

Chapitre 7 : Maintenance et modules

Construction et maintenance de logiciels

Guy Francoeur

basé sur du matériel d'Alexandre Blondin Massé, professeur

5 septembre 2019

UQÀM | **Département d'informatique**

Table des matières

1. Documentation
2. Introduction à la maintenance
3. Maintenance - techniques
 - Programmation conditionnelle
 - Insertion de traces
 - GNU gdb
4. Modules en C
 - Prototypage
 - static vs extern

Table des matières

1. Documentation
2. Introduction à la maintenance
3. Maintenance - techniques
4. Modules en C

Plusieurs types de documentation

- ▶ Il est important a stade du cours de comprendre que les entreprises vont impérativement se doter de politiques et standard qui concerne la documentation de logiciel ou encore des sources (code).
- ▶ Quelque soit le langage, il y aura toujours de la documentation.

Plusieurs types de documentation

- ▶ En-tête de **fonctions** (*docstrings*);
- ▶ En-tête de **fichiers** (auteurs, licence, version, etc.);
- ▶ Guide de l'**utilisateur**;
- ▶ **Tutoriels**;
- ▶ Guide du **développeur**;
- ▶ Documentation des **modifications** apportées;
- ▶ Code **source**, etc.

- ▶ Il existe de nombreux **compilateurs** C/C++ :
 - ▶ **gcc**;
 - ▶ **Borland C++**;
 - ▶ **Intel C++**;
 - ▶ **Microsoft (Visual Studio/Code) C/C++**;
- ▶ C a été **standardisé** dans les années **80** (norme **ANSI**);
- ▶ Il n'y a aucun **standard** de **documentation** qui a été proposé;
- ▶ Faute d'avoir un standard en C, il est possible de prendre ce qui fonctionne, tel que "doctrings" utilisé avec le langage **Java** et compatibles avec le générateur de documentation **javadoc**.

- Voici un exemple pour documenter une fonction :

```
1 /**
2  * Retourne une valeur non nulle si le point donné
3  * se trouve à l'intérieur du triangle donné.
4  *
5  * @param t    Un triangle
6  * @param p    Un point
7  * @return     Une valeur non nulle si le point donné
8  *             se
9  *             trouve à l'intérieur du triangle , 0
10  *             sinon
11 */
12 int estDansTriangle(Triangle t, Point2D p);
```

- Voici un exemple pour documenter un fichier :

```
1 /**
2  * Fichier geometrie.h
3  *
4  * Ce module fournit différents services de
5  * manipulation
6  * de figures géométriques en dimension 2.
7  *
8  * Exemples typiques d'utilisation :
9  * ...
10 *
11 * @author Guy Francoeur
12 * @version 1.1
13 * @since 2019-09-04
14 */
```


Étiquette	Description
@author	Auteur du module ou de la fonction
@deprecated	Indique que la fonction ou le module ne devrait plus être utilisé
@exception	Décrit le type d'exception qui peut être soulevée
{@link}	Insère un lien vers un autre module, fonction, etc.
@param	Une description d'un paramètre de fonction
@return	Une description de la valeur de retour de la fonction
@see	Indique une fonction ou un module relié
@version	Indique le numéro de version de la fonction ou du module
@since	Informe depuis quand le bout est disponible

Doxygen - un outil similaire à JavaDoc

- ▶ Logiciel : générateur de documentation pour **plusieurs langages**;
- ▶ Mais, surtout pour le code source rédigé en C/C++;
- ▶ Il permet de générer une documentation de format **HTML**;
- ▶ Il est aussi possible de générer un **manuel de documentation** en format **L^AT_EX**;
- ▶ Il est un logiciel qui s'exécute en ligne de commande;
- ▶ Sous licence GPL Copyright ©1997-2018 by Dimitri van Heesch.;
- ▶ Il est **Portable** et **configurable**;
- ▶ **Site officiel**.

- ▶ **Étape 1** : Installation = télécharger l'exécutable.
- ▶ **Étape 2** : Génération du fichier de configuration.

```
$ doxygen -g config
```

- ▶ **Étape 3** : Configuration. On peut choisir entre autres la **langue**, les fichiers qu'on souhaite **documenter**, etc.
- ▶ **Étape 4** : Génération de la documentation.

```
$ doxygen config
```

Configuration Doxygen

Avec modification des paramètres suivants :

```
# Choix de la langue  
OUTPUT_LANGUAGE = French  
  
# Documentation de tous les fichiers  
EXTRACT_ALL      = YES  
  
# Affiche les fichiers sources  
SOURCE_BROWSER   = YES
```

Table des matières

1. Documentation
2. Introduction à la maintenance
3. Maintenance - techniques
4. Modules en C

- ▶ Rôle (but)
- ▶ Quand
- ▶ Pourquoi
- ▶ Type
- ▶ Risques
- ▶ Coûts
- ▶ Stratégies

Les buts (rôle) principaux de la maintenance sont :

- ▶ Corriger;
- ▶ Compléter;
- ▶ Changer (dans le sens de changement d'idée);
- ▶ Réadapter (au nouveau besoin);
- ▶ Migration;

À quel moment peut survenir la maintenance ?

- ▶ N'importe quand et toujours;
- ▶ Si on fait la somme des :
 - ▶ Bogues (mineurs ou majeurs);
 - ▶ Demandes de changements;
 - ▶ Finir un projet oublier;
 - ▶ Mise à niveau (upgrade);
- ▶ Il est possible que la somme soit supérieure aux nouveaux projets.

- ▶ On change les systèmes (OS);
- ▶ L'entreprise a été acquise par une autre;
- ▶ L'entreprise introduit une nouvelle façon de faire;
- ▶ L'entreprise introduit un nouveau forfait ou produit;
- ▶ L'entreprise veut simplement se moderniser ou être à la mode;

Type - Maintenance de logiciels

- ▶ Adaptative lié a l'intégration d'un progiciel;
- ▶ Code source lié à la modification du code;

Il y a des risques changer les choses.

- ▶ Compréhension;
- ▶ Expertise (existe-t-elle encore);
- ▶ Stress : Sommes-nous arrêtés ?;
- ▶ Temps, Quel est le temps pour faire le changement;
- ▶ Technique, le code version, le système version;
- ▶ Compatibilité des éléments à modifier. (vieux vs nouveau);

Ici c'est très paradoxal, mais ce n'est peut-être pas la modification qui sera l'élément le plus coûteux !

- ▶ Nouvel équipement (coût assez fixe);
- ▶ Temps est sûrement un facteur
 - ▶ Compréhension (technique, fonctionnelle);
 - ▶ Changement;
 - ▶ **Test et la conformité (QA/QC);**

En résumé la maintenance c'est beaucoup de test pour un petit changement. Encore, beaucoup de test pour un gros changement.

Stratégies - Maintenance de logiciels

Afin d'avoir du succès, une stratégie pour une modification de logiciel écrit en C, est sûrement utile :

- ▶ Utiliser un gestionnaire de version;
- ▶ Comprendre la demande (la schématiser);
- ▶ Réfléchir à la couverture de test;
- ▶ Préparer des tests (TDD) déterministes;
- ▶ Mettre en place un environnement stable d'exécution de test;
- ▶ Penser modulaire;
- ▶ Coder vos stratégies à l'extérieur du source code officiel;
- ▶ Garder ça simple;
- ▶ Ne pas réinventer la roue; En C, ceci est vital;

- ▶ Ne jamais remettre à plus tard;
- ▶ Faire les choses simplement au fur et à mesure;
- ▶ Documenter si ceci est requis pendant pas après;
- ▶ Toujours faire le minimum; Parfois, minimum égal beaucoup;
- ▶ Bien évaluer les impacts pas pour soi, mais, pour autrui;
- ▶ Toujours garder ça propre, on ne sait pas qui pourrait y jeter un œil;

Table des matières

1. Documentation
2. Introduction à la maintenance
3. Maintenance - techniques
 - Programmation conditionnelle
 - Insertion de traces
 - GNU gdb
4. Modules en C

Directives au préprocesseur

- ▶ Préfixées par le symbole #;
- ▶ Directives :
 - ▶ #include;
 - ▶ #define;
 - ▶ #if;
 - ▶ #endif;
 - ▶ #ifndef, etc.
- ▶ Les directives sont **lues et interprétées** par le préprocesseur avant même de procéder à la **compilation** des différents fichiers.

Symboles

- ▶ Pour définir un **symbole** ou une **macro**, on utilise la directive

```
#define <identificateur> <valeur>
```

- ▶ Le préprocesseur remplace toutes les occurrences de **<identificateur>** (comme mot) par **valeur**;
- ▶ La valeur est donnée par **le reste de la ligne**;
- ▶ Pour affecter une valeur sur **plusieurs lignes**, il faut utiliser le caractère ****;
- ▶ La **portée** du symbole s'étend jusqu'à la **fin du fichier** dans lequel il est défini;
- ▶ Sauf si on trouve une commande

```
#undef <identificateur>
```

Définition de symboles à la compilation

- ▶ Il est possible de définir des symboles à la compilation seulement :

```
$ gcc -DLINUX fichier.c
```

ce qui est équivalent à mettre la directive suivante dans `fichier.c` :

```
#define LINUX
```

- ▶ On peut également donner une **valeur** au symbole :

```
$ gcc -DLANGUE=FR fichier.c
```

ce qui est équivalent à :

```
#define LANGUE FR
```

Symboles prédéfinis

```
1 //predefini.c
2 #include <stdio.h>
3
4 int main() {
5     printf("%s\n", __FILE__); // Nom du fichier source courant
6     printf("%d\n", __LINE__); // Numéro de la ligne courante
7     printf("%s\n", __DATE__); // Date de compilation (format MMM
8         JJ AAAA)
9     printf("%s\n", __TIME__); // Heure de compilation (format HH
10        :MM:SS)
11     printf("%d\n", __STDC__); // 1 si le compilateur est
12        conforme à la norme ISO
13     return 0;
14 }
15 /*
16 predefini.c
17 7
18 Oct 27 2017
19 08:12:52
20 1
21 */
```

Constantes

- Dans certains cas, il est **nécessaire** d'utiliser des symboles pour définir des **constantes** :

```
1 #include <stdio.h>
2
3 int main() {
4     const int nbLig = 2;
5     int a[nbLig] = {1,2};
6 }
```

tableau.c: In function main:

tableau.c:6:5: erreur: un objet de taille variable
peut ne pas être initialisé

```
    int a[nbLig] = {1,2};
    ^
```

tableau.c:6:5: attention : éléments en excès dans l'
initialisation de tableau [enabled by default]

tableau.c:6:5: attention : (near initialization for
a) [enabled by default]

Directives

- ▶ Pour le **compilateur**, les variables constantes sont des **variables** qu'on ne peut modifier, mais pas des **constantes**.
- ▶ Il est nécessaire d'utiliser une **directive #define** pour créer un symbole utilisable avec les tableaux;
- ▶ Les avertissements vont disparaître;

```
1 //tableau2.c
2 #include <stdio.h>
3
4 #define NB 2
5
6 int main() {
7     int a[NB] = {1,2};
8     printf("%d, %d\n\n", a[0], a[1]);
9 }
```

- ▶ Une **macro-fonction** est un symbole **paramétrable**;

- ▶ **Syntaxe :**

`#define` $f(x_1, x_2, \dots, x_n)$ `<corps>`

- ▶ Le remplacement ne se fait que pour les **occurrences** de la forme

$f(v_1, v_2, \dots, v_n)$

Dangers associés aux macro-fonctions

- ▶ Mauvaise **substitution** si le corps et les paramètres ne sont pas correctement **parenthésés**;
- ▶ Les paramètres peuvent être évalués **plusieurs fois**;
- ▶ **Erreurs** lorsqu'il y a des **effets de bord**;
- ▶ **Inefficacité** lors d'évaluations **multiples**;
- ▶ Conclusion : ne pas utiliser de **macro-fonctions** et favoriser l'utilisation de **fonctions** de la façon habituelle.

Utilisations fréquentes

- Gestion du **paramétrage** de **différentes versions** du même programme :

```
1 #ifdef LINUX
2 #include "linux.h"
3 #else
4 #ifdef MAC_OS
5 #include "mac_os.h"
6 #endif
7 #endif
```

- Blocage des **inclusions multiples** des en-tête :

```
1 #ifndef PILE_H
2 #define PILE_H
3
4 //code ici ...
5
6 #endif
```


Trace conditionnelle

- Il est possible d'avoir des traces conditionnelles grâce aux directives;

```
1 //trace.c
2 #include <stdio.h>
3 #include "outils.h"
4 int main () {
5 #ifdef TRACE
6     printf("argc est:%d",argc);
7 #endif
8     for (int i=0;i<argc;++i) {
9         cmdline(argc, argv);
10    }
11    return 0;
12 }
```

- Pour activer les traces nous compilons avec :
\$ gcc -DTRACE -std=c99 -o trace trace.c

- ▶ GNU gdb est le *débugger* de base inclus avec le compilateur **GCC** ou **G++**;
- ▶ `$ gcc -g` est requis pour utiliser gdb;
- ▶ il est lancé par la commande : `$ gdb`;
- ▶ ou avec le programme : `$ gdb ./exo8`;

GNU gdb - les options

option		description	usage	description FR
breakline	b		b 1	arrête à la ligne 1
next	n	step over	n	exécute une ligne
step	s	step into	s	exécute, si fonction on entre
print	p		p a	affiche la variable a
run	r	running	run	exécute jusqu'au break
quit	q	quit gdb	quit	quitter gdb

Table des matières

1. Documentation
2. Introduction à la maintenance
3. Maintenance - techniques
4. Modules en C
Prototypage
static vs extern

Déclaration et implémentation

- ▶ C'est une bonne pratique de déclarer les **prototypes** des fonctions au **début** du fichier où elles sont **définies** et/ou **utilisées**;
- ▶ Il n'est **pas nécessaire**, mais tout de même **encouragé** de donner un **nom** aux paramètres;
- ▶ Lors de la **définition**, le nom des variables est **obligatoire**.
- ▶ Contrairement à C++ et Java, la **surcharge** de fonctions est **interdite** :

```
1 int max(int x, int y);  
2 int max(int x);
```

test.c:2: error: conflicting types for 'max'

test.c:1: error: previous declaration of 'max' was here

- ▶ Il est également possible de définir des variables **extra modulaire à plusieurs fichiers**, par l'intermédiaire du mot réservé **extern**;
- ▶ Par opposition aux **variables externes**, les variables **statiques**, déclarées à l'aide du mot réservé **static**, ont une portée limitée au **fichier** dans lequel elles sont déclarées.
- ▶ Les variables et fonctions globales sont **visibles** de leur déclaration jusqu'à la **fin du fichier** où elles sont définies;
- ▶ **Utilisables** jusqu'à la fin du programme;
- ▶ **Initialisées** à 0 par défaut;
- ▶ Les **fonctions** ont la même visibilité, accessibilité et durée de vie que les variables globales.

Variables et fonctions globales

Fichier main.c

```
1 #include <stdio.h>
2 #include "math.c"
3
4 int main() {
5     printf("PI = %f\n", PI);
6     printf("Le carre de %d
7         est %d\n", 4, carre
8         (4));
9     return 0;
10 }
```

Fichier math.c

```
1 const float PI =
2     3.141592654;
3
4 int carre(int x) {
5     return x * x;
6 }
```

Affiche :

PI = 3.141593

Le carre de 4 est 16

Variables et fonctions statiques

```
1 static char tampon[TAILLE_TAMPON];  
2 static int x;  
3 static int factorielle(int n);
```

Les variables **locales statiques** sont

- ▶ associées à un espace de stockage **permanent**;
- ▶ existent même lorsque la fonction n'est pas **appelée**.

Les variables **globales statiques** et les **fonctions statiques** se comportent

- ▶ exactement comme les variables **globales** et les **fonctions**,
- ▶ à l'exception qu'elles ne peuvent être utilisées **en dehors du fichier** où elles sont définies.

Variables externes

- ▶ Permettent de définir des variables **globales à plusieurs fichiers**;
- ▶ Par défaut, toute variable **non locale** est considérée externe;
- ▶ Par l'intermédiaire du mot réservé **extern**;
- ▶ Uniquement pour une **déclaration** sans **initialisation**;
- ▶ Utiles lorsqu'on souhaite compiler les fichiers **séparément**;
- ▶ Ont une durée de vie aussi longue que celle du **programme**;
- ▶ Pour les **tableaux**, il n'est pas nécessaire d'indiquer une **taille**.

```
1 extern int x, a [];
```

Documentation d'une fonction *facultatif*

- ▶ Bien qu'il n'y ait pas de **standard** de documentation en C, on utilise souvent le standard **Javadoc** :
- ▶ Aussi, si la **déclaration** (du **prototype**) et l'**implémentation** sont séparées, on documente plutôt la **première**.

```
1  /**
2   * Calcule la n-ième puissance de x.
3   *
4   * La n-ième puissance d'un nombre réel x, n étant un entier
5   * positif, est le produit de ce nombre avec lui-même répété
6   * n fois. Par convention, si n = 0, alors on obtient 1.0.
7   *
8   * @param x   Le nombre dont on souhaite calculer la puissance
9   * @param n   L'exposant de la puissance
10  * @return    Le nombre x élevé à la puissance n
11  */
12 float puissance(float x, unsigned int n);
```

- ▶ Typiquement, un **module** en C est divisé en **deux fichiers**;
- ▶ Un premier **fichier.h**, qui contient l'**interface**;
- ▶ Et un second **fichier.c** qui contient l'**implémentation** de cette interface;
- ▶ Avantages de **séparer** l'interface de la **mise en oeuvre** ?

Extensions des fichiers

- ▶ En principe, pour les systèmes **Unix**, les extensions n'ont pas d'importance;
- ▶ Par contre, elles guident le compilateur **gcc** :
 - ▶ **.c** : code source en C;
 - ▶ **.cpp**, **.C** et **.cc** : code source en C++;
 - ▶ **.s** : code source en assembleur;

```
$ gcc -S tp1.c  
$ gcc -c tp1.s
```
 - ▶ **.o** : fichier objet;
 - ▶ **.a** : fichier archive.

Rôles du .h déclaration

- ▶ Il est possible de lister toutes les fonctions sans les implémenter;
- ▶ Il est aussi possible de déclarer et implémenter dans le .h;
- ▶ Il permet de garder les fonctions d'un même sujet ensemble;
- ▶ Permet d'un seul coup d'œil de trouver ce que nous recherchons;
- ▶ Simple à construire;
- ▶ Permet d'éviter les inclusions multiples.

- ▶ Garde le code de vos fonctions;
- ▶ Maintient la modularité et la recherche de fonction spécifique;
- ▶ Améliore la performance (compilation, et maintenance);

Exemple du .h - interface (*header*)

► Exemple :

```
1 #ifndef OUTILS_H
2 #define OUTILS_H
3
4 int cmdline(int, const char **);
5
6 #endif
```

Exemple du .c - implémentation (*source*)

```
1 // implementation de mes outils
2 int cmdline(int __argc, const char **__argv)
3 {
4     int n = 1; int c = 0;
5     int VALID = 0;
6     while(n <= __argc) {
7         #ifndef TRACE
8             printf("debug: argument %d est %s\n",argc, argv[n]);
9         #endif
10        if (__argv[n][0] == '-') {
11            switch (__argv[n][1]) {
12                case 'd' : c++; break;
13                case 'i' : c++; break;
14                case 'o' : c++;
15                default : VALID = 1;
16            }
17        }
18    }
19    if (c < 2) VALID = 2;
20
21    return VALID;
22 }
```


- **Étape 1** : Compilation des fichiers sources.

```
$ gcc -c outils.c
```

- **Étape 2** : Édition des liens.

```
$ gcc -o prog prog.c outils.o
```

- **Étape 3** : Exécution.

```
$ ./prog
```

- **Étape 1** : Compilation courte des fichiers sources.

```
$ gcc -std=c99 -O2 -o prog prog.c outils.c
```