```
    Doub doub() {
    Returns a random double-precision floating value between 0. and 1. Slow!!
        return 2.32830643653869629E-10 * ( int32() +
               2.32830643653869629E-10 * int32() );
    }
};
```

Notice that there is a lot of overhead in starting up an instance of `Ranbyte`, so you should not create instances inside loops that are executed many times. The methods that return 32-bit integers, or double floating-point values, are *slow* in comparison to the other generators above, but are provided in case you want to use `Ranbyte` as a test substitute for another, perhaps questionable, generator.

If you find any nonrandomness at all in `Ranbyte`, don't tell us. But there are several national cryptological agencies that might, or might not, want to talk to you!

7.1.5 Faster Floating-Point Values

The steps above that convert a 64-bit integer to a double-precision floating-point value involves both a nontrivial type conversion and a 64-bit floating multiply. They are performance bottlenecks. One can instead directly move the random bits into the right place in the `double` word with `union` structure, a mask, and some 64-bit logical operations; but in our experience this is not significantly faster.

To generate faster floating-point values, if that is an absolute requirement, we need to bend some of our design rules. Here is a variant of "Knuth's subtractive generator," which is a so-called *lagged Fibonacci generator* on a circular list of 55 values, with lags 24 and 55. Its interesting feature is that new values are generated directly as floating point, by the floating-point subtraction of two previous values.

ran.h
```
struct Ranfib {
Implements Knuth's subtractive generator using only floating operations. See text for cautions.
    Doub dtab[55], dd;
    Int inext, inextp;
    Ranfib(Ullong j) : inext(0), inextp(31) {
    Constructor. Call with any integer seed. Uses Ranq1 to initialize.
        Ranq1 init(j);
        for (int k=0; k<55; k++) dtab[k] = init.doub();
    }
    Doub doub() {
    Returns random double-precision floating value between 0. and 1.
        if (++inext == 55) inext = 0;
        if (++inextp == 55) inextp = 0;
        dd = dtab[inext] - dtab[inextp];
        if (dd < 0) dd += 1.0;
        return (dtab[inext] = dd);
    }
    inline unsigned long int32()
    Returns random 32-bit integer. Recommended only for testing purposes.
        { return (unsigned long)(doub() * 4294967295.0);}
};
```

The `int32` method is included merely for testing, or incidental use. Note also that we use `Ranq1` to initialize `Ranfib`'s table of 55 random values. See earlier editions of Knuth or *Numerical Recipes* for a (somewhat awkward) way to do the initialization purely internally.

`Ranfib` fails the Diehard "birthday test," which is able to discern the simple relation among the three values at lags 0, 24, and 55. Aside from that, it is a good,

but not great, generator, with speed as its principal recommendation.

7.1.6 Timing Results

Timings depend so intimately on highly specific hardware and compiler details, that it is hard to know whether a single set of tests is of any use at all. This is especially true of combined generators, because a good compiler, or a CPU with sophisticated instruction look-ahead, can interleave and pipeline the operations of the individual methods, up to the final combination operations. Also, as we write, desktop computers are in transition from 32 bits to 64, which will affect the timing of 64-bit operations. So, you ought to familiarize yourself with C's "`clock_t clock(void)`" facility and run your own experiments.

That said, the following tables give typical results for routines in this section, normalized to a 3.4 GHz Pentium CPU, vintage 2004. The units are 10^6 returned values per second. Large numbers are better.

Generator	`int64()`	`doub()`	`int8()`
`Ran`	19	10	51
`Ranq1`	39	13	59
`Ranq2`	32	12	58
`Ranfib`		24	
`Ranbyte`			43

The `int8()` timings for `Ran`, `Ranq1`, and `Ranq2` refer to versions augmented as indicated above.

7.1.7 When You Have Only 32-Bit Arithmetic

Our best advice is: Get a better compiler! But if you seriously must live in a world with only unsigned 32-bit arithmetic, then here are some options. None of these individually pass Diehard.

(G) 32-Bit Xorshift RNG

state:	x (unsigned 32-bit)
initialize:	$x \neq 0$
update:	$x \leftarrow x \wedge (x \gg b_1)$,
	$x \leftarrow x \wedge (x \ll b_2)$,
	$x \leftarrow x \wedge (x \gg b_3)$;
or	$x \leftarrow x \wedge (x \ll b_1)$,
	$x \leftarrow x \wedge (x \gg b_2)$,
	$x \leftarrow x \wedge (x \ll b_3)$;
can use as random:	x (32 bits, with caution)
can use in bit mix:	x (32 bits)
can improve by:	output 32-bit MLCG successor
period:	$2^{32} - 1$