La modularité et le débogage en C

Par Jérôme Collin

Souvent, en C++, la question de la modularité se règle assez facilement. On dit: « déclarez une classe dans un .h et la fonctionnalité dans le .cpp ». Il est vrai que cette affirmation règle passablement de problèmes. La première raison est qu'une classe n'est qu'un nouveau type introduit dans le cours du programme. Aucun espace mémoire n'est encore réservé à une variable par cette déclaration. Ce n'est que la déclaration d'un objet (dans le .cpp) qui changera les choses. De plus, la classe permet de regrouper les variables et la façon dont elles sont manipulées par les fonctions membres. C'est souvent une façon efficace de fonctionner. C'est même une raison suffisante pour adopter C++, par rapport à C, comme langage de développement pour plusieurs. Quelques patrons de conceptions, comme le singleton par exemple, permettent d'éviter également bien des problèmes de modularité.

Mais qu'en est-il de la modularité en pur C ? D'ailleurs, bien des sites nous rapportent que le bon vieux langage est encore beaucoup utilisé. En fait, l'IEEE le place presque à égalité dans l'usage par rapport à C++ et encore parmi les langages les plus utilisés, même aujourd'hui. Beaucoup d'autres nouvelles qui passent ici et là (particulièrement sur slashdot.org) arrivent toujours à peu près aux mêmes chiffres. De toute façon, toute question qui se pose en C a de bonnes chances de se poser aussi en C++ puisqu'un objet est aussi une variable et que les fonctions continuent d'exister en C++.

Le préprocesseur CPP

Un concept important à garder à l'esprit est que l'inclusion d'un fichier .h n'est qu'un mécanisme de « copier-coller » par le préprocesseur, CPP. En d'autres termes, tout ce qui se trouve dans un fichier .h, est copié en lieu et place de la directive #include «… ». La compilation au sens très strict du terme par GCC ne voit donc que le résultat de cette expansion. Il peut être intéressant d'ailleurs, au moins une fois, de voir ce que le compilateur reçoit vraiment comme entrée, après cette expansion. Exécutez la commande suivante sur un de vos fichiers C et observez le résultat, par curiosité :

```
% cpp -E monFichier.c
```

Alors, quel est le problème ? On pourrait, en théorie, transférer tout le code C dans un fichier .h et avoir une seule ligne dans un .c qui inclut le .h. Ce n'est pas particulièrement élégant, mais ça marche… pour un temps! Qu'est-ce qui arrivera quand deux fichiers .c différents incluront le même fichier .h ? Si des variables ou des fonctions sont décrites dans le .h, elles se retrouveront dans 2 fichiers à la compilation. La compilation fonctionnera toujours. Par contre, à l'édition de liens (*linking*) pour former l'exécutable, il y aura redéfinition de symboles et ça ne passera pas. On pourrait être alors tenté de placer le qualificatif «static» aux déclarations ce qui a pour effet de leur donner une visibilité qui est confinée uniquement au seul fichier .c dans lequel elles se trouvent. Par contre, on se retrouve avec des fonctions et variables en double (en

même potentiellement en triple et en quadruple…) si on continue de faire des inclusions dans d'autres fichiers .c. Il en résultera la confusion la plus totale. Modifier LA variable perdra de son sens puisqu'il en existe plusieurs copies dans le code… même si ça compile toujours!

Les unités de compilation

Avant de poursuivre sur ce problème et sa solution, il faudrait commencer par regarder pourquoi les fichiers .c et .h ont été inventés. La première raison est évidemment qu'on ne peut pas écrire de gros programmes dans un seul fichier .c. Il est préférable de répartir le code en plusieurs fichiers .c pour être capable de réutiliser certains d'entre eux dans différents projets (possiblement en les plaçant dans une librairie). De plus, on évite de se déplacer constamment dans un éditeur qui manipule plusieurs milliers de lignes de code. Mais alors, quel est le rôle d'un fichier .h au fait ? On pourrait penser que le fichier .h se contente de servir à déclarer « des choses tranquilles » comme des constantes et des macros (*#define…).* Par contre, son véritable rôle est de servir de «liens» entre les fichiers .c tout en étant à la base de la modularité.

Un fichier .c mène généralement à la formation d'un fichier .o une fois la compilation terminée. On parle donc ici d'unités de compilation, terme utilisé pour marquer le confinement du code à des fichiers séparés. Le problème est de relier ces unités entre elles d'une certaine façon.

Disposition dans les répertoires

Il peut y avoir plusieurs façons de disposer les fichiers .h par rapport aux fichiers .c, mais il y en a deux qui sont plus généralement utilisées. La première est de regrouper tous les .h dans un même répertoire (ce répertoire est souvent appelé « include », évidemment) dans un endroit d'assez haut niveau dans l'arborescence des fichiers sources du projet. La seconde est d'avoir chaque fichier .h qui accompagne un fichier .c dans le même répertoire. Normalement, une disposition hybride tirant parti de ces extrêmes est utilisée dans un même projet.

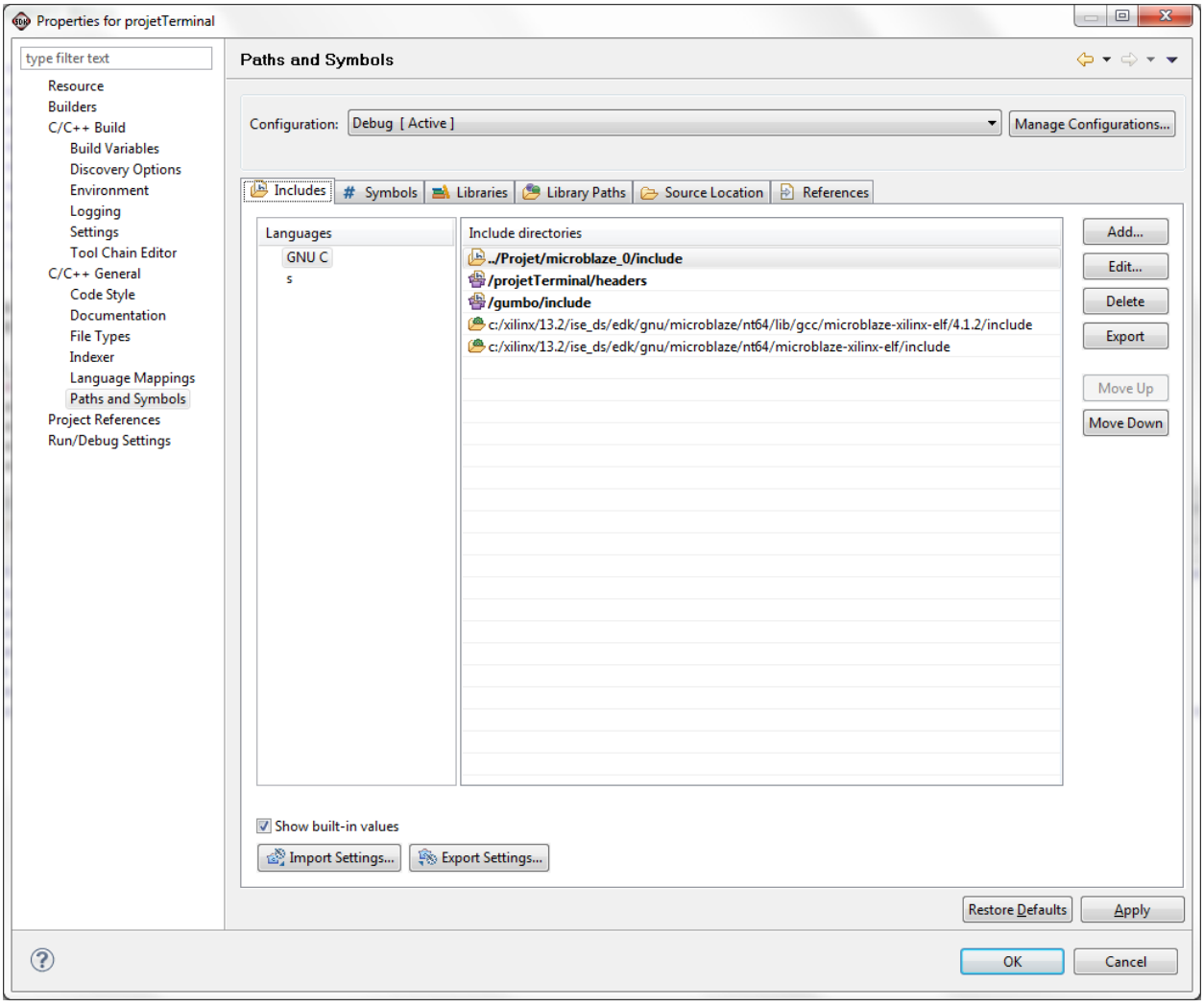
Il est très mal vu d'avoir un fichier .h inclus de la façon suivante dans un fichier .c :

```
#include "../../../monRepertoire/mon.h"
```

Il s'établit alors une dépendance hiérarchique entre les deux fichiers et une réorganisation des répertoires dans un projet apporte un lot important de corrections d'erreurs de compilation qui s'en suivent. Le nombre de fichiers .c peut devenir impressionnant avec le temps dans un projet et les dépendances beaucoup plus compliquées qu'estimées en premier lieu. Il est nettement préférable de placer dans un Makefile une liste des répertoires possibles où localiser les fichiers .h. L'argument -I (i majuscule) passé au compilateur réglera le problème de recherche de chemins vers les fichiers .h.

Dans SDK, il faut aller dans le menu « Project » et le sous-menu « Properties » pour accéder à une fenêtre dont la colonne de gauche présentera des possibilités pour ajuster des paramètres de compilation. Sous « C/C++ General » et ensuite « Paths and Symbols », on peut ajuster des chemins vers des répertoires à consulter par le compilateur (le préprocesseur, pour être précis) pour trouver les fichiers .h d'un module ou d'une librairie. L'exemple ci-dessous montre que le répertoire /projetTerminal/headers sera inclus comme sous-répertoire, à partir de la racine du projet, dans la recherche de répertoire de .h. Il y aura donc une ligne d'appel du compilateur dans les logs qui pourrait ressembler à ceci :

```
gcc ... -I "z:\sandbox\inf3995-03\projet\sdk\projetTerminal\headers" ...
```



La possibilité de faire un « git clone » de votre entrepôt n'importe où sans référence à un répertoire particulier devient intéressante en plus de garder la possibilité de réaménager vos fichiers dans le temps, au besoin. Il devient aussi plus facile d'isoler dans un répertoire du code dans votre projet (comme une librairie trouvée sur internet

effectuant un travail particulier), mais d'y inclure les fichiers .h dans le reste de votre application sans trop de dépendances à une arborescence de répertoires particulière.

Le cas des fonctions

Une ligne telle que celle-ci dans un fichier .h est souvent rencontrée :

```
int maFunction1 (int a, float b);
```

Tout ce que fait cette ligne est de dire qu'une fonction ayant ce nom prend deux paramètres en entrée et retourne un entier. C'est sa signature, un genre de contrat entre deux partis. Un parti devra implémenter la fonction quelque part dans un fichier .c, une seule fois, et un autre parti pourra appeler la fonction autant de fois que nécessaire dans divers fichiers .c. La compilation vérifiera l'intégrité de l'un et l'autre des partis par rapport à la signature définie dans le .h et inclus par tous les fichiers .c impliqués. Si l'appel de cette fonction se fait avec 3 arguments, le compilateur va rappeler le programmeur à l'ordre. Si l'implémentation de la fonction ne respecte pas la signature, le compilateur se plaindra aussi. S'il y a 2 implémentations ou plus pour la fonction, on se retrouve dans le cas mentionné plus haut et l'éditeur de liens détectera le problème.

On voit donc que le rôle de la signature est de garantir que `maFunction` corresponde à une «réalité unique» même si cet identifiant apparaît dans divers fichiers sources. Dans certaines situations, on veut l'inverse. On veut conserver une fonction très locale à un fichier .c sans la possibilité de l'appeler d'une autre unité de compilation. Dans ce cas, on placera le qualificatif «static» avant la fonction. Très souvent également, on n'aura pas à déclarer la signature d'une telle fonction puisque sa déclaration et ses appels se font dans le même fichier. On combinera la déclaration et l'implémentation au même endroit, souvent vers le haut d'un fichier C :

```
static int maFunction2 (int a, int b)
{
   if (a > b)
      return a;
   return b;
}
```

Se souvenir qu'il faut tout de même avoir une fonction déclarée (en d'autres termes, au minimum la signature) avant de l'appeler pour éviter des problèmes.

Très souvent, les programmeurs prudents définissent leurs fonctions locales à un fichier .c pour être « static » explicitement. De cette manière, il devient impossible d'appeler ces fonctions par accident à l'extérieur de l'unité de compilation. Cependant, comme on l'a vu précédemment, il devient très dangereux d'employer le qualificatif «static» dans un fichier .h en C.

Les variables

Peut-on avoir l'équivalent d'une déclaration avec une signature de fonction, mais pour des variables globales ? Absolument. Il faudra écrire dans un fichier .h :

```
extern int maVariable1;
```

Ici, l'emploi de « extern » ne mène pas le compilateur à réserver de l'espace mémoire pour la variable. Il est simplement dit qu'il existe une variable globale dans une unité de compilation quelque part ailleurs dans le code qui a ce nom et ce type et qui peut être utilisée. Un seul fichier .c devra cependant définir la variable une seule fois pour garantir son existence unique de la façon suivante :

```
int maVariable1;
```

Dans tous les autres fichiers .c, l'utilisation de l'identificateur maVariable1 fera référence à cette variable unique. Si elle n'est jamais définie, l'éditeur de liens dira que le symbole n'est pas défini. Il y aura erreur aussi si elle est définie plus d'une fois.

Cette façon d'organiser le code demande de la discipline et de l'ordre dans le code. C'est ce qui a amené à la recommandation du bannissement des variables globales par plusieurs. On leur préfère des variables locales aux unités de compilation (et donc souvent statiques) et leur manipulation par des fonctions déclarées dans un .h et implémentées dans un fichier .c. Tout de même, il est bon de savoir d'où viennent les mécanismes et surtout les sources d'erreurs possibles.

Il reste des situations en C où l'emploi de variables globales peut devenir inévitable (cas des interruptions, par exemple). Regrouper les variables globales dans une structure (`struct`) peut être intéressant pour éviter l'éparpillement.

Stratégie de débogage simple

Maintenant que le mécanisme de fonctionnement du préprocesseur a été présenté, on pourrait se demander comment tirer avantage de son mécanisme de «copier – coller» pour une stratégie de débogage uniforme et structurée. L'extrait suivant est intéressant lorsque placé dans un fichier ayant comme nom, par exemple, `debug.h` :

```
#ifdef DEBUG
#  define DEBUG_PRINT(x) printf (x)
#else
#  define DEBUG_PRINT(x) do {} while (0)
#endif
```

Et son utilisation de la façon suivante dans tout fichier .c ou cpp (oui, c'est utile aussi en C++) :

```
DEBUG_PRINT(("var1: %d; var2: %d; str: %s\n", var1, var2, str));
```

Notez que le «do-while» n'a pas à être placé. Mon opinion est qu'il est bien de le placer pour montrer qu'on ne veut rien faire. Il sera retiré par l'optimisation (code « mort ») du compilateur de toute façon si `DEBUG` n'est pas définie. Donc, «en production», aucun code de débogage ne se trouve ajouté au code utile. Si `DEBUG` est définie (avant l'inclusion de `debug.h`), le mécanisme de copier-coller introduira un message à afficher à la sortie standard. Consultez la référence de StackOverflow à la fin de ce document pour d'autres possibilités d'organisations de messages de débogage.

L'utilisation des macros prédéfinies par le langage C, `__LINE__`, `__FILE__` et `__FUNCTION__` peuvent permettre de raffiner les messages à afficher. On peut aussi définir plus d'une macro pour l'affichage de messages de débogage selon la sévérité (*info, warning, error*, etc…). L'option –DDEBUG sur la ligne de commande (ou son absence) lors de l'appel du compilateur permet souvent de choisir entre l'un ou l'autre des cas souhaités. Dans SDK, on peut ajouter l'option pratiquement au même endroit où on précise les répertoires vers les fichiers «include» à considérer.

Utilité limitée des macros

Plusieurs auteurs recommandent d'éviter les plus possible l'utilisation de macros de manière générale. Il y a quelques décennies, on les utilisait pour des fins d'optimisation de code. De meilleures techniques de compilation et l'introduction du `inline` et des constantes en C++ rendent inutile cette technique de programmation. Les macros sont aussi un mécanisme de «copier-coller». Il peut arriver qu'un code utilisant des macros devienne incompréhensible et que le résultat final ne soit pas celui attendu alors que son écriture avec de simples fonctions C serait plus claire.

Conclusion

À la lumière de ce qui vient d'être exposé, voici comment devrait être structurée la modularité en langage C :

- Limiter l'emploi de variables globales;
- Donc, préférer déclarer des variables statiques locales aux unités de compilation;
- Pensez à regrouper les variables globales si leur emploi est inévitable;
- Déclarer uniquement la signature des fonctions dans les fichiers .h et déclarer uniquement les fonctions qui doivent être utilisées à l'extérieur de l'unité de compilation;
- Déclarer les variables et les fonctions locales à une unité de compilation comme étant statiques pour empêcher leur visibilité à l'extérieur;
- Avoir une ou quelques macros uniformes pour le débogage;
- Éviter le plus possible les macros et privilégier l'utilisation de fonctions ou de constantes le plus possible.

On pourrait ajouter que, tout comme en C++, il est préférable qu'un concept unique, une fonctionnalité ou une entité bien définie soit complètement modélisé dans l'utilisation d'une paire fichier .h (déclaration) et fichier .c (implémentation).

Références

Al Kelley et Ira Pohl, Benjamin Cummings, *A Book on C, Programming in C*, 4ème édition, 1998.

http://stackoverflow.com/questions/1941307/c-debug-print-macros [consulté le 19 octobre 2014]

http://www.lemoda.net/c/line-file-func/index.html [consulté le 19 octobre 2014]