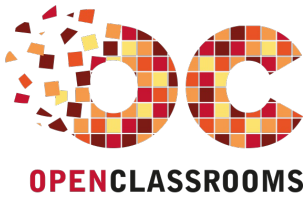
L'aléatoire en C et C++ : se servir de rand

Par Natim
et sebsheep



www.openclassrooms.com

Dernière mise à jour le 6/03/2010

Sommaire

L'aléatoire en C et C++ : se servir de rand

Par  Natim et sebsheep

Mise à jour : 06/03/2010
Difficulté : Facile Durée d'étude : 1 heure

Envie de coder un Yams ? Besoin de simuler un phénomène physique ? Enfin bref, vous voulez utiliser des nombres aléatoires dans votre programme en C ?

Ce tutoriel est là pour vous !

Après avoir décrypté la fonction rand, nous verrons comment générer simplement des nombres aléatoires, entiers ou en virgule flottante. Je vous montrerai ensuite une technique pour générer une suite d'entiers sans doublons, utile pour simuler des cartes tirées dans un paquet par exemple.

Ce tutoriel a été écrit pour le langage C, mais si vous utilisez le langage C++, ce tutoriel est encore valable. Il suffit juste de modifier les "include" : vous devez retirer le ".h" à la fin du nom de la bibliothèque et insérer un "c" au début. Par exemple `#include <stdlib.h>` deviendra `#include <cstdlib>` ; et `#include <time.h>` deviendra `#include <ctime>` .

Sommaire du tutoriel :

Le cœur de l'aléatoire en C : rand

Des nombres pas si aléatoires que ça !

Avant de commencer, il faut que vous ayez en tête que l'on ne peut pas avoir de nombres parfaitement aléatoires avec la méthode utilisée ici. Vous aurez l'**impression** qu'ils le sont, mais en réalité, il ne s'agit que de nombres "pseudo-aléatoires". Si vous voulez en savoir plus, je vous renvoie à l'article traitant de ce sujet. Cela n'est en général pas très gênant, sauf si vous faites de la cryptographie ou d'autres applications où la sécurité est primordiale, car on pourra "deviner" les nombres qui vont sortir. Dans la suite, j'emploierai "aléatoire" à la place de "pseudo-aléatoire" par abus de langage.

Votre première expérience avec l'aléatoire

Vous l'attendez tous, alors voici la fonction qui permet de générer des nombres aléatoires :

Code : C - La fonction rand

```
int rand(void)
```

Elle fait partie de la bibliothèque `stdlib` , vous devrez donc l'inclure (`"stdlib.h"` si vous êtes en C, `"cstdlib"` en C++).

"rand" est la troncature du mot "random" qui en anglais veut tout simplement dire "aléatoire".

Dans la foulée, un exemple pour s'en servir ; le code suivant affiche cinq nombres aléatoires :

Code : C - Génération de 5 nombres aléatoires

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //Ne pas oublier d'inclure le fichier time.h

int main(void){
 int i = 0;
 int nombre_aleatoire = 0;
 for(i=0; i<5; i++){
  nombre_aleatoire = rand();
  printf("%d ",nombre_aleatoire);
 }
 return 0;
}
```

Comme vous le voyez, chaque nouvel appel à `rand` génère un nouveau nombre aléatoire.

Mais elle est nulle ta fonction ! J'ai relancé le programme et j'ai eu exactement les mêmes nombres !

Oui, cela illustre bien ce que je vous disais au début : `rand` va en réalité toujours renvoyer la même séquence de nombres. En l'occurrence pour le programme ci-dessus, on aura :

Code : Console

```
41 18467 6334 26500 19169
```

Donc à priori, pas du tout aléatoire ! En pratique, cette séquence est très (très) longue, c'est ce qui va nous permettre de "faire croire" que les nombres sont tirés au hasard. Mais si on part toujours du même point, ce n'est pas intéressant. On va donc "dire" à rand de se placer dès le départ à une autre position dans la séquence. Mais là, on se mort un peu la queue : il faut choisir "aléatoirement" la position initiale.

Le plus courant est d'utiliser la fonction `int time(int*)` qui est définie dans `time.h` ; cette fonction donne le nombre de secondes écoulées depuis le premier janvier 1970 (il est très rare qu'on lance le programme deux fois dans la même seconde ). On ne se servira pas du paramètre que l'on mettra donc à `NULL`, je vous renvoie à la doc officielle si ça vous intéresse. On l'appellera donc comme ceci : `time(NULL);` .

Il me reste à vous dévoiler comment "dire à rand de se mettre à telle position" : on le fait grâce à la fonction `void srand(int start)` , où `start` indiquera où se placer dans la séquence.

Le code ci-dessus devient alors :

Code : C - Génération de 5 nombres aléatoires

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h> //Ne pas oublier d'inclure le fichier time.h

int main(void){
 int i = 0;
 int nombre_aleatoire = 0;
 srand(time(NULL)); // initialisation de rand
 for(i=0; i<5; i++){
  nombre_aleatoire = rand();
  printf("%d ",nombre_aleatoire);
 }
 return 0;
}
```

Comme vous le voyez, `srand` n'a été appelée qu'une seule fois. En général, il est inutile de la rappeler, c'est `rand` qui se chargera du boulot ensuite.

Restez entre les bornes

Avec des entiers

Je vous vois déjà râler : "Moi, je veux faire un Yams, mais rand me renvoie que des chiffres super grands, je pourrais jamais simuler un dé !"

Nous allons voir ici comment résoudre ce problème en écrivant la fonction `int rand_a_b(int a, int b)` qui renvoie un entier au hasard entre a (inclus) et b (exclu).

Vous ne vous en rendez peut-être pas compte, mais c'est déjà le comportement de `rand` : elle renvoie des entiers entre 0 et `RAND_MAX` , qui est une constante définie dans `stdlib.h` . La valeur de `RAND_MAX` peut varier suivant les compilateurs, mais elle est forcément d'au moins 32767.

Pour revenir à notre problème, on va commencer par générer un nombre entre 0 et un nombre `c`. Pour cela, nous allons utiliser l'opérateur '%' qui renvoie le reste de la division entière ; effectivement, je vous rappelle que le reste de la division de `x` par `c` est toujours compris entre `0` et `c` (exclu). Donc nous avons juste à faire :

Code : Autre

```
rand()%c
```

Maintenant, mettons que vous vouliez tirer un entier dans l'intervalle [a,b[(c'est à dire a inclus et b exclu), il vous suffit de tirer un nombre entre 0 et la longueur de cet intervalle (qui est égale à b-a), puis d'ajouter a. Le nombre sera bien dans [a,b[. Au final, la fonction ressemblera à :

Code : Autre

```
// On suppose a<b
int rand_a_b(int a, int b){
    return rand()%(b-a) +a;
}
```

Un tirage vraiment uniforme ?

Le standard C nous garantit que la répartition des valeurs de rand() entre 0 et `RAND_MAX` est uniforme, c'est à dire qu'on a exactement la même probabilité de tomber sur l'un ou l'autre des entiers contenus dans cet intervalle. Et on doit se poser la question de savoir si cette propriété est respectée par rand_a_b, ce qui n'est pas évident.

Effectivement, pour prendre un exemple, si `RAND_MAX=5`, et qu'on cherche à tirer à pile ou face (donc, soit 0 soit 1) :

- 0 sera obtenu à partir de 0,2 et 4, donc aura une probabilité d'apparition de 3/5=0,6 ;
- 1 sera obtenu à partir de 1 et 3, donc aura une probabilité d'apparition de 2/5=0,4.

La répartition n'est donc pas uniforme.

Bon, alors tout ce qu'on a lu, ça n'a servi à rien ?

Eh bien non. Car ici, on a pris un `RAND_MAX` très petit. Rappelez-vous, il est d'au moins 32767. Dans notre exemple, on aura donc 32767/2 = 16383 apparitions du 1, et 16383+1 apparitions du 0. Je vous laisse faire le calcul pour vous apercevoir que la différence

de fréquence d'apparition est ridicule. Sans faire le calcul précis, on se rend bien compte que, tant que la longueur de l'intervalle est "petite" devant `RAND_MAX`, on peut faire l'approximation que rand_a_b est bien uniforme (c'est en réalité un peu plus complexe qu'une simple histoire de taille d'intervalles, il y a des histoires de divisibilité qui entrent en jeu également ... avis aux fous furieux de l'arithmétique).

La question est après de savoir ce que veut dire "petit". Tout dépend des applications que vous voulez en faire. Si vous faites des simulations physiques/financières, il faut étudier le cas assez précisément et trouver des solutions adaptées. Si vous implémentez une animation simplement esthétique utilisant l'aléatoire (par exemple, une chute de flocons neige), est-ce bien la peine de se prendre la tête ?

Et les flottants ?

Pour générer un nombre entre deux nombres réels a et b (représentés par des `float` et des `double` en C), c'est à peu près la même idée. On a vu que `rand()` était compris entre 0 et `RAND_MAX`. Donc mathématiquement parlant, `rand()/RAND_MAX` est compris entre 0 et 1. Il suffit maintenant de multiplier par `(b-a)` puis d'ajouter `a` pour avoir un nombre compris entre a et b.

Il y a tout de même un petit point technique que je me sens obligé de souligner : par "défaut" en C, la division est entière, c'est à dire que `a/b` est en réalité égal au nombre (entier) maximum de fois que l'on peut "mettre" b dans a. Dans notre cas, vu que `rand()` est inférieur ou égal à `RAND_MAX`, `rand()/RAND_MAX` vaudra 0 dans la plupart des cas, et 1 seulement si `rand()=RAND_MAX`.

On doit donc effectuer une conversion d'une des deux opérandes ; cela forcera la division à se faire comme on l'espère. Pour ce faire, il suffit d'ajouter `(double)` devant la variable que l'on veut convertir.

En réalité, vous ne générerez pas toutes les valeurs entre a et b ; la plus petite distance entre deux nombres tirés sera de `(b-a)/RAND_MAX`. Si (b-a) est petit devant `RAND_MAX`, pas trop de problèmes. Sinon, il faudra trouver d'autres méthodes.

Voilà donc la fonction qui génère des nombres flottants entre 2 bornes :

Code : Autre

```
double frand_a_b(double a, double b){
    return ( rand()/(double)RAND_MAX ) * (b-a) + a
}
```

La génération sans doublons

On est parfois amené à devoir générer une suite de nombres sans doublons ; c'est à dire que si un certain nombre a déjà été tiré, il n'est pas possible de le tirer à nouveau. Les exemples sont multiples : tirer des cartes dans un tas, établir un ordre de passage aléatoire à un examen pour une liste d'étudiants, faire un diaporama aléatoire, ... bref les applications ne manquent pas.

Pour expliquer l'algo, je générerai une liste sans doublons d'entiers de 0 à `MAX` ; la modification à apporter pour avoir une génération entre `MIN` et `MAX` étant minime, elle sera apportée tout à la fin.

Une première idée

La première idée, naïve, qu'on pourrait avoir serait de tirer un nombre au hasard, regarder si ce nombre a déjà été tiré, et en retirer un autre tant que ce nombre n'a pas été tiré.

Mais comment peut-on savoir si un nombre a déjà été tiré ?

On peut procéder ainsi : les nombres tirés sont stockés dans un tableau de taille `MAX`, dans leur ordre d'apparition. La première case contiendra le premier nombre tiré, la deuxième case le deuxième, et ainsi de suite.

Lorsqu'on tire un nouveau nombre, on parcourt le tableau pour voir s'il y est déjà présent ; si c'est le cas, on ne touche pas au tableau et on recommence en tirant un nouveau nombre ; sinon, on ajoute ce nombre à la dernière case libre du tableau. Pour connaître l'indice de cette case, on peut utiliser une variable "compteur" qu'on initialisera à 0 au début, puis qu'on augmentera de

1 à chaque fois qu'on ajoute un nombre.

On a donc potentiellement un algo qui fonctionne. Maintenant, si vous avez du temps à perdre, vous pouvez l'implémenter et tester différentes valeurs de `MAX` : 200, 2 000, 20 000, 200 000 ... Personnellement, ma bécane traite le cas 20 000 en une dizaine de secondes et ne s'arrête pas en temps raisonnable (inférieur à la minute) pour 200 000. C'est dommage, j'ai 200 000 candidats qui vont passer le Bac (à sable) et je dois leur donner un ordre de passage aléatoire...

Ce qui est très long dans cette façon de faire, c'est qu'on parcourt toute la partie du tableau déjà remplie pour savoir si un nombre a été tiré. Si on pouvait connaître instantanément cette information, on gagnerait beaucoup en efficacité. L'idée est donc de la stocker dans un deuxième tableau `test` : si le nombre `i` a déjà été tiré, alors `test[i]` est à `1`, sinon il est à `0`.

> L'intuition nous pousse à penser que cet algo peut ne jamais s'arrêter. C'est faux ! Prenons par exemple le cas où il ne reste plus qu'un nombre à tirer : la probabilité de NE PAS tirer ce nombre au premier coup est extrêmement élevée, mais tout de même strictement inférieure à 1. Notons la `p`. Pour que ce nombre ne soit pas apparu au deuxième coup, il ne doit pas être sorti au premier coup, ET ne doit pas être sorti au deuxième.
>
> Si vous avez déjà manipulé un peu de probas, vous devez savoir que la probabilité de l'événement "A ET B" est égale à la probabilité de l'événement "A" multipliée par celle de l'événement "B" (si les événements sont indépendants, ce qui est le cas). Donc ici la probabilité de "le nombre n'est pas apparu au deuxième coup" est de : $p*p=p^2$.
>
> Et ainsi de suite, vous comprenez sans peine que la probabilité de "le nombre n'est pas apparu au `n`-ième coup" est de p^n. Mais `p` est strictement inférieur à 1. Donc lorsque n devient très grand, p^n devient aussi proche de 0 que l'on veut. Donc l'événement "le nombre n'apparaît jamais" a une probabilité de 0, tout pile.
>
> En conclusion, le nombre de coups nécessaires pour avoir le dernier nombre est bien fini ; il peut en revanche être très grand.

Une approche plus efficace !

L'idée

La dernière version tourne assez rapidement (MAX=200 000 est traité chez moi en un clin d'œil), mais il subsiste deux points noirs :

- comme indiqué dans l'encadré juste au-dessus, même si le nombre de tirages est fini, il peut être très grand, ce qui prendra beaucoup de temps ;
- on a besoin d'un tableau annexe (`test`), ce qui prend de la place.

Pour comprendre le nouvel algo, je vais faire l'analogie avec un jeu de cartes ; remarquez que distribuer toutes les cartes aux joueurs revient à générer sans doublons toutes les cartes et les donner aux joueurs au fur et à mesure. Pour faire cela, est-ce que vous utilisez l'algo que j'ai décrit au dessus lorsque vous jouez aux cartes en famille ? Je ne pense pas (ou alors allez vite consulter un médecin, c'est grave); non, vous **mélangez** vos cartes !

C'est exactement la même idée ici. Nous allons encore avoir besoin d'un tableau (mais d'un seul cette fois). Pour l'initialiser, nous le remplissons "dans l'ordre" : à la case n° 0, on y met 0, à la n°1, on y met 1 et ainsi de suite jusqu'à `MAX`. Ensuite il suffit de le mélanger.

> Alors un paquet de cartes, je vois comment le mélanger, mais un tableau stocké dans la `RAM`, j'ai plus de mal ...

Pas de panique ! L'idée est toute simple : vous tirez un nombre aléatoire `a` entre 0 et `MAX` ; puis vous échangez le contenu de la case n°`0` avec la case n°`a`. Et vous faites ça pour toutes les cases. Cet algo corrige bien les deux écueils que j'avais relevés à propos du premier et génère bien une suite sans doublons.

Il reste une petite broutille : pour l'instant, l'algorithme génère des entiers entre 0 et un nombre. Pour générer une suite à valeurs dans `[a,b[`, il n'y a qu'une modification à faire, lors de la génération du tableau : à la case n°0, on met `a`, à la case n°1, on met `a+1` et ainsi de suite. C'est tout ! Le reste du code se contentera de mélanger les cases, sans se préoccuper de leurs valeurs (il faut tout de même bien penser à tirer les valeurs dans l'intervalle `[0,b-a[` pour que les nombres tirés soient bien des indices du

tableau).

L'implémentation

On pourrait écrire une fonction qui prend en argument `a` et `b`, les bornes de l'intervalle dans lequel on veut générer notre suite, et qui alloue un tableau, lui applique l'algo décrit au-dessus et le renvoie.

Problème : si on veut générer plusieurs suites aléatoires du même intervalle, on devra allouer et initialiser un nouveau tableau à chaque fois. Ce qui prend du temps. Et vous l'aurez remarqué, je n'aime pas en perdre !

On va donc écrire deux fonctions : une qui alloue et initialise le tableau et une qui se contente de mélanger un tableau passé en paramètre. Si on veut une nouvelle suite, il nous suffira de re-mélanger le tableau, sans avoir à le réallouer.

La première ne devrait pas vous poser de problèmes ; elle prend deux arguments entiers `a` et `b` qui sont les bornes de l'intervalle et renvoie un pointeur sur un tableau de taille `(b-a)` contenant `{a, a+1, a+2, ..., b-1}`. La voici :

Code : C

```
int* init_sans_doublons(int a, int b){
 int taille = b-a;
 int* resultat=malloc((taille)*sizeof (int));
 int i=0;
 // On remplit le tableau de manière à ce qu'il soit trié
 for(i = 0; i< taille; i++){
  resultat[i]=i+a;
 }
 return resultat;
}
```

Pour la deuxième, la seule difficulté se situe peut-être dans l'échange des valeurs de deux cases ; pour ce faire, on est obligé d'utiliser une variable `temp`. Cette fonction prend deux arguments : un pointeur vers un tableau et la taille de celui-ci.

Code : C

```
void melanger(int* tableau, int taille){
 int i=0;
 int nombre_tire=0;
 int temp=0;

 for(i = 0; i< taille;i++){
  nombre_tire=rand_a_b(0,taille);
  // On échange les contenus des cases i et nombre_tire
  temp = tableau[i];
  tableau[i] = tableau[nombre_tire];
  tableau[nombre_tire]=temp;
 }
}
```

Mais elle ne renvoie rien cette fonction !?

Effectivement, elle va directement modifier le tableau qu'on lui passe en paramètre, il n'y donc rien besoin de renvoyer !

Voilà, tout est prêt, il ne me reste plus qu'à vous donner un `main` qui permet de tester ces fonctions : le programme suivant génère une suite de nombres compris entre deux nombres entrés par l'utilisateur. Note importante : la fonction `init_sans_doublons` alloue un tableau, **il faut impérativement penser à libérer l'espace par la suite !**

Code : C

```
int main(){
        // A ne pas oublier !
        srand(time(NULL));
```

```
        int a=0;
        int b=0;
        int i =0;
        int* t=NULL; // Va contenir le tableau de nombres

        do{
                printf("Rentrez le premier nombre : ");
                scanf("%d",&a);
                printf("Rentrez le second, plus grand que le premier : ");
                scanf("%d",&b);
        }while(b<=a);

        // On commence pour de vrai ici :
        t=init_sans_doublons(a,b);
        melanger(t,b-a);

        printf("La suite aléatoire est : ");
        for(i=0; i<b-a; i++){
                printf("%d ",t[i]);
        }
        printf("\n");

        // Ne pas oublier de libérer le tableau
        free(t);
        return 0;
}
```

Ici, j'ai choisi de passer la taille du tableau en second argument de la fonction `melanger`. C'est assez désagréable : on est obligé de la calculer (alors que `init` le fait déjà) puis de se souvenir de cette taille pour appeler cette fonction. La solution habituelle est de créer un `struct` qui contient deux champs : le tableau et la taille de celui-ci. `init_sans_doublons` renverrait alors un pointeur vers une structure de ce type, calculant la taille une fois pour toute. Et `melanger` prendrait alors un seul argument : (un pointeur vers) la structure considérée.

Pour des raisons de simplicité, je n'ai pas opté pour cette solution pour le tuto ; dans la vraie vie, si vous devez vous servir à de nombreuses reprises de ces fonctions, je ne peux que vous conseiller de l'implémenter.

Voici un premier aperçu de l'utilisation des nombres aléatoires en C. Si vous voulez vous entrainer un peu, vous pouvez vous amuser à coder des jeux comme un yams ou un Master Mind, une animation en SDL qui simule une chute de neige ou encore plein d'autres applications auxquelles je ne pense pas.

Vous pouvez aussi vous amuser à vérifier "numériquement" ceci : si on a une file de personnes placées aléatoirement contenant Pierre et Paul, le nombre de personnes **le plus probable** entre les deux comparses est de 0 (autrement dit, Pierre et Paul sont côte à côte). Cela vous fera utiliser la génération sans doublon  . Cela paraît surprenant, mais c'est pourtant vrai, un simple calcul de dénombrement permet d'arriver au résultat.

Pour une utilisation un peu plus surprenante de l'aléatoire, vous pouvez allez voir le tuto sur la méthode Monte Carlo.

Bonne continuation et à bientôt pour un autre tuto !