Compte-rendu de correction du TP 2

Comme pour le TP 1, il vous a été remis un exemplaire corrigé et commenté de votre rendu ; ce document vient en complément pour expliciter les principaux problèmes détectés au cours de l'évaluation.

1 Non-respect des conseils et du compte-rendu du TP 1

Il est inadmissible qu'après le premier TP et l'effort conséquent pour rédiger un compte-rendu de correction, il reste autant d'erreurs ou de maladresses pourtant signalées et documentées. Le tableau ci-après récapitule les points revenant le plus souvent :

Document « conseils »	
2.1	indentation
2.3	lignes de plus de 80 colonnes
3.7	débordement de tampon
Document « compte-rendu correction TP 1 »	
2.1	programmes ne compilant pas
2.2	programmes non testés
2.8	absence de tests de retour
2.10	messages d'erreur sur la sortie standard
3.1	messages inutiles
3.2	abus de `static`
3.6	boucles `while` maladroites
3.8	constantes magiques
3.9	formattage des nombres
3.10	variables et fonctions mal nommées

Les erreurs répertoriées ci-dessus ont donc logiquement été plus sévèrement sanctionnées. Par exemple, 9 programmes sur 55 rendus ne compilaient pas : conformément à ce qui avait été annoncé dans le compte-rendu du TP 1 et l'énoncé du TP 2, ils n'ont pas été examinés.

Le script de test fourni sur Moodle a servi pour l'évaluation : seuls 25 programmes réussissent [1] le test ; les autres n'ont donc pas pu avoir la moyenne. Ces tests doivent être considérés comme étant une partie de la spécification du programme à réaliser au même titre que l'énoncé, et ne doivent surtout pas être modifiés pour correspondre à votre implémentation (comme cela a été fait une fois).

Les points cités dans le tableau ci-dessus renvoient à des documents en votre possession. À l'exception de quelques uns particulièrement fréquents, ils ne seront pas davantage explicités dans la suite.

2 Les erreurs les plus fréquentes

2.1 Mauvais type pour la taille des fichiers

Le type utilisé pour représenter la taille d'un fichier (cf notice simplifiée, page 16) est `off_t`. Tout autre type (`int`, `long int`, `size_t`) est à proscrire, même si la taille est la même sur votre implémentation.

2.2 Débordement de tampon

Malgré les avertissements dans le document « conseils » ou en cours et TP, de très nombreux programmes omettent de vérifier que la place est suffisante dans les tableaux, pour stocker le chemin composé du nom du

1. Ce qui n'empêche pas qu'ils peuvent receler des erreurs sur des points non testés.

répertoire et du nom de l'entrée trouvée, ou pour ajouter un nouveau fichier dans le tableau utilisé lors du tri. Pour l'anecdote, notons qu'un programme a testé la place disponible, mais après effectué le débordement !

2.3 Absence de test de retour

Toute fonction (primitive système ou fonction de bibliothèque) pouvant échouer (i.e. renvoyant un code de retour) doit être systématiquement testée, comme cela a pourtant été rappelé dans le précédent compte-rendu de correction. Ne pas le faire vous expose à des problèmes graves (dans vos programmes, mais également pour votre scolarité).

Cette erreur ainsi que la précédente (débordement de tampon) ont été sévèrement sanctionnées.

2.4 Messages d'erreur

Les messages d'erreur doivent être utiles, explicites et précis.

- Si le nombre d'arguments fourni n'est pas correct, indiquer à l'utilisateur qu'il s'est trompé est sympathique, mais lui indiquer la bonne syntaxe lui est plus utile. C'est pour cela qu'on trouve souvent des messages tels que `fprintf(stderr,"usage: %s répertoire",argv[0])` dans les commandes du système.
- Les messages d'erreur doivent indiquer la raison de l'erreur : il ne suffit pas d'indiquer qu'il y a un problème avec `opendir`, encore faut-il indiquer la raison exacte, c'est-à-dire utiliser `errno` (par exemple avec `perror`). Beaucoup de fonctions de bibliothèque, comme par exemple `malloc`, modifient également `errno` et permettent l'utilisation de `perror`. En cas de doute, vérifiez avec le manuel en ligne (dans la section « Errors »).
- Lorsqu'une erreur ne résulte pas d'une primitive système (exemples : erreur d'argument de l'utilisateur, chemin trop long provoquant un débordement de tampon, etc.), il ne faut surtout pas appeler `perror`.

Bien évidemment, il faut impérativement signaler les erreurs et ne pas les passer sous silence : dans quelques cas, des programmes testent la condition d'erreur (arguments, retour de fonction) mais s'arrêtent sans message. Comment l'utilisateur peut-il alors savoir ce qui s'est passé ?

Notez qu'une fonction `raler` basique est fournie dans les transparents de cours et une version plus élaborée est fournie dans les corrections du TP 1 et 2. Vous pouvez les réutiliser sans modération.

2.5 Utilisation de `lstat`

Pour ignorer les liens symboliques comme le suggère l'énoncé, ainsi que pour satisfaire le test baptisé « gros volume », il fallait utiliser la primitive `lstat` et non `stat` qui « suit » les liens symboliques de manière transparente.

Par ailleurs, l'appel aux primitives systèmes doit être minimisé : par exemple, l'utilisation de la bufferisation avec `getchar` permet de minimiser le nombre d'appels à `read`. Il en va de même avec `stat` et `lstat` : elles doivent analyser tout un chemin pour lire l'inode du fichier et retourner les attributs. Les appels à ces primitives doivent être minimisés : en toute rigueur, un seul appel à `lstat` devrait être réalisé pour chaque objet de l'arborescence. Certains font 2, 3 voire 6 appels pour chaque objet. Le pire est atteint avec des appels dans la fonction de comparaison utilisée pour `qsort` : sachant que la complexité moyenne du tri mis en œuvre par `qsort` est en $O(n \log_2 n)$, où n est la taille du tableau à trier, le nombre d'appels à `lstat` sera en $O(2n \log_2 n)$, ce qui est considérable !

2.6 Parcours récursif et dimensionnement du tableau des fichiers

Le tableau servant à stocker tous les fichiers en vue du tri doit être alloué dynamiquement. Pour ce faire, plusieurs méthodes ont été constatées.

- Dimensionnement du tableau avec une taille définie dans le programme, sans aucun redimensionnement prévu. Ce cas ne gère pas toutes les arborescences, car il s'en trouvera toujours de plus grandes que ce que le programmeur avait prévu ; par ailleurs, pour les petites arborescences, beaucoup de place est allouée pour peu d'entrées utilisées. Cette méthode est donc à proscrire. Notons pour l'anecdote

qu'un programme a alloué un nombre d'entrées juste suffisant pour passer le test « gros volume », ce qui peut être considéré comme de la triche.
— Parcours récursif double : le premier pour comptabiliser le nombre de fichiers, suivi de l'allocation du tableau avec le nombre d'entrées trouvées, puis deuxième parcours pour obtenir les tailles des fichiers. Outre que cela oblige à faire deux appels à `lstat` pour chaque objet, cette méthode est très risquée dans la mesure où un autre processus peut agir sur le nombre de fichiers entre les deux parcours : si un fichier est ajouté, par exemple, cela conduira inévitablement à un débordement de tampon.
— Parcours récursif simple, avec réallocation du tableau en fonction du nombre d'entrées trouvées. Pour minimiser le nombre d'appels à `realloc`, certains ont eu la bonne idée d'implémenter une technique de réallocation par blocs, soit linéaire (incrémentation de n entrées à chaque fois), soit exponentielle (incrémentation de n, puis de $2n$, puis de 2^2n, etc.). Le parcours récursif simple est évidemment la méthode à privilégier.

Enfin, signalons qu'un programme a effectué un parcours à base de changement de répertoire avec `chdir`. La plupart des parcours d'arborescences à base de `chdir` se révèlent faux. Celui-là n'a pas échappé à la règle.

2.7 Mauvaise utilisation du résultat de `readdir`

Certains ont utilisé le champ `d_type` de la structure `dirent` retournée par `readdir` : certes, ce champ permet d'éviter un appel à `stat` mais il n'est pas compatible POSIX. La notice simplifiée contient un avertissement explicite (cf page 50) indiquant qu'il faut proscrire son utilisation, ce que la page de manuel de `readdir` sur Linux confirme en précisant que tous les systèmes de fichiers ne le supportent pas.

2.8 Cas particuliers

Plusieurs cas particuliers ont parfois été implémentés : ils n'ont pas lieu d'être, et ne font que rendre les programmes plus compliqués sans apporter une quelconque valeur ajoutée.

Le premier concerne les répertoires vides : certains programmes testent en amont le cas où le répertoire donné en argument est vide. Ce cas n'est absolument pas un cas particulier, les répertoires vides pouvant d'ailleurs exister au plus profond des arborescences explorées, la liste des fichiers étant alors naturellement réduite à 0.

Le deuxième cas particulier concerne le test de répertoire ou d'accessibilité concernant l'argument : celui-ci est testé avec `stat` (pour le type répertoire) ou `access` (pour l'accessibilité en lecture) préalablement au parcours récursif. Là encore, il s'agit d'un cas particulier qui n'a pas lieu d'être puisque ces deux conditions sont testées par `opendir` lors du parcours récursif.

Le troisième et dernier cas particulier concerne le traitement des « / » superflus. Ainsi, par exemple, si l'argument (ou n'importe quel répertoire pour certains) contient le séparateur « / » en fin de chaîne, un test est effectué pour le retirer, afin sans doute d'éviter d'avoir des chemins comme `/tmp//toto`. Cependant, ce test est inutile car on peut très bien avoir des chemins comme `/tmp//toto` (ou même `////tmp/////toto`) sans que cela soit gênant. Et pour que ce cas particulier soit rigoureusement traité, il aurait fallu supprimer tous les « / » en fin de chaîne et pas seulement le dernier.

Dans l'ensemble, ces cas particuliers n'ont pas donné lieu à retrait de point, les auteurs ayant été déjà suffisamment punis par le travail superflu qu'ils se sont eux-mêmes généré.

2.9 Mauvaise utilisation de `strncpy` et `strncat`

Certains, qu'ils aient traité correctement ou non les débordements de tableau, utilisent `strncpy` et `strncat` sans vraiment comprendre leur intérêt. Par exemple :
— `strncpy (chemin, rep, strlen (rep))` : le troisième argument doit indiquer la place disponible dans `chemin`, et non la longueur de `rep`. Avec ces arguments, cet appel à `strcpy` revient à `strcpy (chemin, rep)`, mais sans recopier le `\0` final...
— `strncat (chemin, d->d_name, strlen (d->d_name))` : là encore, le troisième argument doit indiquer la place disponible dans `chemin` (soit la taille de `chemin` moins ce qui y a déjà été mis moins un). Ici aussi, cet appel revient à faire un `strcat` simple (toujours sans le `\0` final).

Vous êtes invités à consulter la correction de ce TP 2 pour une méthode générale pour éviter les débordements de tampon sur des chaînes de caractères.

2.10 Utilisation répétitive d'une fonction

Dans certains programmes, on voit des appels successifs à une fonction renvoyant toujours le même résultat, comme par exemple trois appels à `strlen(arg)` en deux lignes, ou alors un appel à `nbChiffres(n)` dans le test d'une boucle `for`, donc exécuté à chaque tour de la boucle. Éliminez ces inefficacités.

3 Améliorations de style

Comme pour le TP 1, vous trouverez ici des suggestions d'amélioration de vos programmes.

3.1 Abus de `typedef`

Certains utilisent systématiquement des `typedef` pour cacher les structures de données. Cela suppose une « indirection » supplémentaire pour le lecteur (« ah oui, cette variable est déclarée avec le type `SzFichier`, qui correspond en fait à la structure `fichier` ») : le lecteur doit retenir à la fois le nom du type, le nom de la structure et la correspondance entre les deux, parfois encore compliquée par la présence d'un pointeur.

Pour que l'utilisation de `typedef` soit réellement intéressante, il faut que le type soit *abstrait* et le code séparé en deux : une partie « implémentation » définit les opérations sur les objets du type, et l'autre partie n'utilise que les opérations définies (utilisation du type abstrait). Pensez par exemple au type `FILE` dont vous n'avez pas à connaître les détails intimes pour utiliser `fopen`. Ici, dans un si petit programme (165 lignes en moyenne), l'utilisation de `typedef` ne se justifie à notre avis pas.

3.2 Sizeof (char) == 1

On rencontre encore souvent des lignes comme : `p = (char *) malloc (n * sizeof (char))`

La conversion de type n'est pas nécessaire[2] car `malloc` renvoie un résultat de type `void *`, compatible avec tous les types pointeurs. De plus, `sizeof(char)` vaut 1 par la définition du langage C. Alors, avouez que `p=malloc(n)` est plus lisible et plus simple, non ?

3.3 Non-booléens et opérateurs booléens

On trouve quelquefois des tests de la forme :

- `d = opendir (chemin) ; if (! d) raler (chemin) ;`
- `if (! strcmp (d->d_name, ".") && ...)`

Dans ces deux exemples, `opendir` et `strcmp` ne renvoient pas une valeur sémantiquement booléenne : l'utilisation d'opérateurs booléens (« `!` » et « `&&` ») induit le lecteur en erreur et est donc à proscrire.

De manière inverse, quand vous avez une valeur booléenne, ne la comparez pas avec une valeur précise. Par exemple : `if (S_ISREG (...) == 1)` a toutes les chances de ne jamais être vrai.

4 Conclusion

Comme précédemment, n'hésitez pas à vous inspirer de ce compte-rendu pour améliorer vos programmes futurs. En cas de doute où de question, interrogez vos enseignants.

2. Elle n'est *plus* nécessaire depuis la norme C89 de 1989, ce qui ne nous rajeunit guère...