Département IMA / 3A (S6) / 2012 **Programmation Avancée**
François Boulier, Julie Jacques, Julien Forget, Shahin Gelareh et Laure Gonnord.

Rapport sur le projet de programmation avancée 2012

Laure Gonnord, Juin 2012

Contexte

Ce document est le rapport général distribué aux étudiants d'IMA3, Polytech'Lille, en juin 2012. Il contient des remarques d'ordre général sur les projets de développement logiciel de la matière *Programmation Avancée* (S6).

Cette année, le projet de développement logiciel en binômes consistait en la réalisation de statistiques sur l'œuvre de Shakespeare, à partir de données prises sur un site Web (et donc en format HTML). Le sujet peut être trouvé à l'adresse http://laure.gonnord.org/pro/teaching/PA1112/ProjetPA_IMA3_2011_2012.pdf

1 Remarques générales

1.1 Rendu

Sur 26 binômes :
- 18 binômes ont rendu à l'heure avec un dépôt SVN qui vérifiait les consignes, éventuellement en se faisant aider par J. Forget à la dernière minute.
- 5 binômes ont rendu dans la demi-journée sans aide des enseignants.
- 2 binômes ont rendu dans les 24h, 1 avec aide de L. Gonnord
- Un binôme a rendu 2 jours après.
- Un autre binôme a rendu 6 jours après une version2.

Globalement donc, les consignes ont été respectées et tous les projets ont été rendus, ce qui est très satisfaisant. Les retards ont été sanctionnés de manière à ne pas pénaliser trop les projets déjà faibles (attention, ce ne sera pas le cas tout le temps).

1.2 Utilisation du logiciel de contrôle de version

Ce projet a été l'occasion d'utiliser un gestionnaire de contrôle de versions.

Il y a des disparités notoires pour l'utilisation de SVN :
- Malgré le support offert en séances, certains binômes n'ont quasiment pas fait usage de SVN, et leur manque de pratique a causé des demandes de support répété par mail.
- Certains binômes (plus de la moitié) ont fait un usage intensif de SVN en séance et en dehors des séances, comme le montrent leurs logs SVN (Le logiciel StatSVN : http://statsvn.org/ a été utilisé pour cette partie de la correction).
- Les messages de commit sont en général clairs, évitez d'utiliser les messages de commit vides qui n'apportent aucune information.

Les principales difficultés/demande de support Elles sont de deux types :

- L'accès au dépôt de l'extérieur de Polytech, qui n'est pas facile à réaliser vu qu'il utilise un tunnel SSH. Cette manipulation technique a été effectuée de manière trop tardive pour certains, avec des demandes de support par mail la veille du rendu !
- La gestion des copies locales, notamment par la copie massive de répertoires à la souris. Rappelons que tous les sous-répertoires de votre copie locale contient des répertoires `.svn` (qui sont cachés lorsque vous utilisez votre navigateur classique) et écraser ces répertoires ou les copier n'importe comment cause des dommages irréversibles à votre copie locale. Faites attention à ne jamais copier des répertoires contenant des `.svn` vers votre copie locale.

Nous avons aussi noté la difficulté de certains à accéder au dépôt de chez eux, par exemple en cas d'absence de connexion Internet complète (les wifi "free" ne semblent pas permettre d'accès ssh vers l'extérieur, par exemple, par défaut, donc il est impossible d'effectuer des commandes SVN).

Modifications prévues pour l'année prochaine (IMA3 et IMA4)

- Mise en place d'une solution pour contourner les limitations des accès wifi "free".
- Mise en place d'une solution pour ne pas avoir à taper son mot de passe tout le temps lors d'une utilisation intensive de SVN.

Pour information, il existe des gestionnaires de version (par exemple git http://git-scm.com/) qui permettent de faire des "commit" hors ligne (donc en l'absence de connexion Internet), puis des commit en ligne.

1.3 Compilation - Exécution

Tous les projets compilent, à l'exception de 2. Ces deux projets ont la note de 7. Un des binômes concerné a néanmoins repris complètement le sujet et rendu un projet satisfaisant 5 jours après, ce qui ne peut pas rentrer en ligne de compte dans sa note mais prouve qu'un certain niveau est atteint en programmation.

Citons aussi 5 binômes pour lesquels les Makefile ne comportent pas de règle "clean" correcte.

Seuls 5 (en plus des 2 cités précedemment) projets ne s'exécutent pas correctement jusqu'au bout. Ces projets ont des notes inférieures à 11, à l'exception notable d'un projet qui n'a des problèmes de mémoire qu'après avoir exécuté un nombre de traitement très gros.

2 Fonctionnalités et solutions apportées

Comme le remarque un des binômes dans son rapport "une fois la structure de données correctement conçue et implémentée, les interrogations de ces structures et les statistiques sont très faciles et rapides à implémenter". Effectivement, le cœur du projet est la construction d'une SD suffisamment efficace et contenant les informations des textes lus.

2.1 Algorithmique, choix des SD

Quelques solutions algorithmique-ment mauvaises :

- Rechercher si un mot apparaît dans un texte, en ouvrant le texte à chaque appel. Les structures de données sont faites pour cela, pour stoker en interne des informations sur le texte pour ne plus avoir à parcourir celui-ci entièrement.
- Lorsqu'un mot est trouvé dans le texte, rechercher son nombre d'occurrences avant de le stocker.

– Les échanges de fichiers sont inutiles ici, on pouvait s'en sortir avec un unique parcours.
– Stocker pour chaque mot les informations de ligne+fichier, notamment le nom du fichier complet. Les noms peuvent être grands, une table de correspondance numéro ¡-¿ nom de fichier aurait été largement plus judicieuse.

2.2 Sur le code C

Solutions C d'implémentation

– La manipulation des fonctions de librairie `string`, qui peut être compliquée, est maintenant maîtrisée pour la majorité des binômes. Citons notamment les fonctions très intéressantes `strstr` pour repérer les sous-chaînes, et surtout `strtok` qui permet de découper une chaîne en éléments syntaxiques plus petits à l'aide de séparateurs.
– L'utilisation des fonctions de la librairie `dirent` (readDir notamment) n'était pas évidente, mais beaucoup l'ont fait avec succès.
– La fonction de Bernstein peut être implémentée de façon plus efficace que des multiplications par 33, en utilisant les décalages de bits (<< en C). Aucun groupe n'a utilisé cette implémentation.
– 1 binôme a implémenté des listes génériques avec void*, et 3/4 binômes les tables de hachage génériques en passant la fonction de hachage en paramètre. Cette fonctionnalité supplémentaire non vue en cours était évidemment réservée aux binômes les plus à l'aise, comme indiqué dans l'énoncé.

Erreurs/Maladresses de code souvent rencontrés

– En général, une fonction ne doit faire qu'un seul traitement. Si vous voulez savoir si une ligne est correcte, vous écrivez une fonction qui renvoie true ou false, si vous voulez en plus récupérer des infos, une autre fonction peut être judicieuse. N'oubliez pas non plus de nommer correctement ces fonctions. Une fonction qui contient 6/7 variables locales et qui fait plus de 30 lignes est en général une fonction qui peut être découpée en 2.
– Évitez les constantes en dur dans le code et utilisez les constantes symboliques.
– Évitez les goto. En général l'utilisation de `else` et de conditions adéquates dans les tests des `while` permet d'éviter ces goto.
– Récupérez toujours le pointeur de fichier après fopen, le pointeur de structure alloué après malloc, etc, et testez qu'il n'est pas null. Pareil avec les arguments de la ligne de commande, testez que les arguments sont bien du bon nombre, etc

Commentaires

– N'oubliez pas les commentaires dans les .h, pour chaque fonction d'interface, écrivez leur fonctionnalité. Ce sont ces fichiers d'interface qui sont les plus importants, ils font office de documentation minimale. En commentaire dans le .c un simili algo peut être décrit.
– Ne mettez dans les fichiers d'interface que les signatures des fonctions qui sont utilisées à l'extérieur.

2.3 Remarques importantes à lire !

Le projet que vous venez de faire met en oeuvre des solutions très spécifiques que vous avez apprises en PA. Néanmoins, si il est utile de savoir comment marche une solution *à la main*, il est important de savoir que des solutions plus évoluées existent. C'est l'objet des points développés ci-dessous.

– L'écriture des Makefile à la main (notamment lorsqu'ils s'appellent mutuellement) peut être judicieusement remplacé par l'utilisation d'un logiciel qui génère automatiquement

ledit Makefile (et qui évite de s'arracher les cheveux avec la syntaxe atroce des Makefile). Citons par exemple CMake (http://www.cmake.org/)
- La gestion d'un fichier html *à la main en C* est un exercice de style, il existe des façons plus rapides de récupérer des informations d'un fichier *qui a une certaine forme*. Parmi eux, citons les analyseurs Lex et Yacc qui résolvent le problème dans le cas général, et des analyseurs dédiés pour html, xml, etc. Dans la filière SC, au S8, le module d'Informatique fondamentale sera l'occasion de manipuler ce genre de solutions.
- Beaucoup d'entre vous ont du coder plusieurs types de listes, des listes d'entiers, des listes de mots, ..., et donc écrire une librairie à chaque fois. Si il existe une solution C assez lourde à mettre en oeuvre (les void*), ce travail fastidieux peut être évité avec des langages de plus haut niveau, qui permettent de faire des listes de type quelconque (Java, C++, ...). Mieux, les algorithmes de liste, hashtable ..., ne seront pas à réécrire. Vous n'aurez qu'à utiliser les librairies fournies *en vous rappelant les complexités des opérations pour chaque structure de données, quand même !*

Tests logiciels Le test des fonctionnalités est le point le moins bien réalisé durant les séances. Tout comme la compilation, qui est à effectuer souvent pour corriger les erreurs de type, les tests font partie du *processus de développement*. Dans l'idéal, chaque fonction doit être testée intensivement avant de l'utiliser dans un contexte plus large, dans le cas du projet Shakespeare par exemple :
- on peut tester la lecture des répertoires en imprimant chaque chaîne obtenue avec readdir. Si cette impression n'est pas correcte, cela ne sert à rien d'ouvrir les fichiers avec fopen.
- on peut tester les fonctions de "mise en minuscule", "enlèvement des tirets", sur des exemples précis avant de les utiliser dans la fonction qui traite un html complet.
- on peut tester les fonctions qui traitent les html durant leur développement par exemple : écriture de la boucle principale et impression des lignes complètes, ensuite ajout d'un test qui ne prend que les lignes qui conviennent, ensuite récupération et impression des mots uniquement (à comparer visuellement avec le contenu de la ligne imprimée), et ensuite seulement ajout dans la table de hachage ...
- à la fin, on peut regarder si le nombre d'occurrences d'un certain mot dans un fichier correspond au nombre d'occurrences dans le fichier par exemple en faisant `grep -c mot nomdufichierhtml`.
- on peut tester les traces mémoires des algorithmes en lançant *régulièrement* Valgrind.

Les tests logiciels qui permettent de s'assurer que votre programme fonctionne correctement sont à fournir dans le rapport.

Pensez au fait que le correcteur / la correctrice a plus de 20 projets à corriger, donc il faut que :
- l'on puisse voir rapidement quelles fonctionnalités sont OK (dans le README, dans le rapport)
- l'on puisse exécuter rapidement ces fonctionnalités.

Il est donc *indispensable* de fournir les fichiers de tests (ici quelques html d'input, pas tous !, et le fichier dico si votre programme l'utilise). De plus, il est indispensable que le README contienne des exemples de test. Ici un test est constitué d'une ligne de commande exemple ainsi que ce doit rentrer l'utilisateur pour tester les fonctionnalités. (vous pouvez aussi fournir un fichier à utiliser de la sorte : `./shakespeare <fichier`)

3 Sur les Rapports et Readme

Readme

- Un fichier Readme doit donner à l'utilisateur final l'ensemble des fonctionnalités du programme et les lignes de commande à taper pour lancer ces fonctionnalités.
- Si vous voulez mettre des instructions de compilation, il faut les mettre dans un autre fichier, par exemple un fichier texte s'appelant INSTALL.

Rapports Les rapports ont tous 5/6 pages comme demandé, et ont été globalement très satisfaisants.

Quelques remarques cependant :

- Respectez l'orthographe du nom Shakespeare (il y a un 'e' à la fin !)
- Vous ne racontez pas une histoire, restez factuels. Évitez les phrases à la première personne. Évitez les auto satisfactions trop flagrantes ou au contraire la dévalorisation excessive de votre travail.
- Utilisez les fonctionnalités de mise en page de votre logiciel de traitement de texte. Mettez des titres corrects et qui sont placés à l'aide de l'environnement "titre" (et non pas avec des tabulations).
- Relisez-vous !
- Un exemple d'application d'un algo est souvent plus efficace qu'une longue explication.
- Si votre fonction est une fonction classique, par exemple "rechercher dans une liste chaînée triée", donnez cette explication au lieu de reexpliquer un algo classique.
- Si une fonctionnalité n'est pas implémentée, dites le ! si une fonctionnalité ne marche pas, dites le !
- Si vous faites des tests de comparaison, faites une analyse de ces tests, sinon, ils ne servent à rien.

Voici les contenus qui ont été appréciés :

- Des dessins des structures utilisées !
- Une étude précise de la SD utilisée et de son utilité, certains ayant même fait une étude de complexité des fonctions utilisées souvent.
- Des tests sur les tailles des listes avec les différentes fonctions de hachage utilisées. Des graphiques sur la répartition de la taille de ces listes, et une conclusion sur ces graphiques.
- En annexe du rapport, un exemple d'exécution du programme.
- Des limitations clairement expliquées.

4 Conclusion

Ce sujet était ambitieux et non trivial (malgré la grande quantité de code déjà fournie), et la grande majorité des binômes l'a traité avec un nombre de fonctionnalités satisfaisant. Les binômes ayant traité un nombre de fonctionnalités minimale, mais ayant un code qui s'exécute correctement, ont des notes supérieures aux binômes qui ont essayé de tout faire en même temps, et qui se sont engluées dans leur code faute de stratégie de développement cohérente.

Les consignes ont été bien respectées, et des solutions algorithmiquement différentes ont été apportées. On peut aussi signaler 9 projets très bons (note supérieure à 16). La moyenne générale est de 13.9 et l'écart-type 3.5. Globalement, cette session de projet 2012 est un très bon cru.