La gestion des erreurs

Table des matières

La première chose à faire pour gérer d'éventuelles erreurs lors de l'exécution d'un programme, c'est avant tout de les détecter. A l'exécution d'une fonction, et qu'une erreur a lieu, celle-ci doit prévenir l'utilisateur d'une manière ou d'une autre. Et elle peut le faire de plusieurs manières différentes...

1. Fonctions partielles et fonction totales

Il existe de nombreuses fonctions qui peuvent échouer, on appelle cela des fonctions partielles. Quelques exemples que nous utiliserons au cours de ce document :

- Une fonction maximum qui donne la valeur maximum d'une liste. Que se passe-t-il si la liste est vide ?
- Une fonction recherche qui cherche un élément dans un tableau et retourne l'indice où celui-ci se trouve. Que se passe-t-il si l'élément n'est pas dans le tableau ?
- Une fonction lireFichier qui retourne le contenu lue dans un fichier. Que se passe-t-il si le fichier n'est pas lisible ou n'existe pas ?

Pour toutes ces fonctions, il convient de mettre en place une politique de gestion d'erreur adaptée :

- La politique du contrat
- La méthode des valeurs sentinelles
- La méthode du flag de réussite
- La méthode par exception

2. Méthode par contrat

Cette méthode est de loin la plus simple, il suffit de placer un contrat avec l'utilisateur de la fonction disant qu'il ne doit appeler la fonction que dans un contexte où elle doit réussir.

Cette approche est utilisé par exemple en langage C++ pour l'opérateur operator[] ou la méthode at() d'accès aux cases d'un tableau sur un std::vector. Dans le cas où on essaye d'accéder à une mauvaise case, le comportement du programme est indéfini.

```
static void bubble_sort(vector<unsigned int> &liste)
/*
 * fonction : tri à bulle
 * entrée : liste à trier
 * pré condition : la taille du vecteur ne doit pas dépasser max_size
 *
 */
{
    unsigned int i, j(max_size);
    bool permut;

    do {
        permut = false;

        for (int i(0); i < max_size - 1; i++)
            if ( liste.at(i) > liste.at(i+1) )
                permut = _swap(liste[i], liste[i+1]);
        j--;
    } while ( j > 0 && permut );
}
```

Cette méthode est très simple, mais force le développeur à s'assurer que le contrat est respecté avant l'appel de fonction. C'est souvent contre productif voir impossible. Dans le cas de la fonction recherche il faudrait parcourir une première fois la structure pour vérifier que l'élément est dedans

avant d'appeler la fonction recherche pour savoir où il est. Dans le cas de la fonction lireFichier, c'est carrément impossible puisque pour savoir si on peut lire un fichier, il faut le lire, et qu'il n'y a aucune garantie qu'un fichier lue à un instant t sera lisible à l'instant t+1.

3. Méthode par valeur sentinelle

L'idée ici étant d'utiliser une valeur de retour particulière pour matérialiser l'erreur.

C'est une approche très souvent utilisée dans de nombreuses librairies. Par exemple, en C, nous avons la fonction fopen FILE *fopen(const char *path, const char *mode); chargée d'ouvrir un fichier. En cas de réussite, la fonction retourne un pointeur vers un objet utilisé par la suite pour traiter le fichier. En cas d'échec, elle retourne un pointeur NULL.

Dans le cas de la méthode recherche qui renvoie l'indice dans un tableau d'un élément recherché, on pourrait renvoyer une valeur négative, puisque les indices dans un tableau sont toujours positifs.

L'usage se ferait ainsi de la manière suivante en langage C++ :

```
// prototype de la fonction
int rechercher(const Collection &c, const Item item);
//...

int offset = rechercher(maCollection, monItem);

if ( offset >= 0 ) {
    std::cout << "Item trouvé à la position " << offset << std::endl;
    std::cout << "La valeur de l'item est " << maCollection[offset] << std::endl;
    }
else
    std::cout << "Item non trouvé"
```

Cette méthode ne peut cependant pas s'appliquer à tous les cas de figure. Quelle valeur sentinelle pourrait renvoyer la fonction maximum ? Celle-ci devra être garantie de ne pas pouvoir être confondue avec une valeur qui serait le vrai résultat de la fonction.

Cette méthode rend l'erreur facile, en effet, il est aisé d'oublier de tester la réussite, ainsi le code suivant, qui semble anodin, est faux :

```
int offset = rechercher(maCollection, monItem);

std::cout << "Item trouvé à la position " << offset << std::endl;
std::cout << "La valeur de l'item est " << maCollection[offset] << std::endl;
```

En effet, si l'élément n'est pas trouvé, offset vaut -1 et maCollection[offset] n'a pas de sens.

4. Méthode par drapeau de réussite

Ici la fonction vas renvoyer un drapeau (flag), souvent un booléen pour matérialiser la réussite. La valeur de retour étant en fait passée par référence et modifiée.

```
// prototype de la fonction
bool rechercher(const Collection &c, const Item item, int &offset);

//...

int offset;
bool found = rechercher(maCollection, monItem, offset);
```

```
if ( found ) {
    std::cout << "Item trouvé à la position " << offset << std::endl;
    std::cout << "La valeur de l'item est " << maCollection[offset] << std::endl;
    }
else
    std::cout << "Item non trouvé"
```

Cette approche corrige une des limitations de la méthode par valeur sentinelle, elle peut fonctionner pour n'importe quelle fonction, puisque il n'est pas nécessaire de trouver une valeur sentinelle adaptée, la réussite étant matérialisée par le booléen. Cependant on peut toujours oublier de tester le booléen de résultat et ainsi utiliser la valeur de offset qui serait non initialisée (ou initialisée par défaut avec une valeur fausse).

5. Méthode par exception

Avec cette méthode, la fonction de recherche d'un élément dans une liste va soit renvoyer l'indice de l'élément, soit lever une exception qui va remonter la pile d'appel jusqu'à être interceptée ou jusqu'à terminer le programme.

La méthode par exception a de nombreux avantages :

- Si on ne traite pas l'exception, elle finira le programme, souvent avec un message d'erreur explicite, ce qui évite les bugs dissimulés où la valeur sentinelle ou non initialisée est utilisée.
- Le code n'est pas complexifié par une gestion d'erreur avec de nombreux if
- La gestion d'erreur peut être repoussée à plus tard, dans une fonction appelante.

La gestion de l'exception se fait, en langage C++, avec un bloc try / catch :

```
static void bubble_sort(vector<unsigned int> &liste)
{
    try {
        unsigned int i, j(max_size);
        bool permut;

        do {
            permut = false;

            for (int i(0); i < 20; i++)
                if ( liste.at(i) > liste.at(i+1) )
                    permut = _swap(liste[i], liste[i+1]);
            j--;
            } while ( j > 0 && permut );
        }
    catch(std::exception const& e) {
        cerr << "ERROR: " << e.what() << endl;
        }
}
```

Cependant, rien ne force le développeur à gérer les exceptions et rien n'indique à celui-ci qu'une fonction peut lancer une exception, si ce n'est une lecture scrupuleuse de la documentation. Certains langages, comme Java, proposent un système d'exception qui force le développeur à gérer celles-ci ou à les propager explicitement à la fonction appelante.

La méthode par exception est sans doute la plus propre de toutes, mais il est difficile dans les langages qui ne gèrent pas les exceptions vérifiées (comme C++, Python, etc.) d'être certain d'avoir

traité toutes les exceptions possibles.

Enfin, toutes ces méthodes sont très implicites et il n'est pas évident de repérer les oublis de gestion d'erreur lors d'une lecture de code.