TP0 – Découverte de l'environnement – Correction

Projet de programmation M1

23 Septembre 2014

1 Préambule

Tous les TD/TPs que nous ferons ensemble se trouveront sur ma page :

`http://webusers.imj-prg.fr/~florent.capelli/`

avec les corrections (et les couleurs !). Si vous avez une quelconque question, vous pouvez m'écrire un mail à :

`fcapelli@math.univ-paris-diderot.fr`

2 Introduction

Nous rappelons ici que le C est un *langage de programmation*, c'est-à-dire une façon de spécifier sous forme de texte "compréhensible" un programme. Ce texte est ensuite donné à un *compilateur* qui va le transformer en instructions machine que l'ordinateur pourra exécuter. Il existe plusieurs compilateurs différents pour C qui ont chacun leurs spécificités et qui pour un même code, ne produiront pas exactement le même programme. On peut citer, par exemple :

- GCC
- Borland Turbo C
- Visual C++ Express
- CompCert

Dans ce cours, nous utiliserons *uniquement* GCC, qui peut être installé gratuitement sur toute plateforme (GCC est sous licence GPL). Si le choix du compilateur a une influence sur le résultat produit, le choix de l'éditeur n'a pas vraiment d'importance. Choisissez donc celui qui vous convient le mieux et avec lequel vous êtes le plus à l'aise, cela peut vous faire gagner beaucoup de temps.

3 Au travail !

Exercice 1. [Gallimard] Le but de cet exercice est de découvrir un éditeur de texte, qui vous permettra de produire vos fichiers source. Le choix est entièrement vôtre, mais nous donnons une liste pour vous aider.

Sous linux :

- gedit (conseillé pour les débutants qui ne savent pas vraiment quoi choisir)
- emacs (si vous voulez un éditeur qui fait tout sauf le café)
- vim (si vous avez des miettes dans la barbe)

Sous Mac :

- Xcode
- Aquamacs

Le plus important, c'est que votre éditeur colore votre code (cela le rend beaucoup plus lisible). Vous avez choisi ? Alors au travail !

1. Assurez-vous de savoir : ouvrir/enregistrer un fichier, rechercher un mot dans le texte, remplacer toutes les occurences d'un mot par un autre.
2. Créez un nouveau fichier `hello.c` dans lequel vous taperez le code suivant :

```c
#include <stdio.h>

int main() {
  printf("Hello world !\n");
  return 0;
}
```

Vérifiez bien que le printf et le return sont alignés, un peu en retrait par rapport aux accolades (une tabulation). Cela améliore grandement la lisibilité du code.

Exercice 2. [Phase terminal] Dans cet exercice, nous rappelons quelques commandes utiles dans un terminal et compilons notre premier programme ! Quand vous avez un doute sur une commande, tapez `man nom_de_la_commande` pour avoir des explications (cela marche aussi pour les fonctions standard du C).

1. Ouvrez un terminal. Observez qu'à gauche du curseur sont indiqués : le nom de l'utilisateur, le nom de l'ordinateur et le dossier courant. Par exemple : `bob@lamaison:~/documents$`. Le `~` est une abbréviation pour `/home/bob/`. Un `.` désigne le répertoire courant et `..` le répertoire parent. En utilisant la commande `cd` (pour *change directory*), naviguez dans l'arborescence des dossiers jusqu'au dossier où vous avez enregistré `hello.c`. Appuyez deux fois sur la touche tab pour autocompléter les noms de dossiers. Vous pouvez afficher les fichiers du dossier courant avec la commande `ls`.
2. Créez le dossier `tp0` avec la commande `mkdir tp0` et déplacez le fichier `hello.c` dans ce répertoire avec la commande `mv hello.c tp0/`. Allez dans le dossier `tp0`.
3. Compilez le fichier `hello.c` avec la commande `gcc hello.c`. Un nouveau fichier `a.out` a été créé. Vous pouvez l'exécuter avec la commande `./a.out`. Vous pouvez aussi choisir le nom de votre exécutable au moment de la compilation avec l'option `-o` : `gcc hello.c -o hello.out`

   Astuce 1. *Vous pouvez faire défiler les dernières commandes utilisées en appuyant sur la flèche du haut.*
4. Déclarez dans `hello.c` une variable `x` de type `integer` sans l'initialiser (`int x`), compilez. Compilez à nouveau en utilisant l'option `-Wall` (pour Warning All) : `gcc hello.c -Wall`. GCC vous avertit désormais du risque de ne pas initialiser ses variables. Compiler avec `-Wall` est une bonne habitude.

Exercice 3. [L'int qui voulait être plus gros que le long] Le but de cet exercice de comprendre comment afficher des informations à l'écran, apprendre à lire et modifier un programme simple et se sensibiliser aux problèmes liés au codage des entiers.

Créez un nouveau fichier `pow.c` et recopiez-y le code suivant :

```c
#include <stdio.h>

int main() {
  int n = 2, i = 5;
  int m = 1, j = 0;

  for (j = 0; j < i; j++) {
    m = m*n;
  }
  printf("Le resultat est %d", m);
  return 0;
}
```

1. Quelle fonction $f(n, i)$ ce programme calcule-t-il ?

   Réponse. *Le programme affiche n^i. Pour prouver cela formellement, on peut noter m_j la valeur de la variable m avant la j-ième itération de la boucle puis prouver par induction $m_j = n^j$. Pour $j = 0$, on a $m_0 = 1 = n^0$. Puis dans la boucle, on a $m_{j+1} = n \times m_j$, d'où par induction $m_{j+1} = n \times n^j = n^{j+1}$.*
2. Modifiez le programme pour qu'il affiche "Le resultat de f(n,i) est m" où n,i,m sont remplacés par leurs valeurs.

   Réponse. *Pour afficher plusieurs valeurs dans un* `printf`*, on commence par indiquer la position de chaque valeur entière avec "%d", puis on indique chaque valeur séparée par une virgule dans leur ordre d'apparition :*

```c
#include <stdio.h>

int main() {
  int n = 2, i = 5;
  int m = 1, j = 0;

  for (j = 0; j < i; j++) {
    m = m*n;
  }
  printf("Le resultat de %d puissance %d est %d", n,i,m);
  return 0;
}
```

3. Modifiez le programme pour qu'il calcule $f(3,5)$, $f(9,7)$ puis $f(2,31)$. Qu'observez-vous pour le dernier ? Remplacez tous les `int` par des `long`, les `%d` par des `%ld` et calculez $f(2,31)$ à nouveau. Calculez désormais $f(2,63)$. Que se passe-t-il ?

   Réponse. *On remarque que pour les dernières valeurs, le programme affiche des nombres négatifs. Son comportement n'est plus correct.*

4. Pour expliquer ce comportement, il faut comprendre comment les nombres sont représentés en machine. Avec n bits, on peut représenté 2^n nombres différents. On veut pouvoir représenter des nombres négatifs aussi. Pour cela, on utilise la représentation en *complément à deux*. Avec n bits, on représente les nombres de 0 à $2^{n-1}-1$ en les écrivant en binaire sur $n-1$ bits (et en laissant le dernier bit à 0). On représente les entiers x entre -2^{n-1} et 0 par la représentation binaire de $2^n + x$.

   (a) Sachant que les `int` sont une représentation des entiers en complément à 2 sur 32 bits, les `long` sur 64, expliquez les erreurs du programme précédent.

   Réponse. *Avec 32 bits, en complément à deux, on ne peut pas représenter 2^{31}. Maintenant, l'écriture en binaire de 2^{31} représente en complément à deux le nombre $-2^{31} = -2147483648$ ce que votre programme affiche !*

   (b) En C, on peut aussi utiliser les `unsigned int` ou `unsigned long` qui représentent les nombres positifs sur 32 et 64 bits respectivement. Quelles sont alors les plus grandes valeurs pouvant être représentées par ces types ? Essayez-les dans `pow.c`.

   Réponse. *Remarquons que pour afficher un* `unsigned int` *on utilise %u et %lu pour un* `unsigned long int`.

   (c) Créez un nouveau fichier `factorielle.c` et, en modifiant le code de `pow.c`, écrire un programme qui affiche $n!$ pour n une variable. Quel type avez-vous utilisé pour stocker votre résultat ? Pourquoi ? À partir de quel n votre programme n'est-il plus correct ?

   Réponse. *On peut utiliser* `long` *ou* `int`*, cela changera juste la valeur à partir de laquelle le programme n'est plus correct. Cela dépend donc de l'application. Pour un* `int`*, le programme n'est plus correct pour $i = 13$. On peut trouver cette valeur en écrivant : $2^{31} \leq i!$ ssi $31 \leq \sum_{j=1}^{i} \log(i)$. Pour $i = 8$, $\log i = 3$, et on voit aisément que la somme ci-dessus et plus petite que $8 \log 8 = 24$. Puis $\log 9 < 3.5$ et $\log 10 < 3.5$ donc on est sûr que le programme fonctionne encore pour $i = 10$. Reste à tester les valeurs jusqu'à 13.*

```c
#include <stdio.h>

int main() {
  int i = 5;
  int m = 1, j = 0;

  for (j = 1; j <= i; j++) {
    m = m*j;
  }
  printf("Le resultat est %d\n", m);
  return 0;
}
```

Exercice 4. [Fonctions !] On introduit ici le principe de fonction qui vous sera très utile par la suite. Une fonction est une partie du code qui prend des paramètres et retourne une valeur fonction de ces paramètres. On la déclare avant la fonction `main` et on peut l'appeler ensuite sur les paramètres souhaités. Voici un exemple :

```c
#include <stdio.h>

int absolue(int x) {
  int r = 0;
  if (x > 0) {
    r = x;
  }
  else {
    r = -x;
  }
  return r;
}

int main() {
  printf("Le resultat est %d", absolue(-10));
  return 0;
}
```

1. Que calcule la fonction `absolue` ? Adaptez cette fonction pour en faire une fonction `int distance(int x, int y)` qui retourne la distance entre les deux entiers `x,y`. Testez-la.

   Réponse. *Cette fonction calcule sans surprise la valeur absolue. On peut adapter cette fonction ou la réutiliser pour calculer* `distance(x,y) = abs(x-y)` *:*

```c
int distance(int x, int y) {
  int r = 0;
  if (x-y > 0) r = x-y;
  else r = y-x;
  return r;
}
```

2. Sur ce schéma, reprenez les programmes `pow.c` et `factorielle.c` et modifiez-les pour que ce qu'ils calculent soit isolée dans une fonction à part que vous appelerez dans le main.

   Réponse. *On donne ici le code des deux fonctions :*

```c
int factorielle(int n) {
  int r = 1;
  int j;
  for(j=1;j<=n;j++) {
    r = r*j;
  }
  return r;
}
```

```c
int pow(int n, int i) {
  int r = 1;
  int j;
  for(j=1;j<=i;j++) {
    r = r*n;
  }
  return r;
}
```

3. La suite de Fibonacci est définie par : étant donnés u_0, u_1, on a $u_{n+2} = u_{n+1} + u_n$. Écrivez une fonction qui prend en paramètre des entiers u_0, u_1, n en renvoie u_n.

   Réponse. *Attention aux valeurs particulières de* $n = 0, 1$.

```c
int fibonacci(int n, int u0, int u1) {
  int a=u0, b=u1;
  int j;
  for(j=0;j<n;j++) {
    int c=a;
    a=b;
    b=c+b;
  }
  return a;
}
```

4. (question plus difficile) On rappelle l'algorithme d'Euclide : si a, b sont deux entiers naturels tels que $a = b \times q + r$ avec $0 \leq r < b$, alors $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ (on remarque que r est le reste de la division euclidienne de a par b).

   (a) Utilisez cette idée pour écrire une fonction pgcd qui prend deux entiers a, b et renvoie leur pgcd. On pourra utiliser l'opérateur `while(condition) { moncode }` qui exécute `moncode` tant que la condition est vraie. Par exemple, pour écrire les nombres de 10 à 0, on peut utiliser :

```c
int x = 10;
while(x >= 0) {
  printf("%d",x);
  x = x-1;
}
```

   Réponse. *On applique l'égalité ci-dessus jusqu'à ce que $r = 0$. Dans ce cas, on sait que pgcd(b,0) = b.*

```c
int pgcd(int a, int b) {
  while(b != 0) {
    int c=b;
    b = a % b;
    a = b;
  }
  return a;
}
```

   (b) Sachant qu'une fonction peut s'appeler elle-même (on parle de fonction récursive), essayez d'écrire la fonction pgcd sans utiliser de boucle `for` ni de boucle `while`.

   Réponse. *On applique tout simplement la formule :*

```c
int pgcd(int a, int b){
  if (b == 0) return a;
  else return pgcd(b, a%b);
}
```