Chapter 29: Random Number Generation

Section 29.1: Basic Random Number Generation

The function `rand()` can be used to generate a pseudo-random integer value between `0` and `RAND_MAX` (`0` and `RAND_MAX` included).

`srand(int)` is used to seed the pseudo-random number generator. Each time `rand()` is seeded wih the same seed, it must produce the same sequence of values. It should only be seeded once before calling `rand()`. It should not be repeatedly seeded, or reseeded every time you wish to generate a new batch of pseudo-random numbers.

Standard practice is to use the result of `time(NULL)` as a seed. If your random number generator requires to have a deterministic sequence, you can seed the generator with the same value on each program start. This is generally not required for release code, but is useful in debug runs to make bugs reproducible.

It is advised to always seed the generator, if not seeded, it behaves as if it was seeded with `srand(1)`.

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

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Possible output:

```
Random value between [0, 2147483647]: 823321433
```

Notes:

The C Standard does not guarantee the quality of the random sequence produced. In the past, some implementations of `rand()` had serious issues in distribution and randomness of the generated numbers. **The usage of `rand()` is not recommended for serious random number generation needs, like cryptography.**

Section 29.2: Permuted Congruential Generator

Here's a standalone random number generator that doesn't rely on `rand()` or similar library functions.

Why would you want such a thing? Maybe you don't trust your platform's builtin random number generator, or maybe you want a reproducible source of randomness independent of any particular library implementation.

This code is PCG32 from pcg-random.org, a modern, fast, general-purpose RNG with excellent statistical properties. It's not cryptographically secure, so don't use it for cryptography.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */
```

```c
typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}
```

And here's how to call it:

```c
#include <stdio.h>
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;

    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);

    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}
```

Section 29.3: Xorshift Generation

A good and easy alternative to the flawed `rand()` procedures, is *xorshift*, a class of pseudo-random number generators discovered by George Marsaglia. The xorshift generator is among the fastest non-cryptographically-secure random number generators. More information and other example implementaions are available on the xorshift Wikipedia page

Example implementation

```c
#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
    w ^= w >> 19U;
    w ^= t;
    return w;
}
```

Section 29.4: Restrict generation to a given range

Usually when generating random numbers it is useful to generate integers within a range, or a p value between 0.0 and 1.0. Whilst modulus operation can be used to reduce the seed to a low integer this uses the low bits, which often go through a short cycle, resulting in a slight skewing of distribution if N is large in proportion to RAND_MAX.

The macro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produces a p value on 0.0 to 1.0 - epsilon, so

```
i = (int)(uniform() * N)
```

will set `i` to a uniform random number within the range 0 to N - 1.

Unfortunately there is a technical flaw, in that RAND_MAX is permitted to be larger than a variable of type `double` can accurately represent. This means that `RAND_MAX + 1.0` evaluates to RAND_MAX and the function occasionally returns unity. This is unlikely however.