11

Erreur, erreur, erreur…

Depuis le début de ce livre, nous avons été épargnés par les erreurs. Nous avons certes appris à sécuriser un peu les entrées, mais c'est tout. Ça ne sera pas toujours le cas. Il serait de bon ton que nos programmes soient *résistants aux erreurs*, afin de les rendre fiables et robustes.

Le but de ce chapitre est d'introduire la gestion des erreurs ainsi que la réflexion à mener lors de l'écriture de son code.

11.1. L'utilisateur est un idiot

Derrière ce titre un poil provoquant se cache une vérité informatique universelle : peu importe que ce soit consciemment ou non, il arrivera forcément que l'utilisateur rentre une information incomplète, fausse ou totalement différente de ce qu'on attendait. Que ce soit parce qu'il veut s'amuser avec votre programme ou bien parce qu'il a mal compris une consigne, a oublié un caractère ou s'est trompé de ligne, votre programme doit être capable de gérer ça.

Vérifier les entrées veut bien sûr dire s'assurer qu'elles soient du bon type, mais aussi contrôler que la donnée est cohérente et valide. Ainsi, vous ne pouvez pas dire que vous êtes né un 38 avril, car un mois a 31 jours au maximum. Si la donnée n'est pas correcte, même si elle est du bon type, il faut demander à l'utilisateur de la ressaisir. Cela se fait relativement facilement.

```
#include <iostream>

int main()
{
  int jour { 0 };
  std::cout << "Donne un jour entre 1 et 31 : ";

  while (!(std::cin >> jour) || jour < 0 || jour > 31)
  {
    std::cout << "Entrée invalide. Recommence." << std::endl;
    std::cin.clear();
    std::cin.ignore(255, '\n');
  }
```

```
    std::cout << "Jour donné : " << jour << std::endl;
    return 0;
}
```

Tant que l'entrée met `std::cin` dans un état invalide ou qu'elle n'est pas dans la plage de valeurs autorisées, on demande à l'utilisateur de saisir une nouvelle donnée. Vous remarquerez que ce code est tiré de notre fonction `entree_securisee`. Nous l'avons extraite pour pouvoir ajouter des conditions supplémentaires. Au chapitre suivant (Section 12.7, [Pratique] Gérer les erreurs d'entrée • Partie VI), nous verrons une façon d'améliorer ça.

Pour l'instant, retenez qu'il faut vérifier qu'une donnée est cohérente en plus de s'assurer qu'elle est du bon type.

11.2. À une condition… ou plusieurs

Écrire une fonction peut être plus ou moins simple, selon ce qu'elle fait, mais beaucoup de développeurs oublient une étape essentielle qui consiste à *réfléchir au(x) contrat(s) que définit une fonction*. Un contrat est un accord entre deux parties, ici celui qui utilise la fonction et celui qui la code.

- La fonction mathématique `std::sqrt`, qui calcule la racine carrée d'un nombre, attend qu'on lui donne un réel nul ou positif. Si ce contrat est respecté, elle s'engage à retourner un réel supérieur ou égal à zéro dont le carré est égal à la valeur donnée en argument. Le contrat inhérent à la fonction `std::sqrt` se résume donc ainsi : *Donne-moi un entier positif, et je te renverrai sa racine carrée*. Ça paraît évident, mais c'est important de le garder en tête.
- Quand on récupère un élément d'un tableau avec `[]`, il faut fournir un index valide, compris entre 0 et la taille du tableau moins un. Si on fait ça, on a la garantie d'obtenir un élément valide.
- Si l'on a une chaîne de caractères ayant au moins un caractère, on respecte le contrat de `pop_back` et on peut retirer en toute sécurité le dernier caractère.

On voit, dans chacun de ces exemples, que la fonction attend qu'on respecte une ou plusieurs conditions, que l'on nomme les *préconditions*. Si celles-ci sont respectées, la fonction s'engage en retour à respecter sa part du marché, qu'on appelle les *postconditions*. Les passionnés de mathématiques feront un parallèle avec les domaines de définition des fonctions.

Le but de cette réflexion sur les conditions est de produire du code plus robuste, plus fiable et plus facilement testable, et ainsi toujours tendre vers une plus grande qualité.

En effet, en prenant quelques minutes pour penser à ce qu'on veut que la fonction fasse ou ne fasse pas, on sait plus précisément quand elle fonctionne et quand il y a un bug.

Contrats assurés par le compilateur

Le typage

Un exemple simple de contrat, que nous avons déjà vu sans même le savoir, s'exprime avec le typage. Si une fonction s'attend à recevoir un entier, on ne peut pas passer un `std::vector`. Le compilateur s'assure qu'on passe en arguments des types compatibles avec ce qui est attendu.

Avec *const*

Un autre type de contrat qu'on a déjà vu implique l'utilisation de `const`. Ainsi, une fonction qui attend une référence sur une chaîne de caractères constante garantie que celle-ci restera intacte, sans aucune modification, à la fin de la fonction. Cette postcondition est vérifiée par le compilateur, puisqu'il est impossible de modifier un objet déclaré comme étant `const`.

Vérifions nous aussi

Le compilateur fait une partie du travail certes, mais la plus grosse part nous revient. Pour vérifier nos contrats, nous avons dans notre besace plusieurs outils que nous allons voir.

11.3. Le développeur est un idiot

Vous souvenez vous du chapitre sur les tableaux ? Quand nous avons vu l'accès aux éléments avec les crochets, nous vous avons dit de vérifier que l'indice que vous utilisez est bien valide, sous peine de comportements indéterminés. Si vous ne le faites pas, alors c'est de votre faute si le programme plante, car vous n'avez pas respecté une précondition. C'est ce qu'on appelle une erreur de programmation ou *bug*. Lorsque ce genre de problème arrive, rien ne sert de continuer, mieux vaut arrêter le programme et corriger le code. Il existe justement un outil qui va nous y aider : les assertions.

Note › *Pour utiliser les assertions, il faut inclure le fichier d'en-tête* `<cassert>`.

Une assertion fonctionne très simplement. Il s'agit d'évaluer une condition quelconque. Si la condition est vraie, le programme continue normalement. Par contre, si elle se révèle fausse, le programme s'arrête brutalement. Voyez par vous-même ce qu'il se passe avec le code suivant.

```cpp
#include <cassert>

int main()
{
  // Va parfaitement fonctionner et passer à la suite.
  assert(1 == 1);
  // Va faire planter le programme.
  assert(1 == 2);
  return 0;
}
```

```
Assertion failed: 1 == 2, file d:\documents\visual studio 2017\projects
\zdscpp\zdscpp\main.cpp, line 8
```

Le programme plante et nous indique quel fichier, quelle ligne et quelle condition exactement lui ont posé problème. On peut même ajouter une chaîne de caractères pour rendre le message d'erreur plus clair. Cela est possible car, pour le compilateur, une chaîne de caractères est toujours évaluée comme étant `true`.

```cpp
#include <cassert>

int main()
{
  // Va parfaitement fonctionner et passer à la suite.
  assert(1 == 1 && "1 doit toujours être égal à 1.");
  // Va faire planter le programme.
  assert(1 == 2 && "Ouh là, 1 n'est pas égal à 2.");
  return 0;
}
```

```
[Visual Studio]
Assertion failed: 1 == 2 && "Ouh là, 1 n'est pas égal à 2."

------------------------------------------------------------

[GCC]
prog.exe: prog.cc:8: int main(): Assertion `1 == 2 && "Ouh là, 1 n'est
 pas égal à 2."' failed.

Aborted
------------------------------------------------------------

[Clang]
prog.exe: prog.cc:8: int main(): Assertion `1 == 2 && "Ouh là, 1 n'est
 pas égal à 2."' failed.

Aborted
```

Pour donner un exemple concret, on va utiliser `assert` pour vérifier les préconditions de l'accès à un élément d'un tableau. En faisant cela, nous éviterons les comportements indéterminés qui surviennent si nous violons le contrat de la fonction.

```
#include <cassert>
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> const tableau { -4, 8, 452, -9 };
  int const index { 2 };

  assert(index >= 0 && "L'index ne doit pas être négatif.");
  assert(index < std::size(tableau)
    && "L'index ne doit pas être plus grand que la taille du tableau.");

  std::cout << "Voici l'élément "
    << index << " : "
    << tableau[index] << std::endl;

  return 0;
}
```

Si par inadvertance nous utilisons un indice trop grand, alors le plantage provoqué par `assert` nous le signalera et nous poussera ainsi à corriger le code. La documentation est, pour cela, une aide très précieuse, car elle indique comment réagit une fonction en cas d'arguments invalides. Dans la suite du livre, nous apprendrons à utiliser ces informations.

Mais je ne veux pas que mon programme plante. Pourquoi utiliser `assert` ?

En effet, plus un programme est résistant aux erreurs, mieux c'est. Cependant, `assert` s'utilise non pas pour les erreurs de l'utilisateur mais bel et bien pour celles du programmeur. Quand le développeur est distrait et écrit du code qui engendre des comportements indéterminés (comme un dépassement d'indice pour un tableau), tout peut arriver et, justement, ce n'est pas ce qu'on attend du programme. Cela n'a donc pas de sens de vouloir continuer l'exécution.

Par contre, les erreurs de l'utilisateur (par exemple rentrer une chaîne de caractères à la place d'un entier) ne sont absolument pas à traiter avec `assert`. On ne veut pas que notre programme plante brutalement parce que l'utilisateur s'est trompé d'un caractère.