Computer Programming

Timing. Randomization. Exceptions. Review

Marius Minea

marius@cs.upt.ro

9 January 2017

Date and time (time.h)

time.h contains structures and functions to measure time

clock_t and time_t are real types representing times

struct tm holds a broken-down calendar time (sec, min, ... year)
struct timespec holds time in seconds and nanoseconds

```
clock_t clock(void);
```
returns (approximation) of processor time used
divide by CLOCKS_PER_SEC (usually 10^6) to get time in seconds

```
int timespec_get(struct timespec *ts, int base);
```
gives time in s and ns since a reference point base (use TIME_UTC)

```
struct timespec {
  time_t tv_sec;
  long tv_nsec;
};
```

Measuring time

Place the code to be benchmarked in a loop running many times
- total time: order of seconds (account for limited clock precision)

Ensure compiler doesn't optimize away repetition (check assembly)
- e.g. computing/assigning the same value many times
- may need to use `volatile` specifier for variables
  - (forces writing/reading to memory every time, like in source)

Repeat measurements and make an average.

Time may be affected by other running processes, caching, etc.

Pseudo-random numbers (`stdlib.h`)

Only natural phenomena can be truly random.
Computer uses algorithm to generate numbers ⇒ *pseudo-random*
- period of number generator should be high
- all bits should appear to be random

Quality of `stdlib` random number generator may not be high
(esp. for lower bits)
Need to use special RNG in cryptography applications.

```
int rand(void);
```

returns an integer in range 0 to `RAND_MAX` (at least $2^{15} - 1$)
Re-running program will produce the same sequence of numbers!
⇒ need to initialize state of RNG with a *seed*

```
void srand(unsigned int seed);
```

could use calendar time (seconds) as seed – different in each run
e.g. `srand((unsigned)time(NULL));`

Why exceptions ?

Error handling is absolutely needed for any environment interaction

Also needed when proper result can't be returned
  non-numeric string to number; 5th element of 3-element list

Error situations can happen anywhere in the "normal" control flow
  end-of-file, read error, insufficient memory
  or user-level errors (input does not match format)
  handling complicates code, obscures the main functionality

Functions must be designed to return error conditions
  complicates their interface

User code has to check for errors *at all points*
  and propagate recovery up from from deep within processing

Exceptions as a programming language feature

Exceptions are a control flow mechanism
 different from function call/return, breaking from loops
 can transfer control across functions

Exceptions are *raised* and *caught* (handled)
 can be raised by a library function, or by the user

Imagine a statement that says:

`setup` *exception-name* `in` *protected-code* `with` *handler-code*

When this is executed, the runtime system sets up things so that if the named exception appears (is *raised*/*thrown*) when executing *protected-code*, control is transferred to the handling code.

If nothing happens, execution proceeds with the next statement.

Syntax varies:

Java: `try` *protected-code* `catch (` *exception* `)` *handler-code*

ML: `try` *protected-code* `with` *exception* `->` *handler-code*

Exceptions in C: setjmp/longjmp

```
#include <setjmp.h>
jmp_buf myexc;
...
if (setjmp(myexc)) {
  // nonzero: exception was thrown, handle here
} else {
  // protected code where exception is caught
}
...
// somewhere else, usually in another function
longjmp(myexc, nonzero);     // throws myexc with nonzero param
```

Can handle in a `switch`, to distinguish values from `longjmp`:

```
switch (setjmp(myexc)) {
case 0: /* normal code that may throw myexc */ break;
case val1: ...; break;
case val2: ...; break;
default: /* any other value */
}
```