Pourquoi le RAII est 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 forte 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 (peut 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 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 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 (le C++ supporte le C), 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 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 a des fuites mémoire.

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

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

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

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

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

```
    }

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

    try {
        fonction_2();
    } catch(...) {
        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 plein de lignes de code accessoire. De plus, c'est toujours au 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 améliorer la situation par rapport au C.

La source des deux problème 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, le développeur n'a plus besoin de penser à cela (ce qui n'est pas forcement 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 donc est 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 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 (“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();
    // quand est appelé la fonction libérer_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++