ENSEIRB 2006-2007 - Première année
PG101 - Programmation en C

TD 1 : Premiers programmes & compilation (commentaires)

1 Compilation

Exercice 1:

Penser à activer la coloration syntaxique

Exercice 2:

1. on ne cherche pas à éliminer les avertissements du compilateur dans cet exercice (c'est le but de l'exercice 3)
2. on n'a pas recompilé donc on ne voit aucun changement. Leçon : quand on modifie le source, il faut recompiler

Exercice 3:

1. `C-x '` permet de se déplacer d'avertissement/erreur en avertissement/erreur avec mise en évidence syntaxique dans le buffer `emacs`
2. Un nouvel avertissement apparaît lors de la compilation : `prog1.c :3 : warning : control reaches end of non-void function`. Rien de nouveau côté exécution

   La décaration `int main()` signifie que le *programme principal* `main` produit un résultat de type `int` qui permet par exemple d'indiquer une erreur de déroulement du programme. Ceci est par exemple utilisé par `emacs` : son buffer de compilation affiche ou non le message `Compilation finished at ...` suivant que la compilation se soit déroulée sans erreur.

   Le message signifie donc que le programme principal est censé retourner un résultat de type `int` mais que ceci n'est pas le cas.

   Remarque : le standard ISO C fixe ce type de retour pour `main` (il en sera donc toujours ainsi).
3. L'ajout de l'instruction `return 0 ;` provoque justement la production d'une valeur de retour par le programme principal `main`. Donc l'avertissement disparaît (le code `0` signifie que tout s'est bien passé, c'est la tradition des shells UNIX)

   Donc au niveau de la compilation, l'avertissement disparaît, et ceci n'a aucune influence sur l'exécution (en tout cas, rien que nous puissions constater)

4. L'ajout de la ligne `#include <stdio.h>` permet au compilateur `gcc` de savoir comment est définie `printf`, et de vérifier que nous l'utilisons correctement. Plus précisément, `stdio.h` est un fichier qui contient les déclarations d'un certain nombre de fonctionalités d'entrée/sortie.

   L'avertissement restant : `prog1.c :2 : warning : implicit declaration of function 'printf'` disparaît à son tour. Il signifie que `printf` n'est pas déclarée explicitement, donc que le compilateur `gcc` ne peut pas vérifier que l'utilisation qui en est faite est bonne.

   Pas de changement visible côté exécution.

Exercice 4:

Ce programme est bourré d'erreurs fréquentes (oubli de guillemets, de point-virgule, etc) :

```
prog2.c:1:18: error: stdo.h: No such file or directory
prog2.c: In function 'man':
prog2.c:4: warning: implicit declaration of function 'printf'
prog2.c:4: warning: incompatible implicit declaration of built-in function 'printf'
prog2.c:6: error: parse error before 'return'
prog2.c:7: warning: control reaches end of non-void function
```

Il s'agit de corriger ces erreurs en les prenants à partir de la première : le fichier `stdo.h` ne peut pas être trouvé. Il y a une erreur de frappe, il s'agit de `stdio.h`.

On corrige, on sauvegarde, puis on compile à nouveau, et on obtient les avertissements et erreurs suivants :

```
prog2.c: In function 'man':
prog2.c:6: error: parse error before 'return'
prog2.c:7: warning: control reaches end of non-void function
```

Il y a une erreur de syntaxe avant `return`. Il faut bien souvent regarder la ligne précédente. Ici il manque le `;` à la fin de cette ligne

On corrige, on sauvegarde, puis on compile à nouveau :

```
/usr/bin/ld: Undefined symbols:
_main
collect2: ld returned 1 exit status
```

Le programme principal `main` manque. On a là-aussi une faute de frappe : nous avons tapé `man`.

Après cette dernière correction, tout va bien.

Notons que le dernier message d'erreur affiche la valeur de retour du programme principal de l'outil `ld`, qui vaut ici `1` signifiant une erreur. C'est une nouvelle illustration de l'utilisation de la valeur de retour des programmes.

2 Variables et types

Exercice 5:

La structure des paramètres de `printf` : chaîne de *formats* suivie d'autant d'expressions qu'il y a des formats à substituer. Un format précise comment une valeur doit être interpétée pour son affichage. Par exemple, `%d` signifie que la valeur doit être interpétée comme un entier, `%c` signifie qu'elle doit être interpétée comme un caractère, `%f` signifie que la valeur doit être interprétée comme un nombre flottant, et ainsi de suite.

Voici le résultat visible de l'exécution de ce programme :

```
division 2/3=0
division 2.0/3.0=0.666667
```

Dans le premier cas, `2/3` est la division de la constante `2` *entière* par la constante `3` *entière*, le résultat étant le quotient de la division euclidienne.

Dans le second cas, `2.0/3.0` est la division de la constante `2.0` *flottante* par la constante `3.0` *flottante*, le résultat étant le nombre flottant résultant de la division.

Nous constatons donc d'une part que les constantes ont un type : `int` dans le premier cas et `double` dans le second. Ensuite, les opérateurs se comportent de façons éventuellement différentes en fonction du type des opérandes.

Exercice 6:

La compilation de ce programme provoque l'affichage d'un message d'avertissement : `prog3.c :4 : warning : format '%d' expects type 'int', but argument 2 has type 'double'`

Nous avons vu à l'exercice précédent que le résultat de l'expression `2.0/3.0` a le type `double`. Or, ici, nous essayons d'interpréter la valeur résultante comme un entier, ce qui est incompatible (d'où l'avertissement).

On obtient d'ailleurs à l'exécution un résultat qui n'a aucun sens : `division 2.0/3.0=1071994197` puisque la valeur est interprétée d'une façon erronée.

Exercice 7:

Ici deux formats : `%d` pour les entiers signés et `%u` pour les entiers non signés. Pas d'avertissement à la compilation car ces formats sont compatibles avec le type de `x`.

À l'exécution, deux valeurs différentes suivant qu'on interprète la suite des bits de `x` comme signée ou non.

Exercice 8:

On remarque là aussi la compatibilité des formats et du type de `c`.

À l'exécution, on obtient soit la lettre `'A'` pour le format `%c`, soit la valeur `65` pour le format `%d`. En fait, `c` contient la valeur `65` qui peut être affichée comme telle ou interprétée comme un caractère via une conversion par la table ASCII

Exercice 9:

Ces valeurs sont le nombre d'octets (8 bits) utilisés pour stocker une valeur de chacun des types.

On remarque que les types signés et non signés occupent le même espace mémoire : c'est le premier bit qui sert à coder le signe dans le cas signé et qui est exploité pour coder plus de valeurs dans le cas non signé.

Les types entiers, à savoir `char`, `short`, `int`, `long` et leurs déclinaisons non signées, sont de simples séquences de bits. Ainsi, une valeur de type `char` peut coder toute valeur expressible sur 8 bits, soit -128 à 127 en signé et 0 à 255 pour `unsigned char`. De même, `int` peut coder les valeurs de -2^{31} à $2^{31}-1$, et `unsigned int` peut coder les valeurs de 0 à $2^{32}-1$ sur les architectures 32 bits usuelles.

Les types flottants, à savoir `float` et `double`, sont structurés : une partie des bits sert à stocker la mantisse, et une autre sert à stocker l'exposant (voir le polycopié de cours pour les détails). Le type `double` permet de représenter plus de valeurs que le type `float` (il est donc plus précis).

Remarque : reprenez les observations des exercices précédents à la lumière de ceci.

Exercice 10:

La compilation provoque des avertissements lorsque les types des formats et des variables sont incompatibles :

```
prog7.c:7: warning: format '%f' expects type 'double', but argument 3 has type 'int'
prog7.c:8: warning: format '%d' expects type 'int', but argument 3 has type 'double'
```

À l'exécution, on constate bien des incohérences liées à cette incompatibilité. Ceci est du au fait qu'un nombre flottant est stockée comme une structure de bits, alors qu'un nombre entier est une simple séquence de bits.

Exercice 11:

La façon correcte de passer d'un type à un autre est d'en forcer le type. Ainsi, l'expression `(float)c` correspond à la valeur de `c` convertie suivant la représentation binaire des flottants de type `float`. Cette opération transforme donc la séquence de bits codant la valeur de `c` en une structure plus complexe de bits utilisée pour coder les nombre flottants.

Nous voyons que le forçage de type (ou *casting*) permet de convertir des types incompatibles, en dégradant parfois la valeur (par exemple, le passage de `float` à `int` fait bien entendu perdre la partie fractionnaire).

Remarque : le résultat de l'expression `(float)2/(float)3` est identique à celui de `2.0/3.0` du fait du changement de type des constantes.

Exercice 12:

Lors de l'exécution, le programme affiche les valeurs :

```
c=-128
n=4294967295
```

On constate donc qu'il n'y a pas de vérification des débordements de type en C. Les variables sont valuées modulo les valeurs minimales et maximales qu'il peuvent représenter. Ainsi, la variable `c` de type `char` peut représenter des valeurs de `-128` à `127`, et en arithmétique signée modulo `127`, les valeurs `128` et `-128` sont représentées de la même façon. Plus exactement, seule `-128` est représentée : `128` est hors du domaine de `c`.

Exercice 13:

Les problèmes liés aux débordements produit une boucle infinie. En effet, lorsque la valeur de `c` atteint `0`, l'instruction `c--` qui devrait donner la valeur `-1` à `c` lui donne la valeur `255` (arithmétique modulo `255`). Ainsi, la boucle ne termine jamais puisque la valeur de `c` est toujours positive ou nulle.

Notons que le compilateur signale le problème d'un avertissement :

```
prog11.c: In function 'main':
prog11.c:7: warning: comparison is always true due to limited range of
data type
```