Pourquoi le RAII est-il fondamental en C++ ?

En programmation “moderne”, on recherche souvent l'efficacité dans l'écriture de code. Or, l'expérience le montre, la gestion manuelle de la mémoire est une source potentielle importante de perte de temps, aussi bien lors de l'écriture du code que lors des phases de débogage ou de validation du code. Le RAII est une approche permettant d'apporter des garanties plus fortes sur la gestion de la mémoire, libérant ainsi le développeur qui peut se consacrer à d'autres problématiques.

La situation en C

Imaginons le cas d'application suivant : on souhaite créer plusieurs tableaux de taille importante de manière sécurisée. C'est-à-dire que l'on souhaite ne pas avoir de fuite mémoire (allocation d'un bloc mémoire qui ne serait pas libéré).

On pourrait écrire le code suivant par exemple (peu importe les types concrets et l'implémentation des fonctions) :

```
code_erreur une_fonction() {
    Tableau x, y, z;

    x = créer_gros_tableau();
    if (!est_correctement_alloué(x))
        return ERREUR_ALLOCATION;

    y = créer_gros_tableau();
    if (!est_correctement_alloué(y)) {
        libérer_tableau(x);
        return ERREUR_ALLOCATION;
    }

    z = créer_gros_tableau();
```

```
    if (!est_correctement_alloué(z)) {
        libérer_tableau(x);
        libérer_tableau(y);
        return ERREUR_ALLOCATION;
    }

    Erreur erreur_1 = fonction_1();
    if (erreur_1 != aucune_erreur) {
        libérer_tableau(x);
        libérer_tableau(y);
        libérer_tableau(z);
        return ERREUR_FONCTION;
    }

    Erreur erreur_2 = fonction_2();
    if (erreur_2 != aucune_erreur) {
        libérer_tableau(x);
        libérer_tableau(y);
        libérer_tableau(z);
        return ERREUR_FONCTION;
    }

    libérer_tableau(x);
    libérer_tableau(y);
    libérer_tableau(z);
    return AUCUNE_ERRREUR;
}
```

L'approche est de tester les erreurs retournées par les fonctions et si c'est le cas, nettoyer les données avant de quitter la fonction.

On voit ici un premier problème : il faut que le développeur appelle explicitement les fonctions pour libérer la mémoire. Dans un code aussi simple, cela ne posera généralement pas de problème. Par contre, on peut sans difficulté imaginer des situations où l'allocation de la mémoire et la libération ne se seront pas réalisées dans le corps d'une seule fonction, mais à différents emplacements du programme. Le risque d'oubli sera important dans ces conditions.

Un second problème est que le code est lourd à lire. Sans gestion des erreurs, la fonction ne ferait que 8 lignes. Il est beaucoup plus difficile de

comprendre ce que fait un code lorsque les lignes importantes sont noyées au milieu de pleins de lignes accessoires. (Il serait probablement possible de réduire la taille du code, mais l'idée serait la même.)

En C++

Est-ce que le même code est utilisable en C++ ? Il compilera sans problème, mais c'est une très mauvaise idée. En effet, le C++ possède un mécanisme de gestion des erreurs supplémentaire par rapport au C, les exceptions. Si une exception est déclenchée dans l'une des fonctions (`creer_gros_tableau`, `fonction_1` ou `fonction_2`), le programme sortira directement de la fonction `une_fonction`, sans passer par les tests des erreurs. La mémoire ne sera donc pas libérée correctement, il y aura des fuites mémoire.

Une solution est d'attraper les exceptions avec `try` et `catch` :

```
void une_fonction() {
    Tableau x, y, z;

    try {
        x = créer_gros_tableau();
    } catch(creation_tableau_exception) {
        throw std::exception("Erreur Allocation");
    }

    try {
        y = créer_gros_tableau();
    } catch(creation_tableau_exception) {
        libérer_tableau(x);
        throw std::exception("Erreur Allocation");
    }

    try {
        z = créer_gros_tableau();
    } catch(creation_tableau_exception) {
        libérer_tableau(x);
        libérer_tableau(y);
        throw std::exception("Erreur Allocation");
```

```
    }

    try {
        fonction_1();
    } catch(execution_fonction_exception) {
        libérer_tableau(x);
        libérer_tableau(y);
        libérer_tableau(z);
        throw std::exception("Erreur Fonction");
    }

    try {
        fonction_2();
    } catch(execution_fonction_exception) {
        libérer_tableau(x);
        libérer_tableau(y);
        libérer_tableau(z);
        throw std::exception("Erreur Fonction");
    }

    libérer_tableau(x);
    libérer_tableau(y);
    libérer_tableau(z);
}
```

Le code n'est pas plus simple à lire que la version en C, le code important (création des tableaux, appel des fonctions, libération) est perdu au milieu de pleins de lignes de code accessoires. De plus, c'est toujours aux développeurs d'appeler les fonctions de libération des tableaux, le risque d'oubli est toujours présent. Dans ce sens, le C++ n'a pas du tout amélioré la situation par rapport au C.

La source des deux problèmes est la même : 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.

Dans les langages managés (type Java ou C#), la libération de la mémoire est prise en charge dans un mécanisme spécial (Gabage Collector pour la mémoire, dispose-pattern pour les autres ressources, plus `using` en C# et `try-with-resources` en Java 7+), le développeur

n'a plus besoin de penser à cela (ce qui n'est pas forcément une bonne chose de ne plus avoir à penser à la durée de vie des objets et de qui est responsable des objets...) L'erreur souvent faite est de croire que le C++ impose la gestion manuelle de la mémoire et est donc moins sûr, plus compliqué, moins productif que les autres langages. Ce qui n'est pas le cas, comme on va le voir.

Remarque : même si j'ai parlé d'allocation d'un tableau en mémoire dans cet exemple, le terme “acquisition d'une ressource” est à prendre au sens large. Il faut inclure tout ce qui nécessite une prise de responsabilité d'une ressource et la libération de cette responsabilité. Des exemples classiques sont l'ouverture et la fermeture d'un fichier, la connexion et la déconnexion d'une base de données, le verrouillage et la libération d'un mutex, etc.

```
file.open();
...
file.close();

sql.connect();
...
sql.free();

mutex.lock();
...
mutex.unlock();
```

Toutes ces situations présentent le même risque d'absence de libération de la ressource.

Pour en savoir plus sur l'approche exception vs retour de fonction, vous pouvez lire l'article de Aaron Lahman traduit en français : Retour de fonctions ou exceptions ?

Le RAII

Le RAII (“Ressource Acquisition Is Initialization”) est une technique pour éviter d'avoir à appeler manuellement la fonction de libération d'une

ressource. Plus de risque d'oubli ou d'exception qui empêche la libération. L'idée est de réaliser l'acquisition de ressource uniquement lors de l'initialisation d'une classe dédiée :

```
class TableauSecurisé {
    Tableau tableau_interne;
public:
    TableauSecurisé(Tableau t) : tableau_interne(t) {
    }
};

void une_fonction() {
    TableauSecurisé x = creer_gros_tableau();
    TableauSecurisé y = creer_gros_tableau();
    TableauSecurisé z = creer_gros_tableau();
    fonction_1();
    fonction_2();
    // quand sont appelées les fonctions libérer_tableau ?
}
```

Dit comme ça, je suppose que cela ne vous aide pas beaucoup à comprendre comment le RAII garantit la libération de la mémoire. La raison est que le terme RAII est très mal nommé :) (désolé, Stroustrup).

En fait, ce qui fait l'intérêt du RAII, c'est surtout que la libération sera appelée dans le destructeur :

```
class TableauSecurisé {
    Tableau tableau_interne;
public:
    TableauSecurisé(Tableau t) : tableau_interne(t) {
    }
    ~TableauSecurisé() {
        libérer_tableau(tableau_interne);
    }
};

void une_fonction() {
    TableauSecurise x = creer_gros_tableau();
    TableauSecurisé y = creer_gros_tableau();
    TableauSecurisé z = creer_gros_tableau();
```

```
    fonction_1();
    fonction_2();
}
```

Dans ce cas, les objets de type `TableauSecurisé` sont détruits lors de la sortie de la fonction `une_fonction` et la fonction `libérer_tableau` est appelée automatiquement par le destructeur de la classe. Il n'y a plus besoin d'appeler la fonction `libérer_tableau` manuellement. Même en cas d'exception, le destructeur sera appelé et le tableau sera libéré. En fait, le seul cas où le destructeur ne sera pas appelé, c'est lorsque l'exception sera lancée dans le constructeur. Mais dans ce cas, la ressource n'aura pas été acquise et il ne sera pas nécessaire de la libérer.

Remarque : si le constructeur contient plusieurs lignes de code qui peuvent lancer des exceptions, le destructeur ne sera pas appelé. Il faut donc bien faire attention que dans ce cas, les ressources acquises soient manuellement libérées. Pour éviter cette problématique, le mieux est d'avoir une classe qui ne fait qu'une seule chose : gérer la ressource (et donc ne pas pouvoir lancer d'autres exceptions dans le constructeur que l'acquisition de la ressource. De plus, pour des raisons de respect du Principe de Responsabilité Unique (SRP), une classe RAII ne doit rien faire d'autre que de gérer une ressource).

Le cas de l'allocation de tableau est généralisable à n'importe quelle allocation de ressources. On peut créer facilement des classes RAII pour les problèmes cités ci-dessus :

```
class FichierSecurise {
    Fichier fichier_interne;
public:
    FichierSecurise(Fichier f) : fichier_interne(f) {
        fichier_interne.open();
    }
    ~FichierSecurise() {
        fichier_interne.close();
    }
};

class MutexSecurise {
    Mutex mutex_interne;
```

```
public:
    MutexSecurise(Mutex m) : mutex_interne(m) {
        mutex_interne.open();
    }
    ~MutexSecurise() {
        mutex_interne.close();
    }
};
```

Et ainsi de suite.

Le RAII et la STL

En pratique, écrire une classe RAII demande un peu plus de code. Il faut en effet respecter la sémantique de valeur, ajouter les constructeurs par copie, par déplacement, les opérateurs d'affectation, etc. Peut-être également utiliser des abstractions un peu plus évoluées (template) pour avoir un code plus générique.

Heureusement, une grande partie du travail peut être simplifiée, puisque la bibliothèque standard (STL) utilise le RAII et propose de nombreuses fonctionnalités pour la gestion des ressources. Ainsi, en C++, on évitera d'utiliser les syntaxes héritées du C, mais on utilisera (et abusera) des classes de la STL. Par exemple :

Fonctionnalité	**En C**	**En C++**
Créer une chaîne de caractères	char* s;	std::string s;
Créer un tableau de données	Type* v;	std::vector<Type> v;
Créer un fichier	FILE* f;	std::ifstream f;/std::ofstream f;
Créer un objet sur le tas	Object* o;	std::unique_ptr<Object> p;
		std::shared_ptr<Object> p;
Verrouiller un mutex	?	std::lock_guard<std::mutex> l;

On voit qu'une grande partie des problématiques en C vient de l'utilisation des pointeurs nus (“nus” en opposition aux pointeurs “intelligents” `std::unique_ptr` et `std::shared_ptr`, qui garantissent la

sécurité du code). La conséquence est qu'en C++ “moderne”, on appliquera la règle suivante :

> **Aucun pointeur nu, aucun `new`, aucun `delete` (et encore moins de `malloc` ou de `free`).**

(Bien sûr, comme toutes les règles, celle-ci peut être violée, mais on ne doit le faire qu'avec de bonnes raisons, en connaissance de cause et en encapsulant au maximum dans un code dédié.)

Je n'utilise pas le C++11/14, est-ce que le RAII est intéressant pour moi ?

Il faut admettre un état de fait : il n'est pas toujours possible d'utiliser un compilateur à jour, supportant pleinement la version actuelle du C++ (contrainte dans un environnement professionnel, machine n'ayant pas de compilateur récent, etc.). Dans le cas d'une utilisation amateur par contre, aucune raison de ne pas mettre à jour son compilateur, surtout que les compilateurs les plus à jour (Clang 3.4, Gcc 4.9) sont gratuits.

Dans ce cas, il ne sera pas possible d'utiliser les classes comme `std::unique_ptr`, `std::shared_ptr` ou `std::lock_guard`. Cependant, cela ne signifie pas qu'il faut abandonner le RAII. Il existe d'autres classes supportant depuis longtemps le RAII dans la STL (`std::vector`, `std::string`, etc.), qu'il faut absolument utiliser. Pour les autres classes, il sera possible de se servir des bibliothèques apportant ces fonctionnalités (en premier lieu Boost : `boost::scoped_ptr`, `boost::shared_ptr`, `boost::lock_guard`).

Et même si ces classes ne sont pas disponibles, il faut comprendre une chose : le RAII n'est pas défini par un ensemble de classes de la STL, c'est une façon de concevoir sa gestion des ressources. Les classes de la STL ne sont qu'une implémentation du RAII. Si on ne peut pas utiliser les classes de la STL, il ne faut surtout pas écrire du code directement sans RAII :

```
// Non !
```

```
int* i = new int[1000];
```

Il faut au contraire prendre le temps d'écrire des classes RAII apportant des garanties fortes de libération des ressources. L'utilisation de `new` et `delete` doit être localisée uniquement dans ces classes RAII et exclue du reste du code.

```
class IntArraySafe {
    int* p;
public:
    IntArraySafe(int n) : p(new int[n]) {
    }
    ~IntArraySafe() {
        delete[] p;
    }
};
```

On m'a dit d'utiliser les pointeurs nus avec Qt, qui a raison ?

Avec Qt, la situation est un peu différente. Les objets à sémantique d'entité de Qt dérivent tous de la classe `QObject`, ce sont des objets polymorphiques. Pour être manipulés, ils doivent être utilisés avec des pointeurs.

Qt a été créé il y a bien longtemps, à une époque où les pointeurs intelligents n'étaient pas disponibles dans le C++. Pour éviter les fuites mémoires dues à l'utilisation des pointeurs nus, Qt utilise donc son propre système de destruction automatique des objets dérivés de `QObject`, qui peut entrer en conflit avec les approches “modernes” de gestion automatique des objets.

Ainsi, le code suivant provoquera des erreurs de tentatives de double destruction des variables membres :

```
class MyObject : public QObject
{
    QWidget widget_1;
```

```
    std::unique_ptr<QWidget> widget_2;
    QScopedPointer<QWidget> widget_3;
public:
    MyObject(QObject * parent = 0) : QObject(parent) {
        widget_1.setParent(this);
        widget_2.setParent(this);
        widget_3.setParent(this);
    }
};
```

Le principe est que lors de la destruction d'un objet dérivé de `QObject`, le destructeur prend la liste des objets enfants et les détruit (ce qui provoque une destruction en cascade des objets parent-enfant). Lorsque l'on utilise l'une des syntaxes précédentes (ou équivalent), les objets seront aussi détruits comme n'importe quel objet C++. Il y aura donc tentative de double destruction des objets, ce qui provoquera une erreur à l'exécution.

Au contraire, les objets en dehors d'une hiérarchie doivent être détruits selon les approches classiques du C++ :

```
int main(int argv, char** argc) {
    QApplication app(argv, argc); // variable locale,
détruite automatiquement à la fin de la fonction
    QScopedPointer<QMainWindow> mainWindow(new QMainWindow);
    ...
}
```

Il faut donc obligatoirement passer par des pointeurs nus :

```
class AnotherObject {}; // n'hérite pas de QObject

class MyObject : public QObject
{
    QWidget* widget_1; // objet Qt = pointeur nu
    AnotherObject object_1; // objet non Qt = variable
membre
    std::unique_ptr<AnotherObject> object_2; // ou pointeur
intelligent

public:
```

```
    MyObject(QObject * parent = 0) : QObject(parent) {
        widget_1 = new QWidget(this);
        object_2 = make_unique<AnotherObject>();
    }
};
```

Le fait de définir la classe `MyObject` comme parent (avec la fonction `setParent` comme ici, mais également en passant le parent dans le constructeur de l'objet ou en utilisant les layouts) permettra de détruire automatiquement ces objets `QWidget` lorsque `MyObject` sera détruit.

Cela est un héritage des premiers jours de Qt, il n'est malheureusement pas possible de faire autrement. Ce qui veut dire que lorsque l'on utilise Qt, il faut utiliser des syntaxes différentes selon le type des objets et surtout ne pas oublier de définir un parent.

À lire : (Not so much) Fun with QSharedPointer.

C++, Qt