Chapitre 7 : Maintenance et modules

Construction et maintenance de logiciels

Guy Francoeur

basé sur les travaux d'Alexandre Blondin Massé, professeur

29 avril 2019

UQÀM | Département d'informatique

Table des matières

Table des matières

Plusieurs types de documentation

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

Documentation en C

- Il existe de nombreux **compilateurs** C/C++ :
  - **gcc**;
  - **Borland C++**;
  - **Intel C++**;
  - **Microsoft (Visual Studio/Code) C/C++**;
  - etc.
- C a été **standardisé** dans les années **80** (norme **ANSI**);
- En particulier, il n'y a aucun **standard** de **documentation** qui a été proposé;
- Plusieurs conviennent d'utiliser les **"doctrings"** comme dans le langage **Java** compatibles avec le générateur de documentation **javadoc**.

Documentation des fonctions

- L'**en-tête** de chacune des **fonctions** devrait toujours être documentée :
- Exemple :

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

Documentation des modules

- De la même façon, il est important de documenter l'**en-tête des fichiers** :
- Exemple :

```
1  /**
2   * Fichier geometrie.h
3   *
4   * Ce module fournit différents services de
5   * manipulation de figures géométriques en
6   * dimension 2.
7   *
8   * Exemples typiques d'utilisation :
9   * ...
10  *
11  * @author Alexandre Blondin Masse
12  * @version 1.0
13  */
```

Étiquettes Javadoc

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 brève description d'un paramètre de fonction
@return	Une brève description de la valeur de retour d'une fonction
@see	Indique une fonction ou un module relié
@version	Indique le numéro de version de la fonction ou du module
etc.	

Doxygen

- Site officiel;
- Dépôt sur Github;
- Système de **documentation** pour **plusieurs langages**, dont C/C++;
- Il permet de générer une documentation **en ligne** sous format **HTML**;
- Aussi un **manuel de documentation** sous format **LaTeX**;
- Sous licence **GPL**;
- **Portable** et **configurable**.

Utilisation

- **Étape 1** : Installation.
  - Dépend des **systèmes**;
  - Sous les systèmes **Unix**, s'assurer que le **binaire** soit accessible depuis n'importe où (ajouter dans la variable PATH).
- **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
```

Exemple

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

Maintenance de logiciels

- Modification d'un logiciel **déjà livré**;
- Souvent pour **corriger des erreurs** ou des **bogues**;
- Parfois pour **ajouter** une fonctionnalité, améliorer les **performances**, faire une **refactorisation** du code à cause d'un changement de politique ou façon de faire;
- On évalue à **20%** les activités de **développement de nouveaux logiciels**;
- Par opposition à **80%** qui concernent la **maintenance** :
  - **20%** pour la correction de bogues;
  - **30%** pour l'adaptation de logiciels;
  - **50%** pour l'amélioration;

Cas des entreprises non informatiques

- Dans le cas des compagnies qui n'ont pas une **vocation informatique**, la **maintenance** occupe une place plus importante encore;
- Il y a en particulier très peu de **développement logiciel**;
- Souvent, on doit **intégrer** des logiciels déjà existants;
- Par exemple, **développer des interfaces** entre différents logiciels;
- Développement de **rapports** et autres outils de **consultation des données**.

Coût de la maintenance (1/2)

- Il est beaucoup plus difficile de modifier un système en **activité** qu'un système en **développement**;
- Il faut en particulier mesurer les **impacts** sur les opérations et **limiter ces impacts**;
- Souvent, les responsables de la **maintenance** ne sont pas ceux qui ont participé au **développement**;
- Il faut prévoir un **temps d'apprentissage** au niveau :
  - **fonctionnel** : **Que fait** le logiciel ? À quoi **sert**-il ?
  - **structurel** : Quelle est la **structure** du logiciel ? Comment est-il **décomposé** ?
  - **technique** : **Langages** de programmation, **outils**, **styles**, etc.

Coût de la maintenance (2/2)

- Plus un programme est **vieux**, plus il a subi des **activités de maintenance**, plus il est **complexe à modifier**;
- Plusieurs critères font en sorte qu'il est **préférable** de garder un vieux système que de le **changer** :
  - Performance **connue** et satisfaisante du système;
  - Coût d'investissement **trop élevé**;
  - **Risque trop élevé** de changer de système.
- Cas typique : plusieurs de systèmes dans le milieu **bancaires** et **financiers** utilisent encore du **Cobol** et du **Fortran**.

Étapes lors de la maintenance (1/2)

Avant de **programmer** :

- On doit **comprendre le logiciel**, au niveau fonctionnel et structurel;
- Comprendre les **modifications demandées**;
- Évaluer de quelles façons ces **modifications** peuvent être apportées;
- Si possible, proposer une ou plusieurs approches de **mise en oeuvre** des modifications demandées;
- Évaluer l'**impact** de la réalisation de ces modifications :
  - Les **structures de données** sont-elles affectées ? Dans **quelles proportions** ?
  - Quelles sections de code sont **touchées** ?

Étapes lors de la maintenance (2/2)

- Choisir la solution la **moins coûteuse**, la **moins complexe** et la **plus facile à maintenir** à long terme;
- L'**implémenter** en respectant le plus possible le **style de programmation**;
- Mettre à jour les **plans de tests** (unitaires et intégrés) ou en ajouter si inexistants;
- Vérifier que le nouveau programme passe les **tests**;
- Documenter les **modifications apportées**, en particulier, décrire le **problème** et **la solution** apportée;
- Si nécessaire, donner une **formation** aux utilisateurs sur les **nouvelles fonctionnalités**;
- **Coordonner** la mise en production.

Documentation de la maintenance

- Les logiciels de **contrôle de version** prennent en charge de plus en plus la documentation de la **maintenance**;
- Cependant, dans certains cas, il est important de documenter la **maintenance** directement dans le code;
- Dans l'en-tête de chaque **fichier modifié** indiquer la **date**, l'**auteur** et s'il y a lieu la **référence** de la modification;
- Pour chaque **fonction** ou **bloc modifiés**
  - expliquer le **problème** et la **solution** apportée;
  - indiquer l'**auteur** et la **date**;
- En cas de **suppression** ou de **modifications majeures** de sections de code, il est **rarement** pertinent de garder les **anciennes versions** en **commentaires**;

Table des matières

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
           JJ AAAA)
8      printf("%s\n", __TIME__); // Heure de compilation (format HH
           :MM:SS)
9      printf("%d\n", __STDC__); // 1 si le compilateur est
           conforme à la norme ISO
10     return 0;
11 }
12 /*
13 predefini.c
14 7
15 Oct 27 2017
16 08:12:52
17 1
18 */
```

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 disparaitre;

```
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 }
```

Macro-fonctions *dangereux*

- Une **macro-fonction** est un symbole **paramétrable**;
- **Syntaxe** :

```
#define f(x1,x2,...,xn) <corps>
```

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

```
f(v1,v2,...,vn)
```

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**;
- **Ineffacité** 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 conditionnel grâce au 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

- 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

Modules en C

- 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 a construire;
- Permet d'éviter les inclusion multiple.

Rôles du .c implémentation

- 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*)

- Exemple :

```
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      #ifdef 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 }
```

Compilation séparée

- **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
```

Compilation courte

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

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