TP2 : un peu plus de C

Algorithmique et projet de programmation — M1 mathématiques

27 septembre 2016

1 Comme de par hasard

La bibliothèque `stdlib.h` contient une fonction qui génère un nombre (pseudo)aléatoire : `rand()` renvoie un entier aléatoire entre `0` et une (grande) valeur abitraire fixée par `C` et appelée `RAND_MAX`. Pour que `rand` se comporte réellement comme un génrateur aléatoire, il faut l'initialiser avec une *graine*, par exemple l'heure :

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char** argv){

  srand(time(NULL)); //initialize random generation
  int r = rand(); //r is now a random number
  printf("%d is random.\n", r);

  return EXIT_SUCCESS;
}
```

Exercice 1. Écrire un programme qui génère un nombre aléatoire entre 0 et un entier `n` fixé (exclu).

Exercice 2. On peut vérifier expérimentalement si le générateur pseudo-aléatoire est correct : écrire un programme qui tire un millions de fois un nombre au hasard entre `0` et `100` et qui affiche la proportion de `42` parmi ce tirage.

Exercice 3. Écrire un jeu du *plus ou moins*. Le jeu se déroule comme suit :

(1) un nombre `bound` est fixé à l'avance,

(2) un nombre de coups `chances` est fixé à l'avance,

(3) le programme génère un nombre aléatoirement entre `0` et `bound`,

(4) l'utilisateur est invité à entrer un nombre et le programme indique si le nombre cherché est plus grand ou plus petit (ou égal),

(5) on répète cette instruction jusqu'à ce que le joueur trouve le nombre (et dans ce cas il gagne) ou que le nombre de coups soit écoulé (et dans ce cas il perd).

Exercice 4. Échangez les rôles. C'est-à-dire que c'est maintenant l'utilisateur qui choisit le nombre mystère puis l'ordinateur qui essaye de le deviner : à chaque tour, l'ordinateur propose un nombre et le joueur répond `1` si le nombre mystère est plus grand, `-1` s'il est plus petit, et enfin `0` si l'ordinateur a trouvé le nombre.

Faites utiliser une stratégie "bêbête" à l'ordinateur : à chaque tour, il tire un nombre au hasard (dans l'intervalle) .

Exercice 5. Améliorer la stratégie de l'ordinateur :

(1) tout d'abord en le faisant choisir un nombre au hasard dans un intervalle plus malin qu'il mettra à jour avec les informations donnés par l'utilisateur à chaque tour,

(2) puis en lui faisant utiliser la dichotomie : comme précédemment, on met à jour l'intervalle à chaque tour, et on y choisit non plus un élément au hasard mais le milieu pour réduire au maximum la taille de l'intervalle au tour d'après.

2 Des arguments

L'avantage d'appeler un programme/logiciel en ligne de commande plutôt qu'en cliquant sur une icône est la possibilité de passer des *arguments* au dit programme : c'est-à-dire des valeurs que le programme pourra utiliser lors de son exécution. Par exemple, `firefox "www.google.com"` ouvre Mozilla Firefox directement à la page de recherche Google au lieu de la page d'accueil par défaut.

On peut le faire également pour nos programmes écrits en `C` : c'est le rôle des deux variables mystérieuses `argc` et `argv` que je vous incite à placer en arguments de `main`. La variable `argc` de type `int` contient le nombre d'arguments passés au programme tandis que `argv` de type `char**` (tableau de chaînes de caractères) contient les arguments eux-même, sous forme de chaînes de caractères. Pour accéder à l'argument numéro i, on écrit simplement `argv[i]`. Attention : la numérotation commence à `0` (et se finit donc à `argc-1`), et la première case `argv[0]` contient toujours le nom du programme lui-même (en particulier, `argc` est toujours supérieur ou égal à 1).

Exercice 6. Réécrire le programme du plus ou moins en laissant l'utilisateur passer deux arguments en ligne de commande : le premier sera la borne suprieure (appelée `bound` précédemment) et le deuxième le nombre de coups (appelée `chances` précédemment). Indice : il faudra utiliser la fonction `sscanf(s,format,ref)` qui fonctionne comme `scanf` mais prend comme entrée non pas le clavier mais la chaîne `s` passée en premier paramètre. Par exemple, `sscanf(s,"%d",&var)` stocke `10` dans la variable `var` si la chaîne `s` vaut `"10"`.

3 Introduction aux fonctions

L'usage des fonctions en `C`, et de manière générale en programmation, est double : factoriser et modulariser le code. La *factorisation de code* consiste à éviter les répétitions bêtes et méchantes de code. Prenons l'exemple d'un programme mettant en jeu trois entiers, et qui à un moment donné doit vérifier si ces trois entiers sont des carrés parfaits :

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){

  int a, b, c;

  /*  ... computes things with a, b and c ... */

  int i, sqa; //sqa is a boolean
  for(i=0; i*i<=a; i++);
  if(i*i==a){ sqa = 1;} else { sqa = 0;}

  int j, sqb; //sqb is a boolean
  for(j=0; j*j<=b; j++);
  if(j*j==b){ sqb = 1;} else { sqb = 0;}

  int k, sqc; //sqc is a boolean
  for(k=0; k*k<=c; k++);
  if(k*k==c){ sqc = 1;} else { sqc = 0;}

  /* ... code using sqa, sqb and sqc ... */

  return EXIT_SUCCESS;
}
```

On aimerait pouvoir n'écrire qu'une seule fois les instructions déterminant si un nombre est un carré parfait puis l'appliquer à `a`, `b` et `c` indifféremment. C'est exactement ce que réalise une fonction C :

```c
#include <stdio.h>
#include <stdlib.h>

int issquare(int n) {
  int i;
  for(i=0; i*i<=n; i++);
  if(i*i==n){ return 1; } else { return 0; }
}


int main(int argc, char** argv){

  int a, b, c;

  /* ... computes things with a, b and c ... */

  int sqa = issquare(a), sqb = issquare(b), sqc = issquare(c);

  /* ... code using sqa, sqb and sqc ... */

  return EXIT_SUCCESS;
}
```

Mais les fonctions peuvent aussi simplement servir à *modulariser* le code. C'est la pratique qui consiste à garder séparé dans le code les parties du programme que

l'on *pense séparément*. On illustrera cela tout au long du semestre.

Une fonction s'instancie sous la forme suivante :

```
return_type function_name(arg_type1 arg1, arg_type2 arg2, ...) {
  /* code with one or more return of the type return_type */
}
```

Vous avez en fait déjà codé des fonctions C. En effet, tous vos sources contiennent une fonction `main`, où le type de retour est `int`, et qui prend deux arguments, le premier `argc` de type `int` et le second `argv` de type `char**`. La fonction précédente `issquare` a quat à elle pour type de retour `int` et prend un unique argument `n` de type `int`.

Utiliser une fonction dans votre code est aussi naturel qu'appeler une fonction en mathématiques :

```
function_name(val1, val2, ...)
```

À ceci près que `val1` doit être de type `arg_type1`, `val2` de type `arg_type2`, etc. De même, l'expression `function_name(val1,val2,...)` doit être utilisée là où est valide un objet de type `return_type`. Encore une fois, ceci n'est pas une nouveauté, vous utilisez les fonctions C depuis maintenant deux semaines. La fonction `printf`, `scanf`, `rand`, etc.

Exercice 7. Écrire une **fonction** de prototype :

```
float simulate(int n, int k, int inf, int sup)
```

Cette fonction doit simuler le tirage répété n fois d'un entier dans un intervalle (d'entiers) $[\text{inf}, \text{sup}[$ donné et retourner la fréquence d'apparition de k dans ce tirage.

Ainsi le programme de l'exercice 2 doit correspondre au bout de code suivant :

```
printf("42 appears %f%c of the time.\n",
       simulate(1000000, 42, 0, 100), '%');
```

Exercice 8. Améliorer légèrement le programme précédent pour permettre à l'utilisateur de passer en paramètres en ligne de commande le nombre de tirage, le nombre test duquel on renvoie la fréquence, et les bornes de l'intervalle de test.

4 À rendre pour le 4 octobre 2016

Exercice 9. Écrire une **fonction** `prime` de prototype :

```
int prime(unsigned int n)
```

qui prend en argument un entier n positif et renvoie `0` s'il n'est pas premier et `1` s'il l'est.

Utiliser cette fonction dans un programme qui affiche la proportion de nombre premier dans l'intervalle $[0, n]$ pour un nombre t de tirage où n et t sont passés en paramètres en ligne de commande par l'utilisateur.

Exercice 10. Pour tout $m \in \mathbb{N}$, on définit une suite $(s_n^m)_{n\geq 0}$ donnée par

$$s_0^m = m \quad \text{et} \quad \forall n > 0, \begin{cases} s_n^m = \dfrac{s_{n-1}^m}{2} & \text{si } s_{n-1}^m \text{ est pair} \\ s_n^m = 3s_{n-1}^m + 1 & \text{sinon} \end{cases}$$

La conjecture de Syracuse affirme que pour tout $m \in \mathbb{N}$, la suite $(s_n^m)_{n\geq 0}$ stagne en 1. Écrire un programme qui vérifie la conjecture de Syracuse pour $m \in [0, 100]$.