Université de Lille 1 SCIENCES ET TECHNOLOGIES

Licence d'informatique
2e année
Module de BPC

Bases de la Programmation en C

Équipe pédagogique de BPC [1]

février 2021

Ce sujet est disponible en version html à l'adresse suivante :
`https://www.fil.univ-lille1.fr/~ballabriga/bpc/tdtp/tp_theme2.html`.

Table des matières

1. d'après un document CC-BY-SA de Gilles GRIMAUD et Philippe MARQUET, 2018-2020.

Ce support sert à la fois pour les TDs et les TPs. Sauf mention contraire, chaque exercice est à préparer en TD, et ensuite à implémenter, compiler et tester en séance de TP.

Semaine 3 : Les outils de compilation (1/2)

Pour ce thème 2 du module BPC, vous allez travailler sur un autre dépot git :

`https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/theme2-g<i>-y<yy>`

où `<i>` est votre numéro de groupe, et `<yy>` est l'année.

1 Le préprocesseur

Par commodité le langage C exploite un préprocesseur. Un préprocesseur de texte est un logiciel qui parcourt un fichier source pour le transformer en un autre source, avant de donner ce nouveau source au véritable compilateur. A priori, le préprocesseur utilisé par le langage C est destiné au sources écrites en C, mais aussi en C++ et en Objective C. Cependant il peut être utilisé sur d'autres langages pourvu que ces derniers utilisent des conventions de commentaires et de chaînes de caractères équivalentes à celles du langage C. Il pourrait même être utilisé sur des sources Java, même si ce n'est pas l'usage.

1.1 Substitution de texte, les macros du préprocesseur

Le préprocesseur permet donc de remplacer des symboles d'un *fichier source* par « autre chose », c'est-à-dire par une chaîne de caractères quelconque que le compilateur traitera en lieu et place du symbole.

Considérez le code source suivant disponible dans le fichier `macro1.c` (on travaille pour cette section sur le processeur dans le sous-répertoire `prepro/` du dépôt Git) :

```
extern int putchar(int c);

int main()
{
    int u=68;
    putchar(u);                 /* premier char */
    putchar(T);                 /* deuxième char */
}
```

Exercice 1 (Ceci n'est pas défini)
Travail à réaliser en TD :
Pourquoi ce programme ne peut-il pas fonctionner ? □

Il est possible d'indiquer au compilateur que le symbole `T` doit être transformé en autre chose.

Ces symboles qui sont simplement, syntaxiquement, remplacés par autre chose sont appelés des *macros*.

Il existe deux manières de définir une macro.

Exercice 2 (Directive de préprocesseur en ligne de commande)
La première solution consiste à indiquer en ligne de commande que l'on souhaite que le préprocesseur remplace un symbole par autre chose :

```
bash$ gcc macro1.c -DT=65
```

Sur cet exemple, on indique au préprocesseur du compilateur C que les symboles `T` doivent être remplacés par des 65.

Travail à réaliser en TD : Expliquez ce qu'affiche le programme ainsi compilé ? □

Travail à réaliser en TP : Testez ce que vous avez préparé en TD.

La seconde solution consiste à utiliser la directive `#define`. dans le fichier source. À l'instar de la solution précédente, cette directive permet d'indiquer au préprocesseur qu'il doit remplacer un symbole par autre chose. Ajoutez la ligne suivante en tête du fichier source :

```
#define T (65+1)
```

Notez qu'il ne s'agit pas d'une instruction du langage C. Elle ne se termine pas par un `;`. Elle ne correspond pas à une affectation, il n'y a pas de =... Ici on dit simplement au préprocesseur du compilateur, qu'à partir de maintenant, à chaque fois qu'il rencontre `T`, il doit le remplacer par `(65+1)`.

Exercice 3 (Directive `#define` de préprocesseur)
Travail à réaliser en TD : Expliquez ce qu'affiche le programme ainsi compilé ?
Travail à réaliser en TP : Testez ce que vous avez préparé en TD.

Il est possible de demander un compilateur de n'exécuter que le traitement préprocesseur et de produire le fichier résultant de ce traitement sur la sortie standard (ou dans un fichier). Il s'agit de l'option `-E`.

Exercice 4 (Sortie du préprocesseur)
Travail à réaliser en TD : Quel est le flag à passer à gcc pour observer la sortie du préprocesseur ?
Travail à réaliser en TP : Observez la sortie du préprocesseur. Les lignes qui commencent par un `#` sont des informations que le préprocesseur transmet au compilateur. Pour cette question, ne les prenez pas en compte. Vous pouvez retirez ces lignes en filtrant la sortie de `gcc` avec

```
... | grep -v "#"
```

qui ne retiendra que les lignes qui ne contiennent pas un `#`. Comment le préprocesseur a modifié le source ? □

Dans les exemples précédents la substitution opère sur ce qui semble être une simple variable. Il n'en est rien. Il s'agit en fait d'une substitution syntaxique, qui n'a aucune valeur sémantique : `T` n'est pas une variable, elle n'a d'ailleurs pas été déclarée, mais un symbole qui a été remplacé par `(65+1)`.

Il est donc possible de remplacer n'importe quel symbole par autre chose, pourvu que ce soit par quelque chose que le compilateur C définisse, comme une fonction, une instruction élémentaire, ou une séquence d'instructions.

Il existe un certain nombre de macros qui sont prédéfinies. Elles permettent d'instrumenter le code source. Les plus utiles sont les macros implicites suivantes :

`__LINE__` donne le numéro de la ligne courante au moment de la compilation ;

`__FILE__` donne le nom du fichier courant au moment de la compilation ;

`__DATE__` donne la date courante, au moment de la compilation ;

`__TIME__` donne l'heure minute seconde courante, au moment de la compilation.

Exercice 5 (Macros prédéfinies)
En réutilisant votre fonction `putdec()`, afficher la ligne courante, au début de la fonction `main()`. Déclarez aussi une variable globale `ln` qui sera initialisée par

```
int ln=__LINE__;
```

et affichez la valeur de `ln` dans la fonction `main()` avant, et après le `putdec()`. Qu'observez vous ? □

Travail à réaliser en TD : En TD, proposez une fonction `main()` qui réalise ce qui est demandé.
Travail à réaliser en TP : En TP, implémentez et testez, pour vérifier que le numéro de ligne est bien affiché.

Le système de substitution de symboles permet de définir des paramètres de substitution. On parle alors de macros paramétrées. L'exemple suivant illustre cette possibilité (fichier `macrop.c`) :

```
extern int putchar(int c);

#define bit(i,j) (i>>j)&1

int main() {
    int i=16;
    int i0=bit(i,0);
    int i4=bit(i,4);
    putchar('0'+i0);
    putchar('0'+i4);
    putchar('\n');
    putchar('0'+bit(i,0));
    putchar('0'+bit(i,4));
    putchar('\n');
}
```

Dans ce source `macrop.c`, la macro `bit` est paramétrée par deux informations, `i` et `j`. Dans une lecture maladroite, on pourrait percevoir cette macro comme une fonction qui prend 2 paramètres (`i` et `j`) et qui renvoie la valeur du `j`-ième bit de `i` (0 ou 1).

Cependant ce n'est pas le sens de cette ligne de C. Cette ligne doit être comprise comme une indication au préprocesseur pour qu'il remplace, partout dans le code source, les occurrences de `bit(`x`,`y`)` par des `(`x`>>`y`)&1`.

Exercice 6 (Macros paramétrées)
Travail à réaliser en TD : Consultez le source du programme `macrop.c`. Essayez de savoir ce qu'il va afficher.

Travail à réaliser en TP : Compilez et exécuter ce programme `macrop.c`. Expliquez l'affichage obtenu, est-ce que conforme à ce que vous aviez prédit en TD ?

Pour vous aider à mieux comprendre ce qui ce passe exactement, vous pouvez consulter le code produit par le préprocesseur, option `-E` du compilateur. □

1.2 Suppression de lignes de code, la compilation conditionnelle

Le préprocesseur permet aussi de supprimer du fichier source des lignes de codes, selon des conditions qui seront appréciées au moment de la compilation. Il est par exemple possible que le même fichier source soit compilé avec, ou sans, certaines fonctions, selon que des macros soient définies ou pas dans la ligne de commande lors de la compilation.

Le préprocesseur permet de conditionner le fait que certaines portions de codes seront (ou pas) données au compilateur. Le développeur peut par exemple conditionner la compilation d'une portion du code source, en fonction de la valeur d'une macro particulière.

Considérez le programme suivant

```
extern int putchar(int c);

#if NO_LOG==1

int logchar(int c) {
    return 0;
}

#else

int logchar(int c) {
    return putchar(c);
}

#endif

int main() {
    int i=1;
    i=3*i;
```

```
    logchar('0'+i);
    return i;
}
```

disponible dans le fichier `compcond.c` (compilation conditionnelle).

Exercice 7 (Un *fichier source* pour deux programmes)
Travail à réaliser en TD : Donnez la ligne de compilation pour que la macro `NO_LOG` produise un code qui ne va pas afficher le resultat avant de le retourner.

Travail à réaliser en TP : Utilisez l'option `-E` pour vous assurer que vous avez compilé le code voulu. □

Notez que l'expression évaluée par le préprocesseur ne peut porter que sur des constantes et sur des macros. Le préprocesseur ne peut pas apprécier la valeur d'une variable C par exemple. Autrement dit, le préprocesseur ne « comprends rien » au langage C.

En général nous n'avons pas besoin de tester une valeur particulière mais simplement de tester un « oui ou non ». Pour cela l'usage consiste à tester l'existence ou non d'une macro et non sa valeur. Les primitives `#ifdef MACRO` sont alors utilisées.

`#ifdef MACRO` est validée par le préprocesseur si la macro `MACRO` est définie, quelque soit sa valeur (et en particulier, même si elle n'a aucune valeur, ce que l'on obtiendrait avec un `#define MACRO` sans rien de plus...).

Exercice 8 (Tester l'existence plutôt que la valeur)
Travail à réaliser en TD : Comment faire en sorte que l'affichage n'ait pas lieu lorsque la macro `NO_LOG` est définie ?

Travail à réaliser en TP : Implémentez ceci, et vérifiez que votre implémentation fonctionne en faisant :

```
bash$ gcc compcond.c -DNO_LOG
```

Notez qu'il n'est plus utile de donner une valeur à la macro `NO_LOG` puisque seule son existence est testée.

Comparez le résultat avec celui de la compilation obtenue par :

```
bash$ gcc compcond.c
```

□

Considérez maintenant le programme suivant (`compchk.c`) :

```
#ifndef SIZE
#error "définissez SIZE avec l'option -D SIZE=n"
#define SIZE 0
#endif
#if SIZE & (SIZE-1)
#warning "SIZE devrait être une puissance de 2."
#endif

int main(void)
{
    return SIZE;
}
```

Exercice 9 (Erreur de compilation programmée)
Travail à réaliser en TD : Que va-t'il see passe-t'il si vous ne définissez pas `SIZE` sur la ligne de compilation ? □

Travail à réaliser en TP : Testez tout cela en tentant de compiler votre programme.

Exercice 10 (Avertissement de compilation programmé)
Travail à réaliser en TD : Que va t'il se passer si vous définissez une valeur de `SIZE` qui n'est pas une puissance de 2 ? Pourquoi ?

Bon usage du préprocesseur ***...indispensable...***

Le préprocesseur permet d'écrire de transformer le code source, et, dans une certaine mesure, de faire évoluer la syntaxe du langage C, de façon assez radicale. Si cet usage a connu son heure de gloire dans les années 1990, il est aujourd'hui largement déprécié. En effet, l'usage intensif du préprocesseur, pour créer de nouveaux « pseudo mots clefs », pose deux problèmes principaux :

— le code source ainsi traité devient difficile à lire par un tier qui ne connaît pas les macros mises en place ;

— le code source devient difficile à débugger. Considérez

```
#define MAX 3;
int i;
if (i < MAX) {
...
```

Pour mettre en évidence les parties du codes qui seront transformés par le préprocesseur, dans un source C, un certain nombre d'usages se sont progressivement imposés :

1. les symboles du préprocesseur sont écrits en majuscule : `#define MA_MACRO 17`
2. les expressions évaluées sont sur-parenthésées : `#define MA_MACRO (3+i*2)` plutôt que `3+i*2`...
3. les macros paramétrées sont dépréciées au profit de fonctions `inline` du type `static inline int max(int i,int j) { return (i<j)?i:j; }`
4. les notices précisent toujours les fonctions susceptibles d'être implémentées avec des macros : voir par exemple `$ man putc`
5. les `#include` portent toujours sur des fichiers de prototypes `.h` et pas sur des fichiers `.c` ;
6. les fichiers de prototypes ne devraient jamais contenir d'implémentation mais seulement des déclarations (à l'exception des fonctions `inline`) ;
7. il est conseillé d'utiliser des *include guard* : consulter `https://fr.wikipedia.org/wiki/Include_guard`.

Travail à réaliser en TP :
Compilez le programme en définissant une valeur pour la macro `SIZE` qui n'est pas une puissance de 2. Que ce passe-t-il ? Pourquoi ? □

Avant de poursuivre avec l'étude des fonctionnalités du préprocesseur, assurez-vous de bien comprendre les quatre premières recommandations de l'encart *Bon usage du préprocesseur*.

1.3 Inclure des fichiers sources, les uns dans les autres

Le préprocesseur du langage C permet d'inclure un fichier dans un autre. C'est le sens de la directive `#include "`*`un_fichier`*`"`.

Lorsque le préprocesseur rencontre cette commande il effectue l'équivalent d'un « copié-collé » de *un_fichier* en lieu est place du `#include "un_fichier"`. Il est ainsi possible d'inclure un autre fichier source dans un fichier source C.

En utilisant les `#include` il devient possible de mettre des fonctions dans les fichiers que l'on réutilise d'un programme à l'autre. (Cependant le langage C nous invite à ne pas faire usage du préprocesseur en ce sens, mais à plutôt privilégier la création de *fichiers objets* que nous expérimenterons dans un exercice ultérieur.)

L'inclusion de fichiers est trés utilisées pour éviter d'avoir à redéclarer les fonctions externes que nous avons l'intention d'utiliser.

Précédemment, nous avons de nombreuses fois déclaré la fonction `int putchar(int c);` qui est implémentée dans une librairie de fonctions standard du langage C. Cette fonction, comme toutes celles de la librarie standard sont très fréquemment utilisées. Aussi il peut être fastidieux, de déclarer chacune d'elles au début de chaque nouveau programme. C'est pourquoi

un fichier de déclaration de ces fonctions a été standardisé. Notez bien que ce fichier ne contient pas l'implémentation mais seulement la *déclaration* des fonctions (c'est-à-dire la ligne *e.g.* `int putchar(c) ;` sans le corps de la fonction).

Exercice 11 (Les fichiers `.h` - ou fichiers de prototypes)
Travail à réaliser en TP : Trouver le nom du fichier dans lequel la fonction `putchar()` est déclarée en interrogeant le manuel. □

Notez que dans le manuel le fichier a pour extension `.h` et pas `.c`. Un fichier `.h` est un source en langage C au même titre qu'un fichier `.c`. Cependant l'extension `.h` a pour vocation d'indiquer que le fichier contient des déclarations, mais pas d'implementation. On parle de *fichiers de prototypes*. Ils sont donc destinés à être inclus dans d'autres fichiers C mais pas à être compilés pour eux-mêmes.

Vous pouvez observer que votre répertoire courant (celui de vos fichiers sources) ne contient pas le fichier `.h` mentionné par le manuel. Vous pouvez aussi remarquer que dans le manuel le nom du fichier à inclure n'est pas donné entre guillemets `" ... "`, mais pas entre chevrons `< ... >`.

Lorsqu'un fichier est donné entre guillemets, c'est que le préprocesseur doit le trouver dans le repertoire courant (ou qu'il doit éventuellement être trouvé dans des repertoires annexes définis *explicitement* au compilateur par l'option `-I dir`).

Lorsqu'un fichier à inclure est donné entre chevrons, c'est qu'il doit être trouvé dans des répertoires annexes définis *implicitement* par le compilateur. Pour connaitre la liste de ces répertoires définit par `gcc` vous pouvez faire :

```
bash$ echo | gcc -E -Wp,-v -
```

Exercice 12 (Les `#include` par défaut)
Modifier le programme ci-dessous pour qu'il affiche le résultat de `f(3)` sur l'écran avant de sortir. Pour cela utilisez la fonction `putchar()`, mais au lieu de la déclarer en début de fichier, demandez au préprocesseur d'inclure le fichier indiqué par le manuel.

```
int f(int x)
{
    return 2*x;
}

int main()
{
    return f(3);
}
```

Travail à réaliser en TD : Préparez votre modification, en expliquant quoi modifier et où.

Travail à réaliser en TP : Réalisez la modification. Vérifiez que votre nouveau programme fonctionne.

Observez le fichier source produit par le préprocesseur. Combien de ligne contient le fichier produit par le préprocesseur ? Comment est déclarée la fonction `putchar()` ? □

Lorsque les programmes deviennent plus complexes, il peut être tentant de mettre des directives `#include` dans des fichiers `.h`. Dans ce cas, par défaut, le préprocesseur va inclure, récursivement les fichiers référencés. La norme du langage C précise qu'un compilateur doit pouvoir gérer jusqu'à 16 niveaux d'inclusion (d'un fichier, dans un autre, dans un autre...). `gcc` en gère jusqu'à 200.

Considérez le petit exemple suivant composé de trois fichiers :

Un fichier `abs.h` :

```
static inline int abs(int x) { return (x<0)?-x:x; }
```

Un fichier `minmax.h` :

Fonctions `inline` ***...en savoir plus...***

Le mot clé `inline` utilisé lors de la définition d'une fonction indique au compilateur d'essayer de mettre en place une optimisation. Cette optimisation consite à *étendre* le code de la fonction. C'est-à-dire à remplacer le nom de la fonction par son code, évitant ainsi le coût d'un appel de fonction lors de l'exécution.

```
#include "abs.h"

static inline int min(int x,int y) { return (x+y-abs(x-y))/2; }
static inline int max(int x,int y) { return (x+y+abs(x-y))/2; }
```

Et enfin un dernier fichier `guard.c` :

```
#include "minmax.h"
#include "abs.h"

int putchar(int c);

int main() {
    putchar('0' + min(3, 4));
    putchar('0' + max(3, 4));
    putchar('0' + abs(-2));
    return 0;
}
```

Dans cet exemple, on imagine que les deux fichiers `.h` ont été conçus par Alice. Le fichier `abs.h` contient la définition d'une fonction `abs()`.

Veuillez noter qu'en principe on ne doit pas mettre de *définition* de fonctions dans un fichier `.h`, mais seulement des déclarations. Toutefois, la fonction `abs()` dans le présent exercice constitue une exception, car c'est une fonction en *static inline* (consultez l'encart pour de plus amples informations sur *inline*, quant au mot clef *static* il sera détaillé dans la section sur la compilation modulaire), on a alors le droit de la mettre dans le fichier `.h`.

Bob désire utiliser les fonctions proposées par Alice a écrit le fichier `.c` indépendamment.

La compilation de `guard.c` échoue, le compilateur rapporte que la fonction `abs()` est définie plusieurs fois.

Exercice 13 (Règle de la simple définition)
Pourquoi Bob transgresse la règle de simple définition ? □

Travail à réaliser en TD : En lisant le code, dites ce que va produire le préprocesseur pour le fichier `guard.c`

Travail à réaliser en TP : Observez le résultat du traitement du préprocesseur sur le fichier `guard.c`.

Bob pourrait corriger son fichier `guard.c`. Cependant sa première version est raisonnable et ne devrait pas poser de problème de compilation.

C'est à Alice de fournir, en tant que le développeuse expérimentée, de livrer des fichiers sources qui puissent toujours être utilisés.

Exercice 14 (*#include guard*)
Travail à réaliser en TD : Proposez une modification des fichiers d'Alice pour que Bob ne rencontre plus de problème de compilation, sans toucher à son programme.

Notez aussi que Alice ne souhaite pas fusionner les deux fichiers `.h` et qu'elle souhaite bien inclure `abs.h` dans `minmax.h` (pour que `min()` et `max()` profitent des prochaines évolutions de `abs()`). □

Travail à réaliser en TP :
Implémentez et testez votre solution

Semaine 4 : Les outils de compilation (2/2)

2 Compilation modulaire

Avec la notion de préprocesseur, la compilation modulaire est un mécanisme fondamental du langage C. Le but de la compilation modulaire et de concevoir un programme comme l'assemblage de plusieurs « modules ».

Chaque module est le fruit de la compilation séparée d'un fichier source (fichier `.c`) différent. Les modules sont autant de *fichiers objets* (fichiers `.o`) différents. Pour produire un fichier objet à partir d'un fichier source il faut utiliser l'option `-c` du compilateur `gcc`.

L'assemblage des différents modules est appelé *linkage*, *liaison*, ou encore *édition de liens*. Lors de cette opération de liaison, les différents fichiers objets sont réunis pour produire un unique fichier exécutable. Pour que cette opération aboutisse il faut qu'un et un seul des fichiers liés contienne la fonction `main()` qui sera le point d'entrée de l'exécution du programme produit.

Pour ce travail sur la compilation modulaire, vous travaillerez dans le dossier `compil_modulaire` de votre dépôt.

2.1 les fichiers objets

Vous avez réalisé un fichier `numbers.c` qui compile toute une série de fonctions d'affichage des entiers sous leurs formes décimale, hexadécimale ou binaire. Cet ensemble de fonctions est utile dans de nombreux programmes. Selon les canons de la compilation modulaire il est donc pertinent de compiler une fois cet ensemble de fonctions dans un fichier `.o` puis de réutiliser ces fonctions dans tous les programmes que vous produirez, sans les recompiler.

Vous allez travailler dans le répertoire `module/` de votre dépôt.

Exercice 15 (Un module `put_numbers.c`)
On projette de créer un fichier `put_numbers.c` qui propose une série de fonctions permettant d'afficher des nombres sur la sortie standard. Ce fichier ne contiendra pas la fonction `main()`, puis de produire un module `put_numbers.o` qui contiendra l'ensemble des fonctions que vous avez réalisées.

Travail à réaliser en TD : Expliquez comment réaliser cette ligne de compilation ?

Travail à réaliser en TP :

Créez le fichier `put_numbers.c` comme demandé, puis testez cette ligne de compilation. A ce stade la, est-ce que vous pouvez exécuter quelque chose ? Pourquoi ?

Parmi les fonctions de `put_number.c`, certaines, comme `put_dec()` sont publiques, c'est-à-dire qu'elles vont être appelées depuis d'autres fichiers `.c`, et d'autres, au contraire, sont privées, et seront appelées uniquement depuis `put_numbers.c`.

Une bonne pratique consiste à préfixer la déclaration d'une fonction privée par le mot clef `static`, grâce à ce mot clef, ces fonctions ne seront visibles que depuis le module courant. Par exemple :

```
static int mafonctionprivee(int) {
        // ...
}
```

Si ma fonction privée est au préalable déclarée avec un prototype, alors le mot clef `static` doit être utilisé au niveau du prototype.

Exercice 16 (Un module `put_numbers.c`)
Travail à réaliser en TD : Parmi les fonctions de `put_number.c`, identifiez celles qui devraient être publiques, et celles qui devraient être privées.

Travail à réaliser en TP : Préfixez la déclaration de vos fonctions privées par `static`

Exercice 17 (Utiliser un module)
Produisez maintenant un fichier `numbers-test.c` qui implémente la fonction `main()` de test de l'ensemble des fonctions de votre module `put_numbers.o` (vous pouvez largement vous inspirer de ce que vous avez fait lors du thème 1)

Ce fichier ne doit pas contenir les fonctions testées, `putdec()`, `puthex()`...).

Produisez un module `numbers-test.o` à partir de votre programme.

Travail à réaliser en TD : Comment compiler le programme ?

Travail à réaliser en TP :

Faites-le. La compilation provoque des avertissements (*warning*). Pourquoi ? Que faut-il ajouter pour éviter ces *warning* de compilation ? □

Exercice 18 (Fichier de prototype d'un module (ou fichier d'entête))
Les déclarations nécessaires pourraient être placées dans le fichier `numbers-test.c` lui-même. Cependant ces mêmes modifications seraient à reporter dans chaque fichier source qui utiliserait des fonctions du module `put_number.c`.

Travail à réaliser en TD : Proposez une solution générique qui repose sur l'usage du préprocesseur et des fichiers de prototypes vus précédemment.

Est-ce que les fonctions privées doivent être déclarées dans le fichier d'entête ? Pourquoi ?
Travail à réaliser en TP :

Implémentez votre solution proposée, et compilez la.

Exercice 19 (Édition de liens)
Travail à réaliser en TD :

Comment lier ce module `numbers-test.o` maintenant compilé sans *warning* avec le module `put_numbers.o` pour produire un programme executable `numbers-test` ?

Travail à réaliser en TP : Faites-le, et vérifiez que cela fonctionne.

2.2 Compilation globale du projet

Finalement, votre programme `numbers-test` est composé de 3 fichiers sources et 3 fichiers contenant du code machine :

`put_numbers.c` contient le code source des fonctions d'affichage des nombres ;

`put_numbers.h` contient les prototypes des fonctions implémentées par `put_numbers.c` ;

`numbers-test.c` contient le code source du programme de test – la fonction `main()` ;

`put_numbers.o` contient le code machine des fonctions d'affichage des nombres compilé avec `gcc` ;

`numbers-test.o` contient le code machine du programme principal de test ;

`numbers-test` contient le programme exécutable qui teste les fonctions d'affichage des nombres.

La compilation modulaire nous a permis *(i)* de structurer notre programme en différents fichiers servant des objectifs différents et *(ii)* de produire une librairie (ou module) de code `put_numbers.o` que l'on pourra réutiliser dans d'autres programmes grace au fichier `put_numbers.h` (en liant le `.o` avec les programmes qui l'utilisent).

Cependant lorsque vous modifiez le fichier `put_numbers.c` la recompilation de l'exécutable `numbers-test` devient fastidieuse...

Exercice 20 (Un premier script de compilation)
Travail à réaliser en TD : Proposez un script `bash` que vous appellerez `compile_test_numbers.sh` qui exécute, en cascade l'ensemble des commandes de compilation pour produire, à partir des fichiers sources, les fichiers `.o` et le fichier exécutable.

Travail à réaliser en TP : Implémentez ce script et vérifiez son bon fonctionnement.

Ce script effectue systématiquement toutes les phases de compilation pour tous les fichiers, ce qui fait perdre du temps inutilement.

Vous allez voir maintenant la compilation modulaire avec `make`, qui permet d'éviter ces inconvénients en ne recompilant que ce qui est nécessaire.

2.3 Compilation modulaire avec `make`

Pour pouvoir recompiler uniquement ce qui est nécessaire, il suffit de savoir que le fichier `numbers-test` « dépend » des fichiers `numbers-test.o` et `put_numbers.o`. Si l'un de ces deux fichiers a changé depuis la dernière production de `numbers-test`, il faut simplement exécuter la commande `gcc put_numbers.o numbers-test.o -o numbers-test`.

Dans un fichier `Makefile` on exprime cela par une *règle* comme suit :

```
numbers-test: numbers-test.o put_numbers.o
        gcc numbers-test.o put_numbers.o -o numbers-test
```

De la même manière, chaque fichier `.o` dépend de son fichier `.c`, ce qu'on peut exprimer comme suit :

```
numbers-test.o: numbers-test.c
        gcc -c numbers-test.c

put_numbers.o: put_numbers.c
        gcc -c put_numbers.c
```

Le fichier `Makefile` peut contenir un série de *règles*. Chaque règle débute par une *ligne de dépendances* qui pour une `cible` donnée (`numbers-test`), liste ses *prérequis* (`numbers-test.o put_numbers.o`). Cette ligne de dépendances est suivie de lignes de commandes qui seront exécutées si un des prérequis est plus récent que la cible. Ces lignes de commandes doivent être précédées d'une tabulation en début de ligne.

Pour déclencher la recompilation conditionnelle d'un fichier, il suffit ensuite de lancer la commande `make`. Cette commande va essayer de reconstruire, si nécessaire, la cible de la première règle trouvée dans le fichier `Makefile`. Si les prérequis de cette règle sont eux-mêmes l'objet d'autres règles de compilation, ils seront préalablement reconstruits si nécessaire, et ce récursivement.

Ainsi un fichier `Makefile` permet d'exprimer beaucoup plus simplement qu'un fichier de script ce qui doit être fait pour recompiler un programme.

Exercice 21 (Les `Makefile`)
Vous consulterez éventuellement le manuel en ligne `man` pour la commande `make`.

Travail à réaliser en TD : Proposez un fichier `Makefile` qui réalise la même tâche que votre fichier de script `compile_test_numbers.sh`.

Travail à réaliser en TP : Testez votre `Makefile` de la même manière que vous aviez testé votre script de compilation. □

Notez que dans ce `Makefile`, contrairement au script shell, l'ordre dans lequel les règles sont données n'a pas d'importance. Tout au plus, vous pouvez notez que lorsque vous lancez la commande `make` sans paramètre, c'est la première règle trouvée dans le fichier qui est considérée.

L'usage (ou une bonne pratique) consiste à s'assurer que la première règle, la règle évaluée par défaut, assure la compilation de l'ensemble du projet. Pour assurer cela, on place généralement, en première règle du `Makefile` règle blanche (sans commande associée) qui s'écrit :

```
all: ...
```

Les `...` sont remplacés par la liste des règles qui doivent être évaluées lors du lancement du `make`. Donc votre cas, `make` doit produire une chose : `numbers-test`.

Il est aussi d'usage qu'un `Makefile` propose une règle `clean` qui supprime tous les fichiers compilés (`.o` et exécutables). Cette règle ne dépendant de rien, et elle consiste en l'exécution d'une commande `rm` avec les paramètres appropriés.

Pour notre exemple, nous pourrions écrire dans le fichier `Makefile` :

```
clean:
        rm -f numbers-test .o put_numbers.o
        rm -f numbers-test
```

et utiliser la commande

```
bash$ make clean
```

pour supprimer les fichiers non indispensables.

Enfin, une bonne pratique consiste à définir un certains nombre de variables au début du `Makefile` qui pourront être changées, au besoin et qui définissent :

`CC` pour le compilateur C à utiliser – `gcc` pour nous ;

`CFLAGS` pour les options du compilateur ; on utilisera par exemple

```
CFLAGS = -Wall -Werror -ansi -pedantic
```

pour demander au compilateur
- d'afficher tous les avertissments possibles – `-Wall` ;
- de les considérer comme des erreurs – `-Werror` ;
- de se conformer à la norme ANSI du langage C – `-ansi`, et ce de manière stricte – `-pedantic`.

Il est ensuite possible d'utiliser ces variables dans le `Makefile` lors de la description des commandes shell à lancer en écrivant `$(VAR)` là où elles doivent être utilisées. On écrira par exemple (extrait d'un `Makefile`) :

```
CC     = gcc
CFLAGS = -Wall -Werror -ansi -pedantic

numbers-test: numbers-test.o put_numbers.o
        $(CC) $(CFLAGS) numbers-test.o put_numbers.o -o numbers-test
put_numbers.o: put_numbers.c
        $(CC) $(CFLAGS) -c put_numbers.c -o put_numbers.o
```

De la même façon il est possible de faire référence au fichier cible de la règle courante avec `$@`, et à l'ensemble de ses fichiers sources avec `$^`.

Exercice 22 (Règle pour fabriquer la cible)
Considérez la règle suivante :

```
numbers-test: numbers-test.o put_numbers.o
        $(CC) $(CFLAGS) $^ -o $@
```

Travail à réaliser en TD : Quel est le sens de cette règle ?

Quelle sera la commande exécutée si cette règle est considérée ?

Travail à réaliser en TP : Modifiez votre `Makefile` pour qu'il utilise cette règle. Ré-écrivez les autres règles sur le même principe. Proposez aussi une règle par défaut `all` et une règle de nettoyage `clean`. □

Bravo ! Vous savez créer et organiser un projet en langage C !

2.4 Méthodes et outils de résolution des problèmes

Exercice 23 (Les erreurs classiques de compilation et/ou d'éditions de liens)
Dans cet exercice, vous avez a examiner plusieurs scenarios indépendants, dans lesquels on vous donne le contenu d'un Makefile, et d'un ou plusieurs fichiers source.

Pour chaque scénario, vous devez réaliser les actions suivantes :

— (TD) Donnez la liste des unités de compilation, et pour chaque unité de compilation, donnez la liste des fichiers `.h` inclus.
— (TD) Pour chaque fonction, donnez le/lies unité(s) de compilation dans laquelle elle est définie, et dans laquelle elle est déclarée.
— (TD) Vérifiez que chaque fonction utilisée dans une unité de compilation est bien déclarée au préalable, dans la même unité de compilation.
— (TD) Pour chaque fichier exécutable créé par le Makefile, en examinant la commande de création de cet exécutable, vérifiez que chaque fonction utilisée est définie exactement une fois, et qu'il existe exactement une définition de `main`.
— (TP) Si vous avez constaté une anomalie en faisant ces vérifications, réalisez la correction, et testez-votre correction en TP.

2.4.1 Scenario 1

Fichier `foo.h`

```
void foo();
```

Fichier `foo.c`

```
void foo() {
}
```

Fichier `main.c`

```
#include "foo.h"
int main() {
        foo();
}
```

Fichier `Makefile`

```
all: main

main: main.o
        cc -o main main.o

main.o: main.c
        cc -c main.c

foo.o: foo.c
        cc -c foo.c
```

2.4.2 Scenario 2

Fichier `foo.h`

```
void foo() {
}
```

Fichier `foo.c`

```
#include "foo.h"
```

Fichier `main.c`

```
#include "foo.h"
int main() {
        foo();
}
```

Fichier `Makefile`

```
all: main

main: main.o
	cc -o main main.o foo.o

main.o: main.c
	cc -c main.c

foo.o: foo.c
	cc -c foo.c
```

2.4.3 Scenario 3

Fichier `foo.h`

```
void foo();
```

Fichier `foo.c`

```
void foo() {
}
```

Fichier `main.c`

```
int main() {
	foo();
}
```

Fichier `Makefile`

```
all: main

main: main.o
	cc -o main main.o foo.o

main.o: main.c
	cc -c main.c

foo.o: foo.c
	cc -c foo.c
```

Exercice 24 (Utilisation de gdb)
Travail à réaliser en TP :

Rappel : pour utiliser gdb, vous devez passer les options `O0 -g` lors de la compilation et de l'édition de lien de votre programme.

Une fois ceci fait, vous pouvez lancer votre programme avec gdb comme ceci : `gdb ./monProgramme` (au lieu de `./monProgramme` pour lancer le programme normalement).

Rappel des commandes disponibles avec gdb (vous pouvez vous référer au cours, où à la documentation de gdb pour en savoir plus) :

— `break numerodeligne` Ajouter un breakpoint à une ligne donnée
— `run` Lancer le programme
— `print expr` Afficher la valeur d'une expression du programme
— `cont` Continuer le programme après avoir rencontré un breakpoint
— `next` Exécuter une ligne du programme, en traitant les appels de fonctions comme une seule ligne

— `step` Exécuter une ligne du programme, en rentrant dans les appels de fonctions
— `back` Afficher la liste des fonctions actives
— `frame numero` Se positionner sur la fonction active correspondant au numéro

Vous allez expérimenter gdb sur le programme suivant :

```
void foo(int s) {
        printf("foo() appele avec: %d\n", s);
}
void bar(int s) {
        printf("foo() appele avec: %d\n", s);
}
int main() {
        int tab[10] = {1,5,8,3,1,9,10,25,32,42};
        int i;
        int sum = 0;
        for (i = 0; i <= 10 ; i++) {
                sum = sum + tab[i];
        }
        foo(sum);
        bar(sum);
}
```

Compilez le avec les options nécessaires pour gdb. Lancez ensuite le programme sous gdb.

Ce programme est sensé faire la somme des valeurs d'un tableau.

Examinez la valeur de `i` après la première boucle (pour cela, placez un breakpoint, puis utiliez ensuite la commande `print`). La valeur de `i` en fin de boucle devrait vous donner une indice sur la nature de l'erreur du programme. Si vous ne trouvez pas, alors placez un breakpoint sur l'affectation qui est à l'interieur de la boucle, et relancez le programme en examinant la valeur de i à chaque exécution de cette affectation.

Une fois ceci fait, experimentez la commande `next` pour exécuter l'appel à `foo()`, puis testez la commande `step` sur l'appel à `bar()`. Une fois dans la fonction `bar()`, experimentez avec les commandes `back` et `frame`.

Puis utilisez la commande `cont` pour continuer le programme jusqu'à la fin.

Maintenant, réalisez la correction du programme, et testez le programme corrigé.

Exercice 25 (Utilisation d'ASan)
Travail à réaliser en TP :

Avec l'exercice précédent, vous avez constaté que les accès hors des tableaux ne génèrent pas nécessairement d'erreurs à l'exécutions en C, ce qui peut rendre la découverte de certains bugs assez difficile.

L'outil `ASan` (pour Adress Sanitizer) permet de remédier partiellement à ce problème en rajoutant un certain nombre de contrôles à l'exécution, notamment sur l'accès aux tableaux.

Pour utiliser ASan, il faut utiliser des options spécifiques lors de la compilation et de l'édition de liens : `-O0 -g -fsanitize=address`.

Ensuite, vous pouvez lancer votre programme normalement, et en cas d'accès incorrects à des tableaux ou autres erreurs du même type, ASan arretera le programme et vous indiquera la ligne de code source fautive.

Reprenez le programme vu dans l'exercice précédent (dans sa version non corrigée), compilez le avec ASan et lancez le. ASan va stopper le programme en vous affichant beaucoup d'informations, le plus important se trouve dans le premier paragraphe. Vous devriez voir le nom de fichier et le numéro de ligne qui provoque l'accès fautif au tableau.

Exercice 26 (Utilisation conjointe d'ASan et de gdb)
Travail à réaliser en TP :

Vous pouvez utiliser de manière conjointe ASan et gdb en utilisant la commande suivante : `ASAN_OPTIONS="abort_on_error=1" gdb ./monProgramme` (où monProgramme est le nom de votre fichier executable). Reprenez le programme erroné de l'exercice précédent pour tester cette fonctionnalité.

Compilez le programme avec `-O0 -g -fsanitize=address` et lancez le avec `ASAN_OPTIONS="abort_on_error=1" gdb ./monProgramme`. Démarrez votre programme depuis gdb avec `run`, et constatez qu'il est stoppé par ASan à cause du debordement de tableau.

Une fois le programme stoppé par ASan, utilisez la commande `back` de gdb pour voir l'ensemble des fonctions actives. Vous verrez une liste de fonctions peu intéressantes qui font partie d'ASan lui-même, mais vous devriez aussi repérer la fonction `main()` qui appartient à notre code. Avec la commande `frame` positionnez vous sur la fonction main, puis avec `print` affichez la valeur courante de la variable `i`.

Quelle est la valeur de `i` que vous observez ? Est-ce que ce `i` est un index valide de `tab` ?

Comme vous pouvez le constater, l'utilisation d'ASan et de gdb vous permet de voir exactement où votre programme réalise un accès incorrect à un tableau, et d'afficher la valeur des variables au moment où cet acces incorrect se produit, vous aidant à diagnostiquer le problème.