	$O(m.n.log(n))$	$O(n.log(n))$

- En effet, toute manipulation des données dans std::stable_sort s'effectue via std::move

<http://en.cppreference.com/w/cpp/concept/MoveConstructible>

<http://en.cppreference.com/w/cpp/concept/MoveAssignable>

std::swap



- La vraie définition de std::swap utilise la syntaxe de déplacement.
- std::swap<T> est noexcept si la syntaxe de déplacement est disponible pour le type T

```
template <class T>
void swap (T& a, T& b)
{
    T c(std::move(a));
    a=std::move(b);
    b=std::move(c);
}
```

```
template <class T> void swap (T& a, T& b)
noexcept (is_nothrow_move_constructible<T>::value
    && is_nothrow_move_assignable<T>::value);
```

Et encore ...

Vous n'avez pas tout vu...



- Présentation partielle du sujet
- Nécessaire et suffisante pour ASD
- Allocation dynamique uniquement dans les SD
 - offrent toujours des garanties faibles
 - offrent des garanties fortes quand c'est possible
- Allocation automatique seulement dans le code utilisant nos SD

Ce que vous n'utiliserez pas...



- Instructions C telles que `malloc`, `calloc`, `realloc`, `free`, ...
- Notion d'alignement
- `std::allocator` et `std::allocator_traits`
 - Utilisés dans les conteneurs STL comme couche intermédiaire
 - Fournissent des méthodes telles que `allocate`, `deallocate`, `construct`, `destroy`
 - Différent entre C++11, 14, 17 et 20.

Les smart pointers



- auto_ptr (déprécié C++11)
- unique_ptr, shared_ptr, weak_ptr (C++11 et ...)

```
{  
    std::unique_ptr<C> p(new C);  
} // p->~C() est appelé quand p sort de scope
```

```
std::unique_ptr<C> p(new C);  
std::unique_ptr<C> q = p; // ne compile pas  
std::unique_ptr<C> q = std::move(p);
```

- <http://en.cppreference.com/w/cpp/memory>

En pratique, nous utiliserons



- Listes et arbres :
- Allocation et construction combinées avec **new**
- Destruction et libération combinées avec **delete**
- garanties fortes
- Tableaux :
- Allocation et libération avec **::operator new** et **::operator delete**
- Construction et destruction avec **new()** et **std::destroy_at()**
- move semantics sur les éléments
- garanties fortes si possible, mais parfois faibles seulement