

随机数库的设计

领会接口设计原则最容易的方法就是进行一次简单的设计实践。为达到此目的，本节将讲述开发Stanford类库中random.h接口的整个设计过程，它便编写做出似随机选择的程序成为可能。

1 随机数与伪随机数

使用计算机生成随机过程的概念经常被描述为生成特定范围内的一个随机数 (random-number)，部分原因是早期计算机主要用于处理数值应用。**从理论上讲，一个无法预测其值，且在其取值范围内其值等概率出现的数被称为随机数。**例如，掷骰子时出现一个范围从1到6的随机数。如果骰子是正常的，我们将无法准确预测掷出的数字。并且六个值出现的概率均等。

虽然随机数的概念看起来如此直观，但是要将这一概念在计算机上实现是有困难的。因为计算机总是执行内存中一系列特定序列的指令，因此计算机的函数都是遵循确定的模式的。怎么让遵循确定序列指令的计算机产生不可预测的结果呢？如果一个数是一个确定过程的结果，那么，任何用户都应该能通过一组相同的指令来预测出计算机的响应。事实上，计算机正是通过使用特定的过程来产生所谓的随机数。这一策略之所以可行，是因为理论上用户可以通过同样的一组指令预测出计算机的结果，但实际上并没有人会无聊到去做这种事。

在大多数现实应用中，产生的数字是否真正是随机的关系并不大，真正有影响的是该数字看起来是不是随机的。为了使产生的数字看起来是随机的，它们必须具备如下特点：

1. 从统计的观点来看，该数表现的像一个随机数；
2. 事先要预测这个数的值应该很难，以至于没有人想去预测它。

通过计算机中的一个特定算法来产生的“随机数”也被称作伪随机数 (pseudorandom-number)，这一名称也强调了在该数的产生过程中并不涉及真正的随机活动。

2 标准库中的伪随机数

`<cstdlib>` 类库提供了一个低级的函数 `rand`，调用该函数可以产生伪随机数，`rand`函数的函数原型为：

```
int rand ();
```

此函数原型表明`rand`函数无须传入参数，并且函数返回一个整型值。每一次对`rand`的调用将产生一个让用户难以预测其值的不同的随机数。

`rand`函数的结果是一个大于等于零且小于等于常量`RAND_MAX`的整数，而`RAND_MAX`被定义在中。因此，每一次调用 `rand` 函数，这个函数将返回从 0 到 `RAND_MAX`之间的任意一个整数。

[RandTest](#)程序显示了`RAND_MAX` 的值，然后打印出 10 次调用`rand`函数的结果。

```
1  /*
2   * File: RandTest.cpp
3   * -----
4   * This program tests the random number generator in C++ and produces
5   * the values used in the examples in the text.
6   */
7
8  #include <iostream>
9  #include <iomanip>
10 #include <cstdlib>
11 using namespace std;
12
13 const int N_TRIALS = 10;
14
15 int main() {
16     cout << "On this computer, RAND_MAX is " << RAND_MAX << endl;
17     cout << "The first " << N_TRIALS << " calls to rand:" << endl;
18     for (int i = 0; i < N_TRIALS; i++) {
19         cout << setw(10) << rand() << endl;
20     }
21     return 0;
22 }
```

运行结果如下：

```
1 On this computer, RAND_MAX is 32767
2 The first10 calls to rand:
3     41
4     18467
5     6334
6     26500
7     19169
8     15724
9     11478
10    29358
11    26962
12    24464
```

正如你所看到的，rand函数的值总是正的，并且从来不会大于RAND_MAX的值。而且，函数的值看起来在特定数字范围内不停变动，无法预测，这也正是你想要从一个伪随机过程得到的结果。

如果C++类库中已经存在了一个产生伪随机数的函数，那么我们为什么要重新设计一个呢？

1. 提供的原始API太基础，我们并不需要一个过大范围，或者精度类型不正确的API
2. 如果不对原始API进行包装，我们后续的实现会变得更加复杂。

一部分的答案是rand函数本身并不总是返回用户原意所需要的数值。RAND_MAX的值取决于硬件和软件环境。在大多数系统上，RAND_MAX被定义为整型数的最大值，通常为2147 483 647，但在不同的系统上这一数值可能不一样。即使你能确定RAND_MAX拥有特定的值，也只有少部分应用情况会需要一个在0到2147483647之间的值。

作为一个用户，你更想要的是一个落在另一个数值范围内的随机数，通常这一范围会比上面列举的范围要小。例如，如果你想要模拟抛硬币的过程，你仅需要有两种输出可能性的函数：正面和反面。同样，如果你想要表示一个掷骰子的过程，你需要在1到6中产生一个随机整数。如果你正在尝试模拟物理世界，你需要一个在连续范围内的随机数，这个数需用double类型来表示，而不是int类型。如果你可以设计一个符合以上用户需求的接口，那么这个接口会更加灵活，使用起来也会更加简便。

设计一个更高层次接口的另一个原因是：使用<cstdlib>中的低层次接口会使你在实现的过程中提高程序的复杂性，而这些复杂性正是用户所极力避免的。接口设计师的一部分工作就是要最大化地向用户隐藏这些复杂的实现过程。定义一个高层次的random.h接口使得这一设想成为可能，因为这一接口可以使得复杂性局限于实现过程中。

3 为你的库选择正确的函数集（接口）

作为一个接口设计者，你面临的首要挑战就是选择接口对外提供的函数。虽然接口设计看起来更偏向于艺术方面而不是科学方面，但依然有若干通用原则可供设计中遵循。（原书P58页）

特别是，你在`random.h`中提供的函数必须足够简单，并且应尽可能隐藏其背后复杂的实现过程。同时，这些函数还必须提供必要的关键功能来满足用户的广泛需求，这也意味着你必须清楚了解用户有怎样的需求。理解这些需求的能力不仅取决于你自己的设计经验，也通常需要与潜在用户进行交流以更好地理解这些需求。

以我本人的编程经验，我知道用户期望从`random.h`中获得以下功能：

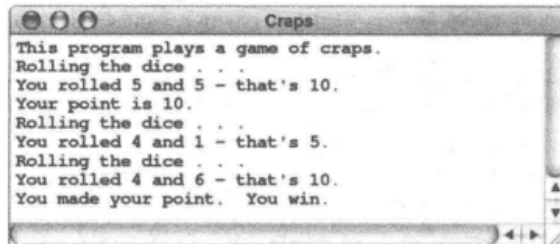
- **从一个特定的区域内选取一个随机整数。**例如，如果你想要模拟掷一个具有六个面的标准骰子的过程，你需要从1到6中随机选择一个整数。
- **从一个特定的区域内选取一个随机实数。**如果你想要一个物体处于空间中的一个随机位置，你需要在所有限制条件中随机生成坐标`x`和`y`值。
- **以某个特定概率模拟一个随机事件。**如果你想要模拟抛硬币这一事件，你需要生成具有概率为0.5的`heads`值，这也意味着在抛硬币时，有50%的概率硬币的正面朝上。将这些概念上的操作转换为一组函数原型是一个相对直接的工作。

`random.h`中的三个函数（`randomInteger`、`randomReal`、`randomChance`）对应了这三个操作。

4 提前编写单元测试

验证接口设计的最好方法就是编写一个程序并使用它。图 2-12 所示的程序给出了使用 `randomInteger` 函数来模拟一种称为 craps 的赌场游戏。craps 的规则在程序开头的注释中进行了介绍，你也可以让程序向玩家介绍这一规则。在这个例子中，为了节省空间我们省略了打印玩家指南这一步骤。

尽管 `Craps.cpp` 程序是不确定性的，并且每次会产生不同的输出，运行该程序一个可能的实例如下：



```
Craps
This program plays a game of craps.
Rolling the dice . . .
You rolled 5 and 5 - that's 10.
Your point is 10.
Rolling the dice . . .
You rolled 4 and 1 - that's 5.
Rolling the dice . . .
You rolled 4 and 6 - that's 10.
You made your point. You win.
```

5 随机数库的实现 ♥♥

1. Generate a random real number d in the range $[0 \dots 1)$.
2. Scale the number to the range $[0 \dots N)$ where N is the number of values.
3. Translate the number so that the range starts at the appropriate value.
4. Convert the result to the next lower integer.

[RANDOM.H](#)

[RANDOM.CPP](#)

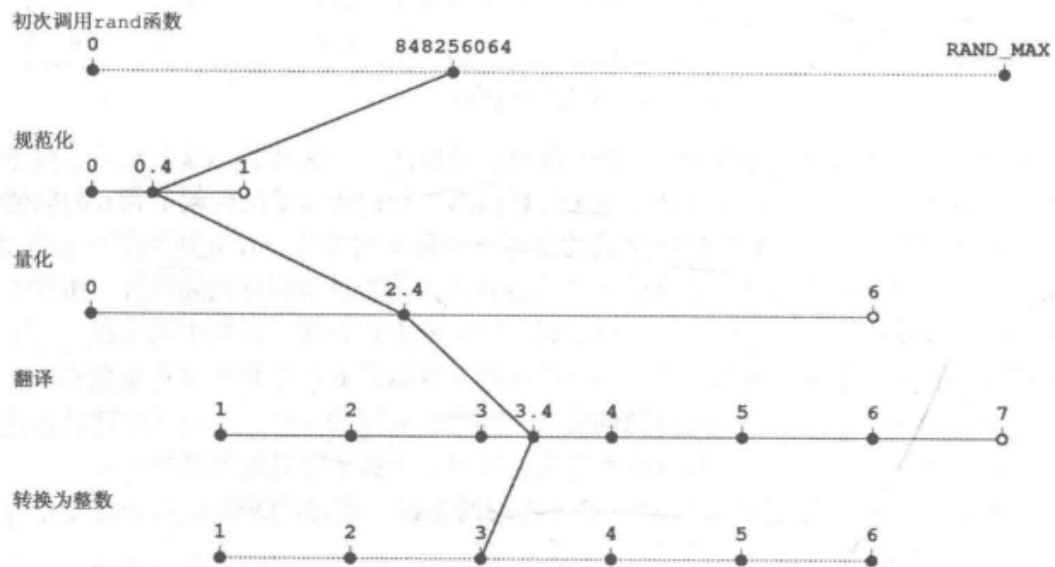


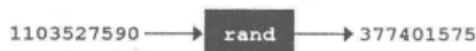
图 2-13 生成 1 ~ 6 区域的一个随机数所需的步骤

6 随机数种子

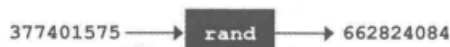
RandTest 程序每次都会产生完全相同的输出，因为C++类库（该类库的基础也就是早期的 C 语言库）的设计者设计的rand函数每次运行都会产生相同的随机序列。

首先，你可能很难理解产生随机数的函数为什么会总是返回同一序列的值。毕竟这种确定的行为似乎与随机概念完全相反。然而，程序的这一行为很好解释：**一个具有确定行为的程序更易于调试。**

与此同时，rand 函数又必须具备输出互不相同结果的功能。为了理解如何实现这一行为，我们必须了解 rand 函数内部的工作原理。rand 函数通过对它最后产生的值进行一系列数学计算来产生一个新的随机数值。因为你不了解这一系列计算过程，因此将上述整个操作看成一个黑箱操作会更好，其中数据从黑箱的一端输入，新的伪随机数将从黑箱的另一端输出。因为第一次调用 rand 函数产生值 1 103 527 590，第二次调用 rand 函数将对应从黑箱的一端输入 1 103 527 590，并在其另一端产生数 377 401 575：



下一次调用 rand 函数时，实现过程将 377 401 575 输入到黑箱中，并返回 662 824 084：



每一次调用 rand 函数将重复上述相同过程。黑箱内部的计算过程被设计为具有如下功能：

- (1) 产生的随机数将均匀地分布在其合法的取值区域内；
- 102 (2) 产生的随机数序列会在很长一段时间后才会重复出现。

但为何在第一次调用 rand 函数的时候会返回 1 103 527 590 呢？因为要实现 rand 函数的计算过程，必须拥有一个开始点。必须存在一个整数 50 被输入到黑箱中，并产生值 1 103 527 590：



这个初始值——即用于启动整个黑箱过程的值被称为随机数产生器的种子 (seed)。在 <cstdlib> 类库中，你可以通过调用 srand (seed) 明确设置该种子值。

正如在多次运行 RandTest 程序时你所看到的，C++ 类库在每次程序启动时都初始化种子为一个常量值，这就是 rand 函数总是输出相同序列随机数的原因。然而这一行为仅在调试阶段会非常有用。很多现代程序设计语言都改变了默认值行为，以便随机数库中的函数每次运行时都返回不同的值，除非程序员进行了特殊设置。为了用户使用 rand 函数更加简单，其设计修改已体现在 random.h 接口中。但是允许用户能产生重复的随机数序列值仍然是必需的，因为这样做简化了调试程序过程。对这一选择的需求也是我们在 random.h 接口中提供 setRandomSeed 函数的原因。在调试阶段，你可以在 main 函数的开头增加以下语句：

```
setRandomSeed(1);
```

之后，对 random.h 接口函数的调用将产生重复的以 1 为初始种子的随机数序列。当确保程序可正常运行后，你可以删除这一语句以恢复程序产生不可预测的结果。