

Beste de savoir

Simplifier la gestion de la mémoire en
C++ avec RAII

4 septembre 2020

Table des matières

1.	Gestion manuelle de la mémoire	1
2.	L’idiome RAI à la rescousse	8
2.1.	Gestion des erreurs	10
2.2.	Un mot sur le <i>dispose pattern</i> ↗	12
3.	Exemples d’application avec la bibliothèque standard	12
3.1.	Cas particulier des pointeurs	13
4.	Bonnes pratiques	15
5.	La const-correctness	16
6.	Et dans les autres langages?	17
6.1.	Avec C	17
6.2.	Avec D	18
6.3.	Avec Rust	19

Tutoriel publié à l’origine sur [Progdupeupl](#) [↗](#)

La gestion des ressources est un problème récurrent en informatique. En effet, on ne dispose que de ressources limitées: RAM, disques durs, nombre de calculs par seconde, etc. Et aujourd’hui, il faut admettre qu’on charge de plus en plus de ressources qui prennent de la place. Il faut donc les gérer efficacement. Certains langages, comme le C, obligent l’utilisateur à allouer et libérer de la mémoire pour les ressources et il faut dire que c’est contraignant.

Le C++, en raison de l’approche historique qui en est malheureusement faite dans beaucoup d’ouvrages, est utilisé par certains développeurs comme le C, en gérant les ressources de manière manuelle. Pourtant, il existe un idiome très simple et efficace que nous allons découvrir dans ce tutoriel. Alors oubliez vos `new` et `delete` et découvrez ce que C++ vous offre.


Un grand merci à [Davidbrcz](#) [↗](#) pour son aide à la validation et à l’amélioration de ce tutoriel, ainsi que tous ceux qui ont relevé des fautes (mention spéciale à [Dominus Carnufex](#) [↗](#)).

1. Gestion manuelle de la mémoire

Bien souvent, dès qu’on manipule des ressources externes, du type image à charger et afficher, connexion à une base de données, ou à un serveur, ou autres, il est inévitable de devoir réserver de la mémoire de façon dynamique. Pour ceux qui ont fait du C, vous pensez sans doute **aux pointeurs** et vous avez bien raison. Prenons donc un bête exemple: on se connecte à une base de données, on récupère un nombre fixé de noms de trains, on ouvre un fichier, on le verrouille, on travaille ensuite dessus avant de tout refermer comme il se doit.

1. Gestion manuelle de la mémoire

```
1 void get_infos_from_db()
2 {
3     SGBD * sgbd = SGBD_Init("trains.db");
4     const int nb_trains = 2;
5
6     char** trains_name = malloc(nb_trains * sizeof(char*));
7     for (int i = 0; i < nb_trains; ++i)
8     {
9         char buffer[256];
10
11         trains_name[i] = malloc(42 * sizeof(char));
12         sprintf(buffer, "SELECT name FROM trains WHERE id = %d",
13                 i);
14         strcpy(trains_name[i], do_request(sgbd, buffer));
15     }
16
17     File * file = fopen("saved.txt");
18     Lock * lock = lock_acquire();
19
20     do_some_stuff(trains_name, file, lock);
21
22     lock_release(lock);
23     fclose(file), file = NULL;
24
25     for (int i = 0; i < nb_trains; ++i)
26     {
27         free(trains_name[i]), trains_name[i] = NULL;
28     }
29     free(trains_name), trains_name = NULL;
30
31     SGBD_release(sgbd);
32 }
```

Pourtant, ce code est juste une horreur à éviter. Pourquoi? Parce qu'aucune vérification n'est faite. Si une seule opération échoue, on est bon pour un *segfault*. Alors, sécurisons ce code (merci à [Taurre](#) )

```
1 void darray_delete(void ** self, unsigned n)
2 {
3     if (self != NULL)
4     {
5         unsigned i;
6         for (i = 0; i < n; ++i)
7         {
8             free(self[i]);
9         }
10        free(self);
11    }
```

1. Gestion manuelle de la mémoire

```
11     }
12 }
13
14 void ** darray_create(unsigned n, unsigned m, size_t size)
15 {
16     void ** self;
17     unsigned i;
18
19     assert(n != 0 && m != 0 && size != 0);
20     assert(SIZE_MAX / sizeof * self >= n);
21     assert(SIZE_MAX / m >= size);
22
23     self = malloc(n * sizeof * self);
24     if (self == NULL)
25     {
26         goto alloc_array_fail;
27     }
28
29     for (i = 0; i < n; ++i)
30     {
31         self[i] = malloc(m * size);
32         if (self[i] == NULL)
33         {
34             goto alloc_element_fail;
35         }
36     }
37
38     return self;
39
40 alloc_element_fail:
41     darray_delete(self, i);
42 alloc_array_fail:
43     return NULL;
44 }
45
46 #define NTRAINS 2
47 #define TRAIN_MAX 42
48 #define BUFFER_MAX 256
49
50 void get_infos_from_db(void)
51 {
52     SGBD * sgbd;
53     void ** trains_name;
54     FILE * file;
55     Lock * lock;
56     unsigned i;
57
58     sgbd = SGBD_Init("trains.db");
59     if (sgbd == NULL)
60     {
```

1. Gestion manuelle de la mémoire

```
61     write_error_log(SGBD_FAIL);
62     goto sgbd_fail;
63 }
64
65 trains_name = darray_create(NTRAINS, 1, TRAIN_MAX);
66 if (trains_name == NULL)
67 {
68     write_error_log(TRAIN_CREATE_FAIL);
69     goto darray_create_fail;
70 }
71
72 for (i = 0; i < NTRAINS; ++i)
73 {
74     char buffer[BUFFER_MAX];
75     int n;
76
77     n = snprintf(buffer, sizeof buffer,
78                 "SELECT name FROM trains WHERE id = %d", i);
79     if (n < 0 || n > sizeof buffer)
80     {
81         write_error_log(SNPRINTF_FAIL);
82         goto snprintf_fail;
83     }
84     strcpy(trains_name[i], do_request(sgbd, buffer));
85 }
86
87 file = fopen("saved.txt");
88 if (file == NULL)
89 {
90     write_error_log(FILE_FAIL);
91     goto fopen_fail;
92 }
93
94 lock = lock_acquire();
95 if (lock == NULL)
96 {
97     write_error_log(LOCK_FAIL);
98     goto lock_acquire_fail;
99 }
100
101 /*
102  * Stuff
103  */
104
105 lock_release(lock);
106
107 lock_acquire_fail:
108     fclose(file);
109
```

1. Gestion manuelle de la mémoire

```
110     fopen_fail:
111     snprintf_fail:
112         darray_delete(trains_name, NTRAINS);
113
114     darray_create_fail:
115         SGBD_release(sgbd);
116
117     sgbd_fail:
118         ;
119 }
120
121 #undef NTRAINS
122 #undef TRAIN_MAX
123 #undef BUFFER_MAX
```

Quelle plaie à écrire! Non seulement c'est long, mais en plus, c'est plus complexe à comprendre, on peut avoir oublié certains cas, bref, un cauchemar. Et encore, on aurait pu avoir à initialiser plus de ressources encore.

Peut-être certains d'entre vous pensent que `goto`, c'est un héritage du C dépassé, et qu'en C++ on devrait plutôt utiliser les exceptions. Soit, essayons.

```
1 void get_infos_from_db()
2 {
3     const int nb_trains = 2;
4     SGBD * sgbd;
5
6     try
7     {
8         sgbd = SGBD_Init("trains.db");
9     }
10    catch (sgbd_exception const & e)
11    {
12        write_error_log(e);
13        throw;
14    }
15
16    char** trains_name;
17    try
18    {
19        trains_name = new char*[nb_trains];
20    }
21    catch (std::bad_alloc const & e)
22    {
23        SGBD_release(sgbd);
24        write_error_log(e);
25        throw;
26    }
```

1. Gestion manuelle de la mémoire

```
27
28 int last_good_alloc_index = 0;
29 for (int i = 0; i < nb_trains; ++i)
30 {
31     char buffer[256];
32
33     try
34     {
35         trains_name[i] = new char[42];
36     }
37     catch (std::bad_alloc const & e)
38     {
39         for (int i = 0; i < last_good_alloc_index; ++i)
40         {
41             delete[] trains_name[i], trains_name[i] = NULL;
42         }
43
44         delete[] trains_name, trains_name = NULL;
45         SGBD_release(sgbd);
46         write_error_log(e);
47         throw;
48     }
49
50     last_good_alloc_index = i;
51
52     sprintf(buffer, "SELECT name FROM trains WHERE id = %d",
53             i);
54     strcpy(trains_name[i], do_request(sgbd, buffer));
55 }
56
57 File * file;
58 try
59 {
60     file = fopen("saved.txt");
61 }
62 catch (file_exception const & e)
63 {
64     for (int i = 0; i < nb_trains; ++i)
65     {
66         delete[] trains_name[i], trains_name[i] = NULL;
67     }
68
69     delete[] trains_name, trains_name = NULL;
70     SGBD_release(sgbd);
71     write_error_log(e);
72     throw;
73 }
74
75 Lock * lock;
76 try
```


1. Gestion manuelle de la mémoire

```
76     {
77         lock = lock_acquire();
78     }
79     catch (lock_exception const & e)
80     {
81         fclose(file), file = NULL;
82
83         for (int i = 0; i < nb_trains; ++i)
84         {
85             delete[] trains_name[i], trains_name[i] = NULL;
86         }
87
88         delete[] trains_name, trains_name = NULL;
89         SGBD_release(sgbd);
90         write_error_log(e);
91         throw;
92     }
93
94
95     do_some_stuff(trains_name, file, lock);
96
97     lock_release(lock);
98     fclose(file), file = NULL;
99
100    for (int i = 0; i < nb_trains; ++i)
101    {
102        free(trains_name[i]), trains_name[i] = NULL;
103    }
104    free(trains_name), trains_name = NULL;
105
106    SGBD_release(sgbd);
107 }
```

Finalement, ce code ne nous apporte aucun avantage par rapport au précédent: toujours aussi gros, toujours aussi illisible, et nous ne sommes même pas sûrs de couvrir tous les chemins possibles: un oubli est possible, une fonction apparemment inoffensive peut lancer une exception, bref, toujours un cauchemar à maintenir.

Que retenir jusque là: que la détection d'erreurs par retour de fonctions et `goto` ou par le biais d'exceptions nécessite d'ajouter des `if` ou des `try catch` toutes les deux lignes. En fait, dans ces cas de figure, chaque ligne où l'on acquiert une ressource qui n'est pas suivie d'un `if` ou entourée d'un `try catch` est suspecte et peut potentiellement faire échouer l'exécution.

Le cœur du problème tient en une phrase: le développeur doit écrire du code spécifique pour la libération de la mémoire et la gestion des erreurs. Pour améliorer la situation, il faut obligatoirement libérer le développeur de cette tâche, qu'elle soit automatique. Or, contrairement au C# ou au Java qui disposent d'un mécanisme de libération de la mémoire transparent et automatique appelé *garbage collector*, il n'est rien de tel en C++¹. Sommes-nous donc condamnés à devoir écrire des codes aussi lourds? Non, car une solution existe déjà.

1. Le C++ peut se voir doter d'un *garbage collector*. C'est quelque chose de prévu par la norme. Dans ce

2. L'idiome RAI à la rescousse

Le C++ propose un idiome particulier appelé **RAII**, pour *Resource Acquisition Is Initialization*, ce que l'on peut traduire par «acquisition de ressources lors de l'initialisation» en français. Comment fonctionne-t-il? Chaque ressource sera manipulée par une variable locale qui va l'acquérir à la construction et la libérer à la destruction. Ainsi, l'utilisateur n'aura même plus à se soucier d'appeler les fonctions `free`, `unlock` et autres `delete` pour que la libération des ressources ait bien lieu.

Pour appliquer cet idiome en C++, nous allons utiliser les classes et en particulier le couple constructeur(s) / destructeur. On peut parler de **capsules RAI**.

- **Toutes les ressources seront acquises dans le constructeur**; si des ressources sont impossibles à acquérir, on lève une exception. Ainsi, il n'y a pas de risque de créer un objet incomplet (*Ill formed* en anglais) donc pas de risque de fuite de mémoire: la norme garantit en effet que si un constructeur lève une exception, toute la mémoire des membres déjà allouée est libérée.
- **Toutes les ressources seront libérées dans le destructeur**. Celui-ci étant appelé automatiquement dès que l'objet est détruit, on y écrira tous les mécanismes de libération de la ressource acquise dans le constructeur.

i

Un constructeur ne peut acquérir, au maximum, qu'une seule ressource non encapsulée par un mécanisme RAI. La classe contenant pour ce constructeur devient alors une capsule RAI pour cette ressource.

Voyons sans plus tarder comment appliquer ce principe à notre code précédent. Commençons par encapsuler nos ressources dans des classes, en prenant par exemple le SGBD.

```
1 class Sgbd_Capsule
2 {
3     public:
4         Sgbd_Capsule(const char * db)
5         {
6             /*
7             Appels de méthodes, initialisations d'attributs,
8             etc.
9             */
10            this->m_sgbd = SGBD_Init(db);
11        }
12
13        ~Sgbd_Capsule()
14        {
15            /*
16            On libère les ressources allouées.
```

tutoriel, nous n'aborderons pas cette possibilité.

2. L'idiome RAII à la rescousse

```
17         */
18
19         SGBD_release(this->m_sgbd);
20     }
21
22     private:
23         SGBD * m_sgbd;
24 };
```

Maintenant, nous pouvons écrire du code aussi simple que celui ci-dessous (et nous verrons que nous pouvons faire encore plus simple dans la section suivante).

```
1 int foo()
2 {
3     Sgbd_Capsule sgbd("trains.db");
4
5     /*
6      Des opérations diverses sur le SGBD.
7     */
8
9     return 42;
10 }
```

Les ressources sont libérées à la sortie du bloc dans lequel nous les avons acquises, c'est-à-dire ici en sortant de la fonction. Voyez par vous-mêmes l'exemple suivant.

```
1 class Test
2 {
3     public:
4         Test(int number)
5             : m_number(number)
6             {
7                 std::cout << "Acquisition de la ressource n°" <<
8                     this->m_number << ".\n";
9             }
10        ~Test()
11        {
12            std::cout << "Libération de la ressource n°" <<
13                this->m_number << ".\n";
14        }
15    private:
16        int m_number;
17 };
18
```

2. L’idiome RAII à la rescousse

```
19 int main()
20 {
21     Test a(1);
22     {
23         Test b(2);
24         {
25             Test c(3);
26             Test d(4);
27         }
28     }
29     Test e(5);
30 }
31 return 0;
32 }
```

```
1 Acquisition de la ressource n°1.
2 Acquisition de la ressource n°2.
3 Acquisition de la ressource n°3.
4 Acquisition de la ressource n°4.
5 Libération de la ressource n°4.
6 Libération de la ressource n°3.
7 Acquisition de la ressource n°5.
8 Libération de la ressource n°5.
9 Libération de la ressource n°2.
10 Libération de la ressource n°1.
```

2.1. Gestion des erreurs

Il reste néanmoins un problème que nous ne gérons pas encore: que fait-on si une erreur survient lors de l’acquisition ou de la libération des ressources? Examinons chacun des cas.

2.1.1. Erreur lors de l’acquisition

Si on ne peut acquérir une ressource, alors l’objet ne peut être construit. Le mieux est donc de lancer une exception.

Lors de la construction d’un objet, si jamais le constructeur lance une exception, alors toute la mémoire réservée pour les membres sera libérée. Si jamais le constructeur a alloué de la mémoire dynamiquement de quelque manière que ce soit, alors cette dernière n’est pas libérée.

```
1 class Sgbd_Capsule
2 {
3     public:
```

2. L'idiome RAI à la rescousse

```
4     Sgbd_Capsule(const char * db)
5     {
6         /*
7         Appels de méthodes, initialisations d'attributs,
8         etc.
9         */
10        this->m_sgbd = SGBD_Init(db);
11        if (this->m_sgbd == nullptr)
12        {
13            throw
14                sgbd_exception("Le SGBD ne peut être initialisé.");
15        }
16    }
17    ~Sgbd_Capsule()
18    {
19        /*
20        On libère les ressources allouées.
21        */
22        SGBD_release(this->m_sgbd);
23    }
24
25    private:
26        SGBD * m_sgbd;
27 };
```

Enfin, un conseil important que je répète: si on a plusieurs ressources à acquérir dans un même constructeur, il vaut mieux que chaque ressource soit encapsulée dans sa propre capsule RAI; ainsi, chaque ressource sera libérée par son propre destructeur et on s'évite bien des soucis.

2.1.2. Erreur lors de la destruction

Ces cas-là sont problématiques. En effet, il est impossible de lancer une exception. Pourquoi? Nous savons que le destructeur d'un objet sera appelé si une exception est lancée dans le code; or, si le destructeur lance lui aussi une exception, nous nous retrouvons avec deux exceptions sur les bras, ce qui provoque un appel à la fonction `terminate()` et donc l'arrêt brutal du programme. De même, n'appellez jamais de fonctions dans le destructeur qui sont susceptibles de lancer des exceptions.

On peut néanmoins utiliser un système de logs pour informer l'utilisateur qu'une erreur dans la libération des ressources est arrivée. Quant à savoir si l'on continue l'exécution ou s'il vaut mieux tout arrêter, c'est à vous de voir en fonction des situations.

3. Exemples d'application avec la bibliothèque standard

2.2. Un mot sur le *dispose pattern* [↗](#)

Peut-être venez-vous d'un langage où il existe un mot-clef `finally`, utilisé à la suite d'un `try catch` et exécuté peu importe si une exception a été attrapée ou non; ou bien existe-t-il des constructions similaires du type `using` (C#), `with` (Python) ou encore *try-with-resources* (Java 7+). Dans tous les cas, le but est le même: empêcher des fuites de mémoire en libérant des ressources précédemment allouées. C'est ce qu'on appelle le *dispose pattern* [↗](#).

Pourtant, C++ ne fournit pas de mot-clef ou de construction similaire à celles de Java ou C# pour la simple et bonne raison que RAI nous permet de faire la même chose de façon plus efficace. Qu'est-ce qui me permet de dire ça? Je laisse le créateur du C++ [répondre](#) [↗](#).

In a system, we need a "resource handle" class for each resource. However, we don't have to have an "finally" clause for each acquisition of a resource. In realistic systems, there are far more resource acquisitions than kinds of resources, so the "resource acquisition is initialization" technique leads to less code than use of a "finally" construct.

Bjarne Stroustrup

Dans un système, il faut une "capsule RAI" pour chaque ressource. Cependant, nous n'avons pas besoin d'une clause "finally" pour chaque acquisition de ressource. Dans des systèmes réalistes, il y a beaucoup plus d'acquisitions de ressources que de types de ressources, donc le RAI conduit à écrire moins de code que l'utilisation d'une construction avec "finally".

Traduction libre.

3. Exemples d'application avec la bibliothèque standard

La bibliothèque standard utilise énormément cet idiome, à travers des noms qui vous sont certainement familiers: `std::string`, `std::array`, `std::vector`, `std::ifstream`, etc. Quand on y réfléchit, a-t-on déjà libéré manuellement un `std::string`? Non, car c'est fait automatiquement pour nous. Et pour vous montrer à quel point la bibliothèque standard est infiniment supérieure à tout ce qu'on pourrait faire manuellement, reprenons notre code de début en utilisant les mécanismes standards.

```
1 void get_infos_from_db()
2 {
3     const int nb_trains = 2;
4     Sgbd_Capsule sgbd("trains.db");
5
6     std::vector<std::string> trains_names;
7     for (int i = 0; i < nb_trains; ++i)
8     {
9         std::string buffer = "SELECT name FROM trains WHERE id = "
10        + std::to_string(i);
11        trains_names.push_back(do_request(sgbd, buffer));
```

3. Exemples d'application avec la bibliothèque standard

```
11     }
12
13     std::ifstream file("saved.txt");
14     Lock * lock = lock_acquire();
15
16     do_some_stuff(trains_name, file, lock);
17
18     lock_release(lock);
19 }
```

N'est-ce pas plus clair à lire et à comprendre? Premier point à retenir: **toujours utiliser au maximum la bibliothèque standard**. Pourquoi se frustrer à faire un code comme on ferait en C quand on peut profiter de mécanismes éprouvés, performants et sûrs comme ceux proposés par la bibliothèque standard? Donc faites-y appel le plus possible, ce sera du temps et du confort de gagnés.

3.1. Cas particulier des pointeurs

Notre code n'est pas encore tout à fait satisfaisant. En effet, il reste un pointeur. Or, les pointeurs nus sont source de beaucoup de problèmes en C++. Et si on pouvait ne pas avoir à écrire `Sgbd_Capsule`, ce serait encore mieux. Heureusement, la bibliothèque standard arrive encore une fois à notre secours en fournissant des **pointeurs intelligents** qui nous libèrent des contraintes de libération que l'on connaît si bien en C.

La norme C++11 nous propose plusieurs types de pointeurs intelligents:

- `std::auto_ptr`: déprécié, à ne plus utiliser;
- `std::unique_ptr`: comme son nom l'indique, à utiliser quand on ne veut avoir qu'un seul pointeur sur un objet;
- `std::shared_ptr`: utilise un système de comptage de références qui permet que plusieurs pointeurs pointent un même objet, ce dernier étant libéré quand le dernier pointeur pointant dessus est détruit;
- `std::weak_ptr`: si l'on n'y prend pas garde, les `std::shared_ptr` peuvent entraîner un problème de références circulaires (lisez donc [cet article de Developpez](#) qui illustre ce problème). Il sert également dans le cas d'une ressource avec plusieurs observateurs non propriétaire. Je vous invite à lire [cet article](#) pour des explications plus approfondies sur lequel choisir.

Nous avons également deux *templates* bien pratiques:

- `std::make_shared<T>`: construit un objet T et le met dans un `std::shared_ptr` (disponible avec C++11);
- `std::make_unique<T>`: construit un objet T et le met dans un `std::unique_ptr` (disponible avec C++14, voir [ici](#) pour une implémentation en C++11).

Ces *templates* sont à utiliser le plus possible car ils permettent d'écrire un code *exception-safe*. Lisez [l'article](#) de Herb Sutter à ce propos.

3. Exemples d'application avec la bibliothèque standard

Et en plus, le mieux du mieux, on peut définir des *deleters*, c'est-à-dire définir comment le pointeur va libérer sa ressource. Il suffit simplement de créer une classe sur ce modèle que l'on passera ensuite en argument à notre pointeur intelligent.

```
1 class Deleter
2 {
3     public:
4         template <typename T>
5         void operator()(T * ptr) const
6         {
7             /* Opérations diverses pour libérer la ressource. */
8         }
9 };
```

Et comme un exemple vaut mille explications, utilisons ce principe avec notre SGBD et notre mécanisme de verrouillage qui se prêtent bien au jeu. Mais comme rien n'est parfait, les fonctions `std::make_shared<T>` et `std::make_unique<T>` ne prennent pas de *deleter* en argument. Il nous faut passer par la construction classique.

```
1 class SGBD_deleter
2 {
3     public:
4         void operator()(SGBD * sgbd) const
5         {
6             SGBD_release(sgbd);
7         }
8 };
9
10 class Lock_deleter
11 {
12     public:
13         void operator()(Lock * lock) const
14         {
15             lock_release(lock);
16         }
17 };
18
19 void get_infos_from_db()
20 {
21     const int nb_trains = 2;
22     std::unique_ptr<SGBD, SGBD_deleter> sgbd
23         {SGBD_Init("train.db"), SGBD_deleter()};
24
25     std::vector<std::string> trains_names;
26     for (int i = 0; i < nb_trains; ++i)
27     {
```


4. Bonnes pratiques

```
27     std::string buffer = "SELECT name FROM trains WHERE id = "
28         + std::to_string(i);
29     trains_names.push_back(do_request(sgbd, buffer));
30 }
31 std::ifstream file("saved.txt");
32 std::unique_ptr<Lock, Lock_deleter> lock {lock_acquire(),
33     Lock_deleter()};
34 do_some_stuff(trains_name, file, lock);
35 }
```

Les pointeurs intelligents nous permettent également d'éviter le problème du constructeur qui alloue lui-même de la mémoire que nous avons vu dans la section précédente. En effet, les pointeurs intelligents seront bien libérés même si l'on rencontre une exception. Donc utilisez-les dès que vous pouvez, quitte à réécrire une version fonctionnelle des pointeurs intelligents ou utiliser Boost si vous ne pouvez pas compiler en C++11 / C++14.

Deuxième point à retenir: chaque fois qu'il est nécessaire d'utiliser des pointeurs, **utilisez des pointeurs intelligents**. Les cas où vous devrez obligatoirement utiliser des pointeurs nus sont très rares, alors utilisez la solution la plus confortable.

4. Bonnes pratiques

L'idéal, quand on gère des ressources, est de les **libérer dès que possible**. Non seulement cela est obligatoire dans certains cas (afin de ne pas faire attendre un processus trop longtemps pour ouvrir un fichier par exemple), mais en plus cela permet de soulager le système. Comment traduire cette bonne pratique en utilisant l'idiome RAI? Eh bien, il faut que l'on détruise nos objets s'occupant des ressources le plus vite possible, ce qui est possible en utilisant des blocs d'instructions.

```
1  int value;
2
3  { // Début du bloc d'instructions.
4      std::ifstream f("test.txt");
5      if (f.is_open()) {
6          f >> value;
7      }
8  } // Fin du bloc d'instruction : appel du destructeur du fichier.
9
10 value = value * 4;
```

Il s'agit d'une pratique courante que vous pourrez voir dans certains codes. Et bien entendu, le corolaire: **ne déclarez vos objets que quand vous en avez besoin** et pas avant. Alors oubliez les réflexes du C89 qui consistent à déclarer toutes les variables au début d'un bloc et ne le faites que pour un usage immédiat (sauf exception).

5. La const-correctness

Ce n'est pas une bonne pratique spécifique au RAII, mais dès qu'une ressource est censée être constante, alors il faut impérativement utiliser le mot-clef `const`. Cela donne des garanties à l'utilisateur et, couplé avec des références, permet un passage en argument plus rapide.

```
1 void bar(std::string const & data)
2 {
3     // Ici, on est certain que data ne sera pas modifiée.
4     // On utilise le passage par référence pour éviter une copie
5     // inutile.
6 }
```

D'ailleurs, petite astuce (merci [Herb Sutter](#) [↗](#)), si l'on veut déclarer un objet constant alors que ses paramètres dépendent de conditions, on peut y arriver grâce aux lambdas.

```
1 #include <iostream>
2
3 class Test
4 {
5     public:
6         Test(int number)
7             : m_number(number)
8             {
9             }
10
11         int number() const
12         {
13             return this->m_number;
14         }
15
16     private:
17         int m_number;
18 };
19
20 int main()
21 {
22     const Test a(1);
23
24     const Test f = [&]
25     {
26         Test f = Test(6);
27         if (a.number() == 1)
28         {
29             f = Test(42);
30         }
31     };
32 }
```

6. Et dans les autres langages?

```
31
32     return f;
33 }();
34
35     std::cout << "Valeur de f : " << f.number() << std::endl;
36
37     return 0;
38 }
```

```
1 Valeur de f : 42
```

6. Et dans les autres langages?

Bien que le C++ ait été le précurseur et le plus grand utilisateur de l’idiome RAII, aujourd’hui, il n’est plus le seul. D’autres langages permettent, par des moyens assez similaires, d’utiliser une sorte de RAII.

6.1. Avec C

Bien que cette possibilité soit offerte par une extension de GCC et donc non standard, elle mérite le détour et peut être intéressante pour ceux dont les applications ne seront compilées que par GCC. Il s’agit de l’attribut `cleanup`. Voici un exemple tiré de la [page](#) [Wikipédia](#) consacrée au RAII.

```
1 static inline void fclosep(FILE ** fp)
2 {
3     if (*fp)
4         fclose(*fp);
5 }
6 #define _cleanup_fclose_ __attribute__((cleanup(fclosep)))
7
8 void example_usage()
9 {
10     _cleanup_fclose_ FILE * logfile = fopen("logfile.txt", "w+");
11     fputs("hello logfile !", logfile);
12
13     /* logfile est correctement fermé sans appel explicite à fclose
14        */
14 }
```

6. Et dans les autres langages?

6.2. Avec D

Le D fournit trois méthodes pour permettre la libération des ressources, dont une identique à celle utilisée en C++: le couple constructeur / destructeur d'une classe. Les exemples suivants sont tirés du [site officiel](#) .

```
1 class Lock
2 {
3     Mutex m;
4
5     this(Mutex m)
6     {
7         this.m = m;
8         lock(m);
9     }
10
11    ~this()
12    {
13        unlock(m);
14    }
15 }
16
17 void abc()
18 {
19     Mutex m = new Mutex;
20     auto l = scoped!Lock(new Lock(m));
21     foo();
22 }
```

La seconde façon se rapproche de celle de Java avec un `try finally`.

```
1 void abc()
2 {
3     Mutex m = new Mutex;
4     lock(m); // lock the mutex
5     try
6     {
7         foo(); // do processing
8     }
9     finally
10    {
11        unlock(m); // unlock the mutex
12    }
13 }
```

Enfin, il existe une troisième méthode, originale par rapport aux deux autres: `scope(exit)`. Tout le code qui sera placé après cette instruction sera exécuté peu importe si la fonction se

6. Et dans les autres langages?

termine normalement ou si une exception est lancée. Elle se décline également sous deux autres formes: `scope(failure)` où le code ne sera exécuté qu'en cas d'exception et `scope(success)` où le code sera exécuté en cas de déroulement normal. La [documentation](#) complètera mes explications.

```
1 void abc()
2 {
3     Mutex m = new Mutex;
4
5     lock(m);           // lock the mutex
6     scope(exit) unlock(m); // unlock on leaving the scope
7
8     foo();             // do processing
9 }
```

6.3. Avec Rust

Rust, langage développé par la fondation Mozilla, utilise le RAII de la même manière que C++. Et comme un code est plus parlant, voici celui tiré de [la page](#) consacrée au RAII avec Rust.

```
1 fn create_box() {
2     // Allocate an integer in the heap
3     let _function_box = box 3i;
4
5     // `_function_box` gets destroyed here, memory gets freed
6 }
7
8 fn main() {
9     // Allocate an integer in the heap
10    let _boxed_int = box 5i;
11
12    // new (smaller) scope
13    {
14        // Another heap allocated integer
15        let _short_lived_box = box 4i;
16
17        // `_short_lived_box` gets destroyed here, memory gets
18        freed
19    }
20
21    // Create lots of boxes
22    for _ in range(0u, 1_000) {
23        create_box();
24    }
```

6. Et dans les autres langages?

```
25     // `_boxed_int` gets destroyed here, memory gets freed
26 }
```

```
1 $ rustc raii.rs && valgrind ./raii
2 ==26873== Memcheck, a memory error detector
3 ==26873== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward
   et al.
4 ==26873== Using Valgrind-3.9.0 and LibVEX; rerun with -h for
   copyright info
5 ==26873== Command: ./raii
6 ==26873==
7 ==26873==
8 ==26873== HEAP SUMMARY:
9 ==26873==     in use at exit: 0 bytes in 0 blocks
10 ==26873==   total heap usage: 1,013 allocs, 1,013 frees, 8,696
    bytes allocated
11 ==26873==
12 ==26873== All heap blocks were freed -- no leaks are possible
13 ==26873==
14 ==26873== For counts of detected and suppressed errors, rerun with:
    -v
15 ==26873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2
    from 2)
```

Nous voilà arrivés à la fin de ce tutoriel qui, je l'espère, vous en aura appris un peu plus sur C++. Bien entendu, le RAII n'est pas parfait: le pire qui puisse arriver est une erreur dans le destructeur. Mais hormis ces cas critiques, c'est un idiome particulièrement pratique et puissant, alors usez-en et abusez-en!