Pourquoi le RAII est fondamental en C++ ?

La situation en C

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

    x = creer_gros_tableau();
    y = creer_gros_tableau();
    z = creer_gros_tableau();

    Erreur erreur_1 = fonction_1();
    if (erreur_1 != aucune_erreur)
        goto liberation_tableau;

    Erreur erreur_2 = fonction_2();
    if (erreur_2 != aucune_erreur)
        goto nettoyer;

    // autre chose

    nettoyer:
    liberer_tableau(x);
    liberer_tableau(y);
    liberer_tableau(z);
}
```

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. L'utilisation de goto est tout à fait acceptable dans ce contexte, c'est une approche classique

En C++

Est-ce que le même code est utilisable en C++ ? Il compilera sans problème (le C++ supporte le C), mais c'est une très mauvaise idée. En effet, le C++ possède une fonctionnalité supplémentarie par rapport au C, les exceptions. Si une exception est déclenchée dans 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 libéré correctement, il y a des fuites.

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

```
void une_fonction() {
    Tableau x = creer_gros_tableau(); // ok, si exception
ici, pas de mémoire à libérer

    try {
        Tableau y = creer_gros_tableau();
    } catch(...) {
        liberer_tableau(x);
    }

    try {
        Tableau z = creer_gros_tableau();
    } catch(...) {
        liberer_tableau(x);
        liberer_tableau(y);
    }

    try {
        fonction_1();
    } catch(...) {
        liberer_tableau(x);
        liberer_tableau(y);
        liberer_tableau(z);
    }

    try {
        fonction_2();
```

```
    } catch(...) {
        liberer_tableau(x);
        liberer_tableau(y);
        liberer_tableau(z);
    }
}
```

Le problème vient du fait que l'acquisition d'une ressource (ici, la création d'un tableau en mémoire) nécessite d'appeler manuellement une fonction pour libérer la ressource.

Le problème est donc que cette libération n'est pas forcement faite. En C comme en C++, le développeur peut oublier (ou ne pas savoir) qu'il faut appeler une telle fonction de libération (créer_gros_tableau pourrait par exemple attribuer un espace sur un tableau déjà existant, il n'y aurait alors pas de libération de mémoire à faire). Ou le lancement d'une exception peut faire sortir le programme de son déroulement “normal” (séquentiel), ce qui rend difficile de savoir si on récupère bien toutes les exceptions.

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 y inclure tout ce qui nécessite une prise de responsabilité d'une ressource et la libération de cette responsabilité. Des exemples classiques sont par exemple 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.

Le RAII

Le RAII signifie “Ressource Acquisition is Initialisation”, 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 déclarée une classe qui fera l'acquisition de ressource uniquement lors de son initialisation :

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

void une_fonction() {
    TableauSecurise x = creer_gros_tableau();
    // quand est appelé la fonction liberer_tableau ?
}
```

Dit comme ça, je suppose que cela ne vous aider pas beaucoup à voir comment le RAII garantie 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 TableauSecurise {
    Tableau tableau_interne;
public:
    TableauSecurise(Tableau t) : tableau_interne(t) {
    }
    ~TableauSecurise() {
        liberer_tableau(tableau_interne);
    }
};
```

```
void une_fonction() {
    TableauSecurise x = creer_gros_tableau();
}
```

Dans ce cas, l'objet x (classe TableauSecurise) est détruite lors de la sortie de la fonction une_fonction, il n'y a pas besoin d'appeler la fonction liberer_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'a pas été acquise et il n'est pas nécessaire de la libérer. Par contre, il est important de respecter une règle fondamentale : il ne doit y avoir qu'un seul endroit qui peut lancer une exception, lors de l'acquisition de la ressource, dans une classe RAII.

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

Fonctionnalité	**En C**	**En C++**
Créer une chaine de caractères	char* s;	std::string s;
Créer un tableau de données	TYPE* v;	std::vector v;

633

Créer un fichier	File f;	std::iofstream 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 (“nu” en opposition aux pointeurs “intelligent” unique_ptr et 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 leur utilisant 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). 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 unique_ptr, shared_ptr ou 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 (vector, string, etc.) qu'il faut absolument utiliser. Pour les autres classes, il sera possible d'utiliser des bibliothèques apportant ces fonctionnalités (en premier lieu Boost : boost.unique_ptr, boost.shared_ptr, boost.loack_guard).

Et même si ces classes ne sont pas disponible, il faut comprendre une chose : le RAII n'est pas défini par un ensemble de classe 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 dois être localisé uniquement dans ces classes RAII et exclu du reste du code.

```
class IntSafe {
    int* p;
public:
    IntSafe(int i) : p(new int(i)) {
    }
    ~IntSafe() {
        delete p;
```

```
    }
};
```

C++