Notion de contrat de fonction et fonction `assert()`.

1. Notion de contrat de fonction 1
2. Gestion des erreurs 2
3. Notion de pré-condition (et de postcondition) 3
4. Gestion d'erreur ou pré-conditions ? 4
5. Exemples de « contrat de fonction » 4
6. Vérification des pré-conditions et fonction `assert()` 5
7. Exemples d'utilisation de assert() 5
8. Remarques finales 6

1. Notion de contrat de fonction

Vous savez déjà comment écrire et appeler une fonction (et une procédure[1]) en C.

Vous connaissez également déjà la notion de prototype (ou de signature) d'une fonction (ou d'une procédure) ; un prototype est constitué du *nom* de la fonction, de *la liste des types de ses paramètres* et de son *type de retour*. Le prototype d'une fonction permet de savoir un certain nombre de choses sur son fonctionnement. Il indique « ce sur quoi elle travaille » et laisse même entendre ce qu'elle fait - si son nom est bien choisi. De plus, le compilateur C doit connaître le prototype d'une fonction avant qu'elle soit appelée, afin de vérifier que l'appel est conforme – en particulier que les paramètres passés lors de l'appel correspondent bien aux types attendus[2]. Une erreur de compilation est générée dans le cas contraire. La *déclaration des prototypes* des fonctions est incidemment l'un des rôles des fichiers d'en tête (ou *header*) .h qui accompagne le code d'un ficher source .c, et qu'on inclut (#include "unHeader.h") dans les fichiers .c qui utilisent ces fonctions.

Mais bien sur, le prototype ne dit pas tout…

On appellera « contrat d'une fonction » la description précise, en français, de ce que fait la fonction[3]. Attention **: un contrat indique le « quoi » (ce que fait la fonction), mais pas le « comment »**. Le comment (la manière dont la fonction remplit son contrat) est dans le code lui même : le contrat voit les choses de plus haut.

Un (tout petit) peu plus précisément, le contrat décrit ce que fait la fonction « dans le cas général », mais aussi dans les cas particuliers (donc la façon dont les erreurs sont gérées). Il décrit également les *pré-* et *post-* conditions (voir ci dessous). En règle général, il ne suffit en effet pas de dire « ma fonction fait ceci ou cela ». Encore faut-il préciser le cadre d'exécution dans lequel la fonction est capable de faire « ceci ou cela », et la façon dont les cas particuliers et erreurs sont gérés.

---

1 Dans la suite, fonction désigne indifféremment une fonction une procédure (dont le type de retour est void).

2 Pour être exact, ce n'est pas strictement « obligatoire » en C, mais « fortement conseillé ». Incidemment, nous vous conseillons fortement de passer l'option -Wmissing-prototypes ou –Wall (« warn all ») au compilateur gcc (donc dans le Makefile) pour le forcer à bien vérifier la bonne déclaration des prototypes.

3 La notion de « contrat de fonction » peut prendre une signification beaucoup plus précise notamment dans le cadre de la « programmation par contrat ». On se contentera de cette entrée en matière simple ici.

Dans tous les cas, lorsque vous vous apprêtez à écrire une fonction, il vous appartient de bien réfléchir à son prototype et à son « contrat ». Ceci fait, on choisit souvent (dès lors que la fonction n'est pas extrêmement simple) de donner corps au contrat de fonction dans des commentaires placés au dessus de la déclaration de son prototype.

Effectuer intelligemment le découpage en fonctions d'un problème et définir précisément le contrat de ces fonctions durant l'analyse du problème, c'est faire un (très) grand pas ; après, il ne reste en quelque sorte plus qu'à écrire le code…

2. Gestion des erreurs

Une fonction doit souvent gérer des erreurs. Par exemple : une fonction qui ouvre un fichier doit vérifier que le fichier a pu être ouvert ; une fonction qui lit un fichier doit, au cours de la lecture, vérifier que le fichier respecte bien le format (la façon dont les données sont organisées dans le fichier) que la fonction sait lire ; une fonction qui alloue de la mémoire (avec malloc() ou calloc() ou…) doit vérifier que la mémoire a pu être alloué ; une fonction qui calcule la racine carrée d'un double doit vérifier que ce double est positif ; etc.

Gérer une erreur implique de la détecter mais aussi de pouvoir « faire savoir » au code appelant (qui utilise la fonction) qu'une erreur a eu lieu et idéalement quelle est l'erreur. Plusieurs mécanismes sont possibles en C. Le plus courant consiste à utiliser le type de retour de la fonction pour « remonter » l'information. Deux cas très utilisés sont :

- la fonction a un type de retour int, dont la valeur encode des informations sur le comportement de la fonction ; retourner 0 signifie en général « pas d'erreur », et retourner une valeur non nulle signifie qu'une erreur a été détectée – la valeur retournée encodant le type d'erreur. C'est, incidemment, le choix fait pour la fonction main() en C.

- une fonction qui alloue de la mémoire et retourne un pointeur sur le bloc alloué retourne NULL si une erreur d'allocation a eu lieu.

En ce qui concerne le *corps* de la fonction (son code), une structuration usuelle consiste à commencer par gérer les cas particuliers, dont les détections d'erreur, puis à traiter le cas général. On a alors quelque chose qui ressemble à :

```
/* ici : explication du contrat de la fonction */
int maFonction( <ses parametres >) {
   // détection des erreurs sur les valeurs de paramètres
   if( <param incorrect> ) {
      return <un nombre non nul>;
   }
   //...
   // gestion des cas particuliers
   if( <cas particulier> ) {
      // code pour un comportement particulier
      return 0 ;
   }

   // Maintenant, le cas général. Pas besoin de "else" !
   // ...
   return 0;
}
```

3. Notion de pré-condition (et de postcondition)

Une fonction, en général, ne peut faire ce qu'on en attend que lorsque ses paramètres vérifient certaines relations, sont dans certaines plages de valeur, etc.

Considérons par exemple la fonction suivante de recherche de minimum dans un tableau :

```
// Recherche et retourne le minimum des n premiers éléments
// du tableau t
// Paramètres :
//     t : tableau de flottant
//     n : nombre d'élément du tableau
float rechercher_minimum(float t[], int n) {
    // on suppose que l'element d'index 0 est le plus petit...
    float min = t[0];

    for( i=1; i < n; i++) {
        if(t[i] < min ) {
            min = t[i];
        }
    }
    return min;
}
```

Si la taille du tableau est nulle (n ==0), la fonction ne fait pas ce qu'il faut (puisqu'il n'y a pas de minimum). Pire, si le tableau n'est pas alloué (t==NULL par exemple), la fonction provoque une erreur de segmentation, puisque l'initialisation de min déréférence le premier élément du tableau, qui n'existe pas. Le « contrat de fonction » doit, en toute rigueur, prendre en compte cette situation.

Deux solutions de conception sont possibles :

- soit la fonction gère explicitement ces cas particuliers comme des erreurs ;
- soit on décide, dans le contrat de fonction, qu'on a le droit d'appeler la fonction *que si le tableau t est bien alloué et que sa taille n est > 0.*

Or, il est absurde d'appeler cette fonction sur un tableau de taille nulle. Ajouter un bloc « if » complexifie inutilement la fonction : il doit être exécuté à chaque appel mais ne sert pour ainsi dire jamais. On préfèrera donc la seconde solution.

Dès lors, le contrat de la fonction rechercher_minimum(…) doit inclure l'idée que *la fonction ne doit être appelée que si le tableau est bien alloué et de taille > 0.* Cette condition constitue ce que l'on appelle une *pré-condition*.

Une pré-condition est une condition booléenne portant sur l'état du système et des paramètres d'une fonction qui est censée être vérifiée avant d'appeler la fonction, par le code appelant. Si jamais la pré-condition n'est pas vérifiée, le comportement de la fonction est non spécifié (elle « a le droit », par contrat, de « faire n'importe quoi »).

Dans le cas de la fonction `rechercher_minimum()`, on pourrait donc écrire le contrat de fonction de la façon suivante :

```
// Recherche et retourne le minimum du tableau t, de taille n
// Paramètres :
//     t : tableau de flottant
//     n : nombre d'élément du tableau
// PRECONDITION : t doit être correctement alloué et donc n > 0
 float rechercher_minimum(float t[], int n) ;
```

4. Gestion d'erreur ou pré-conditions ?

Il vous appartient au cas par cas de décider pour si vous souhaiter utiliser un mécanisme de gestion d'erreur ou une pré-condition pour votre contrat de fonction. Si il faut donner quelques recommandations, disons qu'on réservera les gestions d'erreur pour les cas qui « font sens » (pour lesquels il est logique que la fonction soit appelée et doive indiquer qu'il y a erreur) et les pré-conditions pour des choses plus élémentaires (pour lesquels on est dans des conditions d'utilisation absurde pour la fonction).

5. Exemples de « contrat de fonction »

Fonction qui lit un fichier et stocke ses données dans une structure

```
// Lecture d'un fichier au format <QUELQUECHOSE>
// Lit le fichier pointé par p_file
// et charge les données dans *p_var
// PARAMETRES :
//    p_file : pointeur vers le fichier
//    p_var  : pointeur vers la structure, prealablement allouée
//
// RETOURNE : 0 si tout se passe bien
// RETOURNE : 1 en cas d'erreur (fichier incorrect)`
//
// PRECONDITION : le fichier est ouvert (p_file est valide)
// PRECONDITION : p_file pointe le debut du fichier
// PRECONDITION :p_var pointe une structure correctement allouée
int readFile( FILE * p_file, struct montypestruct * p_var);
```

Fonction de calcul des racines d'une équation du second degré

```
// Resolution d'une equation du second degre
// PARAMETRES :
//   a,b,c   : coefficients de l'equation
//   *x1 *x2 : solutions de l'equation, calculees
//
// PRECONDITION : le coefficient a doit etre non nul
// PRECONDITION : x1 et x2 pointent des variables double allouées
// RETOURNE : -1 si erreur (a == 0)
// RETOURNE : 0 si aucune erreur, deux solutions
// RETOURNE : 1 si aucune erreur, une solution unique
// RETOURNE : 2 si erreur (elta negatif, pas de resultats reels)
//     (=> disctiminant negatif, pas de solution reelle)
int solveEquation2ndDegre(double a, double b, double c,
                          double *p_x1, double *p_x2);
```

Conformément au principe des pré-conditions, la fonction « fait n'importe quoi » si a vaut 0 et/ou si p_x1 ou p_x2 sont NULL ; le développeur doit être sur que ces pré-conditions sont bien vérifiées avant d'appeler la fonction.

6 . Vérification des pré-conditions et fonction `assert()`

Par principe, une pré-condition sur l'état du système et des paramètres d'une fonction est donc censée être vérifiée *avant* d'appeler la fonction, par le code appelant – et non pas dans la fonction elle même.

Toutefois, en phase de développement (donc avant de livrer le programme), il est extrêmement utile de s'assurer que les pré-conditions d'une fonction sont bien vérifiée à chaque appel. Cela ralentira considérablement l'exécution du programme… mais vous permettra de détecter au plus vite de nombreuses erreurs !

Pour faire cela, pendant le développement, on pourrait écrire au début de chaque fonction une série de conditionnelles (if) qui chacune vérifie une pré-condition et qui affiche un message explicatif (par exemple « appel de la fonction invalide : pré-condition truc non vérifiée ») puis arrête brutalement le programme si elle n'est pas vérifiée en appelant `exit()`. Ensuite, après s'être assuré que tous les appels de fonction sont valides en testant le programme, et juste avant de le livrer, il suffirait de mettre en commentaire toutes les vérifications des pré-conditions.

Comme cela serait un peu fastidieux, le langage C dispose d'un mécanisme approprié : la fonction[4] `assert(expression-booleene)` dont le prototype est défini dans `assert.h`.

`assert()` a deux comportement suivant que votre programme est compilé en mode debug (option –g passée au compilateur) ou en more release (pas d'option –g).

- En mode debug (eg pendant la phase de développement), `assert()` ne fait rien si l'expression booléenne est vraie, mais arrête le programme avec un message d'erreur explicite (numéro de ligne, etc.) si l'expression est fausse.
- En mode release, `assert` n'est tout simplement pas exécuté (c'est comme si l'appel n'était pas dans le code).

On pourra donc utiliser `assert()` pour vérifier les pré-conditions, comme dans les exemples suivant.

7. Exemples d'utilisation de assert()

```
#include <assert.h>

// Rechercher l'indice du plus petit element du tableau
// PRECONDITION : n > 0 et t est alloué
float rechercher_minimum(float t[], int n) {
    assert(n>0) ;
    assert(t != NULL) ;

    // ici le « vrai » code de la fonction
}
```

[4] Il s'agit en fait d'une macro…

```
// Lecture d'un fichier au format <QUELQUECHOSE>
// Lit le fichier pointé par p_file
// et charge les données dans *p_var
// PARAMETRES :
//    p_file : pointeur vers le fichier
//    p_var  : pointeur vers la structure, prealablement allouée
//
// RETOURNE : 0 si tout se passe bien
// RETOURNE : 1 en cas d'erreur (fichier incorrect)`
//
// PRECONDITION : le fichier est ouvert en lecture
// PRECONDITION : p_file pointe le debut du fichier
// PRECONDITION :p_var pointe une structure correctement allouée
int readFile( FILE * p_file, struct montypestruct * p_var){
   // verification des préconditions
   assert(p_file != NULL);       // p_file est non NULL
   assert(ftell(p_file) == 0); // debut de fichier
   assert(p_var != NULL);
   // votre code...
}
```

8. Remarques finales

Ce qu'il ne faut pas faire avec `assert()` !

L'erreur à éviter absolument avec `assert()` est de *modifier l'état du système*, e.g. de modifier l'état d'une variable, lors de l'appel. En effet, cette modification sera effectuée lorsque le programme est compilé en mode debug (donc en phase de développement), mais ne sera plus effectué en mode release ! En d'autre terme : *que l'appel de* `assert()` *soit fait ou non, le comportement du programme ne doit pas changer.*

Prenons un exemple… Supposons qu'on veuille vérifier qu'une allocation se soit bien passée, mais sans pour autant traiter ce test comme une erreur potentielle.

Si on écrit :

```
int *p_entier;
assert( p_entier = calloc(10, sizeof(int)) != NULL);
```

alors *en mode debug* l'appel à `calloc()` sera fait, puis `assert()` sera exécuté et vérifiera que `p_entier` est non `NULL`, et tout se passera bien. Mais *en mode release*, `assert()` ne sera pas du tout exécuté, pas plus que `calloc()` et l'affectation de `p_entier` !

Autres utilisations de `assert()`

Bien sur, l'utilisation de `assert()` n'est pas limité à la vérification des pré-conditions ; `assert()` peut être utilisé partout ou vous il vous semble pertinent de vérifier que l'état du système (⇔ des valeurs des variables) est bien conforme à ce que le code doit produire ; à la fin d'une fonction par exemple (notion de post-condition), ou à chaque tour d'une boucle, etc.