Введение

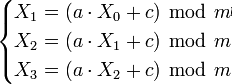
Генераторы случайных чисел — ключевая часть веб-безопасности. Небольшой список применений:

* Генераторы сессий(PHPSESSID)
* Генерация текста для капчи
* Шифрование
* Генерация соли для хранения паролей в необратимом виде
* Генератор паролей
* Порядок раздачи карт в интернет казино

#### Как отличить случайную последовательность чисел от неслучайной?

Пусть есть последовательность чисел: 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9. Является ли она случайной? Есть строгое определение для случайной величины. Случайная величина — это величина, которая принимает в результате опыта одно из множества значений, причём появление того или иного значения этой величины до её измерения нельзя точно предсказать. Но оно не помогает ответить на наш вопрос, так как нам не хватает информации для ответа. Теперь скажем, что данные числа получились набором одной из верхних строк клавиатуры. «Конечно не случайная» — воскликните Вы и тут же назовете следующие число и будете абсолютно правы. Последовательность будет случайной только если между символами, нету зависимости. Например, если бы данные символы появились в результате вытягивания бочонков в лото, то последовательность была бы случайной.

#### Линейный конгруэнтный ГПСЧ(LCPRNG)

Распространённый метод для генерации псевдослучайных чисел, не обладающий криптографической стойкостью. Многие языки программирования, например C(rand), C++(rand) и Java используют LСPRNG. Линейный конгруэнтный метод заключается в вычислении членов линейной рекуррентной последовательности по модулю некоторого натурального числа m, задаваемой следующей формулой:  
  
image  
  
где a(multiplier), c(addend), m(mask) — некоторые целочисленные коэффициенты. Получаемая последовательность зависит от выбора стартового числа (seed) X0 и при разных его значениях получаются различные последовательности случайных чисел.  
  
Для выбора коэффициентов имеются свойства позволяющие максимизировать длину периода(максимальная длина равна m), то есть момент, с которого генератор зациклится.  
  
Пусть генератор выдал несколько случайных чисел X0, X1, X2, X3. Получается система уравнений  
  
  
  
Решив эту систему, можно определить коэффициенты a, c, m.

C provides random number generation function rand() that is found in <stdlib.h> header.  
  
consider the following C statement:

|  |  |  |
| --- | --- | --- |
|  | i = rand(); |  |

The rand function generates an integer between 0 and RAND\_MAX

Standard C states that the value of RAND\_MAX must be at least 32767, which is the maximum value for a two-byte (i.e., 16-bit) integer.

The value of the RAND\_MAX differs from one compiler to another you can check the exact value of the RAND\_MAX for your compiler simply by the following code.

#include <stdlib.h>

#include <stdio.h>

/\* function main begins program execution \*/

int main() {

printf("%d", RAND\_MAX);

return 0; /\* indicates successful termination \*/

} /\* end main \*/

#include <stdio.h>

#include <stdlib.h>

/\* function main begins program execution \*/

int main(void) {

int i; /\* counter \*/

/\* loop 20 times \*/

for (i = 1; i <= 20; i++) {

/\* pick random number from 1 to 6 and output it \*/

printf("%d ", 1 + (rand() % 6));

/\* if counter is divisible by 5, begin new line of output \*/

if (i % 5 == 0) {

printf("\n");

} /\* end if \*/

} /\* end for \*/

return 0; /\* indicates successful termination \*/

} /\* end main \*/

2 5 4 2 6

2 5 1 4 2

3 2 3 2 6

5 1 1 5 5

**Notice that there is a problem with the above programs is that if you run any of the above programs again you will find that it produces the same numbers**and I am going to explain this in the next section.  
  
  
Function rand actually generates pseudorandom numbers. Calling rand repeatedly  
produces a sequence of numbers that appears to be random.   
However, the sequence repeats itself each time the program is executed this can help you debug your program that uses rand function.  
Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution.   
This is called randomizing and is and that can be done using the standard library function **srand**.   
Function srand takes an unsigned integer as a parameter and seeds function rand to produce a different sequence of random numbers for each execution of the program.

#include <stdlib.h>

#include <stdio.h>

/\* function main begins program execution \*/

int main(void) {

int i; /\* counter \*/

unsigned seed; /\* number used to seed random number generator \*/

printf("Enter seed: ");

scanf("%u", &seed); /\* note %u for unsigned \*/

srand(seed); /\* seed random number generator \*/

/\* loop 10 times \*/

for (i = 1; i <= 10; i++) {

/\* pick a random number from 1 to 6 and output it \*/

printf("%10d", 1 + (rand() % 6));

/\* if counter is divisible by 5, begin a new line of output \*/

if (i % 5 == 0) {

printf("\n");

} /\* end if \*/

} /\* end for \*/

return 0; /\* indicates successful termination \*/

} /\* end main \*/

**Note that when I re-entered the number 3 again in the last run it produced the same numbers as the first run because the seed values are equal.**   
Now if we want to randomize using a seed but not have to enter the seed each time we execute the program we can write something like this:  
  
srand( time( NULL ) );   
This causes the computer to read its clock to obtain the value for the seed automatically.  
Function time returns the number of seconds that have passed since midnight on January 1, 1970. This value is converted to an unsigned integer and used as the seed to the random number generator.   
Function time takes NULL as an argument and it is found in the header time.h

##### **Виртуальные функции и ключевое слово virtual**

К моему удивлению, я очень часто сталкивался и сталкиваюсь с людьми (да что там говорить, я и сам был таким же), которые считают, что **ключевое слово virtual делает функцию виртуальной только на один уровень иерархии**. Объясню, что имеется в виду, на примере:

1. #include <cstdlib>
2. #include <iostream>
4. using std::cout;
5. using std::endl;
7. struct A
8. {
9. virtual ~A() {}
11. virtual void foo() const { cout << "A::foo()" << endl; }
12. virtual void bar() const { cout << "A::bar()" << endl; }
13. void baz() const { cout << "A::baz()" << endl; }
14. };
16. struct B : public A
17. {
18. virtual void foo() const { cout << "B::foo()" << endl; }
19. void bar() const { cout << "B::bar()" << endl; }
20. void baz() const { cout << "B::baz()" << endl; }
21. };
23. struct C : public B
24. {
25. virtual void foo() const { cout << "C::foo()" << endl; }
26. virtual void bar() const { cout << "C::bar()" << endl; }
27. void baz() const { cout << "C::baz()" << endl; }
28. };
30. int main()
31. {
32. cout << "pA is B:" << endl;
33. A \* pA = new B;
34. pA->foo();
35. pA->bar();
36. pA->baz();
37. delete pA;
39. cout << "\npA is C:" << endl;
40. pA = new C;
41. pA->foo(); pA->bar(); pA->baz();
42. delete pA;
44. return EXIT\_SUCCESS;
45. }

\* This source code was highlighted with Source Code Highlighter.  
  
Итак, имеем простую иерархию классов. В каждом классе определены 3 метода: foo(), bar() и baz(). Рассмотрим **неверную логику** людей, которые находятся под действием **мифа**:  
когда указатель pA указывает на объект типа B имеем вывод:

pA is B:  
B::foo() // потому что в родительском классе A метод foo() помечен как virtual  
B::bar() // потому что в родительском классе A метод bar() помечен как virtual  
A::baz() // потому что в родительском классе A метод baz() не помечен как virtual

когда указатель pA указывает на объект типа С имеем вывод:

pA is C:  
С::foo() // потому что в родительском классе B метод foo() помечен как virtual  
B::bar() // потому что в родительском классе B метод bar() не помечен как virtual,  
// но он помечен как virtual в классе A, указатель на который мы используем  
A::baz() // потому что в классе A метод baz() не помечен как virtual

С невиртуальной функцией baz() всё и так ясно. А вот с логикой вызова виртуальных функций есть неувязочка. Думаю, не стоит говорить, что на самом деле вывод будет следующим:

pA is B:  
B::foo()  
B::bar()  
A::baz()  
  
pA is C:  
C::foo()  
C::bar()  
A::baz()

Вывод: виртуальная функция становится виртуальной до конца иерархии, а ключевое слово***virtual***является «ключевым» только в первый раз, а в последующие разы оно несет в себе чисто информативную функцию для удобства программистов.  
  
Чтобы понять, почему так происходит, нужно разобраться, как именно работает механизм виртуальных функций.

##### **Раннее и позднее связывание. Таблица виртуальных функций**

Связывание — это сопоставление вызова функции с вызовом. В C++ все функции по умолчанию имеют раннее связывание, то есть компилятор и компоновщик решают, какая именно функция должна быть вызвана, до запуска программы. Виртуальные функции имеют позднее связывание, то есть при вызове функции нужное тело выбирается на этапе выполнения программы.  
  
Встретив ключевое слово virtual, компилятор помечает, что для этого метода должно использоваться позднее связывание: для начала он создает для класса таблицу виртуальных функций, а в класс добавляет новый скрытый для программиста член — указатель на эту таблицу. (На самом деле, насколько я знаю, стандарт языка не предписывает, как именно должен быть реализован механизм виртуальных функций, но реализация на основе виртуальной таблицы стала стандартом де факто.). Рассмотрим этот пруфкод:

1. #include <cstdlib>
2. #include <iostream>
4. struct Empty {};
6. struct EmptyVirt { virtual ~EmptyVirt(){} };
8. struct NotEmpty { int m\_i; };
10. struct NotEmptyVirt
11. {
12. virtual ~NotEmptyVirt() {}
13. int m\_i;
14. };
16. struct NotEmptyNonVirt
17. {
18. void foo() const {}
19. int m\_i;
20. };
22. int main()
23. {
24. std::cout << sizeof(Empty) << std::endl;
25. std::cout << sizeof(EmptyVirt) << std::endl;
26. std::cout << sizeof(NotEmpty) << std::endl;
27. std::cout << sizeof(NotEmptyVirt) << std::endl;
28. std::cout << sizeof(NotEmptyNonVirt) << std::endl;
30. return EXIT\_SUCCESS;
31. }

\* This source code was highlighted with Source Code Highlighter.  
  
Вывод может отличаться в зависимости от платформы, но в моем случае (Win32, msvc2008) он был следующим:

1  
4  
4  
8  
4

Что можно понять из этого примера. Во-первых, размер «пустого» класса всегда больше нуля, потому что компилятор специально вставляет в него фиктивный член. Как пишет Эккель, «представьте процесс индексирования в массиве объектов нулевого размера, и все станет ясно» ;) Во-вторых, мы видим, что размер «непустого» класса NotEmptyVirt при добавлении в него виртуальной функции увеличился на стандартный размер указателя на void; а в «пустом» классе EmptyVirt фиктивный член, который компилятор ранее добавлял для приведения класса к ненулевому размеру, был заменен на указатель. В то же время добавление невиртуальной функции в класс на размер не влияет (спасибо [nullbie](https://habrahabr.ru/users/nullbie/) за [совет](http://habrahabr.ru/blogs/cpp/51229/#comment_1353991)). Имя указателя на таблицу отличается в зависимости от компилятора. К примеру, компилятор Visual Studio 2008 называет его \_\_vfptr, а саму таблицу ‘vftable’ (кто не верит, может посмотреть в отладчике :) В литературе указатель на таблицу виртуальных функций принято называть VPTR, а саму таблицу VTABLE, поэтому я буду придерживаться таких же обозначений.  
  
Что представляет собой таблица виртуальных функций и для чего она нужна? Таблица виртуальных функций хранит в себе адреса всех виртуальных методов класса (по сути, это массив указателей), а также всех виртуальных методов базовых классов этого класса.  
  
Таблиц виртуальных функций у нас будет столько, сколько есть классов, содержащих виртуальные функции — по одной таблице на класс. Объекты каждого из классов содержат именно указатель на таблицу, а не саму таблицу! Вопросы на эту тему любят задавать преподаватели, а также те, кто проводит собеседования. (Примеры каверзных вопросов, на которых можно подловить новичков: «если класс содержит таблицу виртуальных функций, то размер объекта класса будет зависеть от количества виртуальных функций, содержащихся в нем, верно?»; «имеем массив указателей на базовый класс, каждый из которых указывает на объект одного из производных классов — сколько у нас будет таблиц виртуальных функций?» и т.д.).  
  
Итак, для каждого класса у нас будет создана таблица виртуальных функций. Каждой виртуальной функции базового класса присваивается подряд идущий индекс (в порядке объявления функций), по которому в последствие и будет определяться адрес тела функции в таблице VTABLE. При наследовании базового класса, производный класс «получает» и таблицу адресов виртуальных функций базового класса. Если какой-либо виртуальный метод в производном классе переопределяется, то в таблице виртуальных функций этого класса адрес тела соответствующего метода просто будет заменен на новый. При добавлении в производный класс новых виртуальных методов VTABLE производного класса расширяется, а таблица базового класса естественно остается такой же, как и была. Поэтому через указатель на базовый класс нельзя виртуально вызвать методы производного класса, которых не было в базовом — ведь базовый класс о них ничего «не знает» (дальше мы все это посмотрим на примере).   
  
Конструктор класса теперь должен делать дополнительную операцию: инициализировать указатель VPTR адресом соответствующей таблицы виртуальных функций. То есть, когда мы создаем объект производного класса, сначала вызывается конструктор базового класса, инициализирующий VPTR адресом «своей» таблицы виртуальных функций, затем вызывается конструктор производного класса, который перезаписывает это значение.  
  
При вызове функции через адрес базового класса (читайте — через указатель на базовый класс) компилятор сначала должен по указателю VPTR обратиться к таблице виртуальных функций класса, а из неё получить адрес тела вызываемой функции, и только после этого делать call.  
  
Из всего вышесказанного можно сделать вывод, что механизм позднего связывания требует дополнительных затрат процессорного времени (инициализация VPTR конструктором, получение адреса функции при вызове) по сравнению с ранним.  
  
Думаю, на примере все станет понятнее. Рассмотрим следующую иерархию:  
  
  
  
В данном случае получим две таблицы виртуальных функций:

|  |  |
| --- | --- |
| Base | |
| 0 | Base::foo() |
| 1 | Base::bar() |
| 2 | Base::baz() |

и

|  |  |
| --- | --- |
| Inherited | |
| 0 | Base::foo() |
| 1 | Inherited::bar() |
| 2 | Base::baz() |
| 3 | Inherited::qux() |

Как видим, в таблице производного класса адрес второго метода был заменен на соответствующий переопределенный. Пруфкод:

1. #include <cstdlib>
2. #include <iostream>
4. using std::cout;
5. using std::endl;
7. struct Base
8. {
9. Base() { cout << "Base::Base()" << endl; }
10. virtual ~Base() { cout << "Base::~Base()" << endl; }
12. virtual void foo() { cout << "Base::foo()" << endl; }
13. virtual void bar() { cout << "Base::bar()" << endl; }
14. virtual void baz() { cout << "Base::baz()" << endl; }
15. };
17. struct Inherited : public Base
18. {
19. Inherited() { cout << "Inherited::Inherited()" << endl; }
20. virtual ~Inherited() { cout << "Inherited::~Inherited()" << endl; }
22. virtual void bar() { cout << "Inherited::bar()" << endl; }
23. virtual void qux() { cout << "Inherited::qux()" << endl; }
24. };
26. int main()
27. {
28. Base \* pBase = new Inherited;
29. pBase->foo();
30. pBase->bar();
31. pBase->baz();
32. //pBase->qux();    // Ошибка
33. delete pBase;
35. return EXIT\_SUCCESS;
36. }

\* This source code was highlighted with Source Code Highlighter.  
  
Что происходит при запуске программы? Вначале объявляем указатель на объект типа Base, которому присваиваем адрес вновь созданного объекта типа Inherited. При этом вызывается конструктор Base, инициализирует VPTR адресом VTABLE класса Base, а затем конструктор Inherited, который перезаписывает значение VPTR адресом VTABLE класса Inherited. При вызове pBase->foo(), pBase->bar()и pBase->baz() компилятор через указатель VPTR достает фактический адрес тела функции из таблицы виртуальных функций. Как это происходит? Вне зависимости от конкретного типа объекта компилятор знает, что адрес функции foo() находится на первом месте, bar() — на втором, и т.д. (как я и говорил, в порядке объявления функций). Таким образом, для вызова, к примеру, функции baz() он получает адрес функции в виде VPTR+2 — смещение от начала таблицы виртуальных функций, сохраняет этот адрес и подставляет в команду call. По этой же причине, вызов pBase->qux() приводит к ошибке: несмотря на то, что фактический тип объекта Inherited, когда мы присваиваем его адрес указателю на Base, происходит восходящее приведение типа, а в таблице VTABLE класса Base никакого четвертого метода нет, поэтому VPTR+3 указывало бы на «чужую» память (к счастью, такой код даже не компилируется).  
  
Вернемся к мифу. Становится очевидным тот факт, что при таком подходе к реализации виртуальных функций невозможно сделать так, чтобы функция была виртуальной только на один уровень иерархии.  
  
Также становится понятно, почему виртуальные функции работают только при обращении по адресу объекта (через указатели либо через ссылки). Как я уже сказал, в этой строке  
Base \* pBase = new Inherited;  
происходит повышающее приведение типа: Inherited\* приводится к Base\*, но в любом случае указатель всего лишь хранит адрес «начала» объекта в памяти. Если же повышающее приведение производить непосредственно для объекта, то он фактически «обрезается» до размера объекта базового класса. Поэтому логично, что для вызова функций «через объект» используется раннее связывание — компилятор и так «знает» фактический тип объекта.