The Surprising Dynamics of a Simple Year 2000 Bug

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik, Universität Karlsruhe
D-76128 Karlsruhe, Germany
http://wwwipd.ira.uka.de/~prechelt/

Abstract: *These are the reactions (and an analysis of their reasons) of a very simple program containing a rather simple form of century-dependent code. These reactions are extremely surprising and emerge from an interesting daisy-chain of effects.*

Note: In this text, dates will be written in ISO standard notation, e.g. September 21st, 1998 would be 1998-09-21.

1 The bug

The program in question, a short Perl script, contains the following unsuitable statement:

```
sprintf ("19%d-%02d-%02d", $year, $month+1, $day);
```

The correct statement would have been as follows:

```
sprintf ("%d-%02d-%02d", $year+1900, $month+1, $day);
```

The program essentially generates a calendar for a three-month period given as an argument. It relies on the operating system's calendar functionality for computing the day of week and the number of days in each month. The program works by repeatedly converting internal `time()` integer values into a date stamp text and advancing the `time()` value by 86400, the number of seconds in one day. The conversion is performed by the system procedure `localtime()`, which returns the individual date fields (`year, month, day, weekday`), plus the formatting statement given above. The loop stops when the generated time stamp indicates a month after the requested three-month interval.

The unsuitable statement given above was in fact marked as "needs to be changed in the year 2000" and had been written based on the (incorrect) assumption that the `year` value returned by the operating system would always be in the range 0 to 99. In fact, the year value is intended to be "years since 1900". (For all who wonder: I *am* the author of the script.)

2 The effects

Surprise 1: It happened in 1998. In contrast to most expectations, the bug became visible much earlier than January 2000. In August 1998 I needed the calendar for 1999 and called the script for all four quarters of 1999. The script failed to process the end of 1999 correctly.

Surprise 2: The result was an infinite loop. I had never thought of what the actual effect of the bug would be and was surprised to learn that the effect was to keep the script from terminating — as a result it wrote a file of 34 Megabytes before I finally stopped it.

Surprise 3: The different semantics of the ">" operator for strings versus numbers is relevant. So why did the script not stop? After generating the date stamp for 1999-12-31, the script generated 19100-01-01. This should have stopped the script as obviously this date is beyond the requested three month range. However, the loop condition was based on a *string* comparison. Given the intended formatting this should be equivalent to the corresponding arithmetic comparison of year, then month, then day. The stop date had been computed to be 1999-12-31 (last month of the period and largest possible day number), but

```
"19100-01-01" > "1999-12-31"
```

does not hold! (In fact, the actual operator in Perl is called "`gt`" ("greater than") for Strings, not ">", which is valid for numbers only.)

Surprise 4: The script even hit the Year 2038 Bug. The internal Unix clock `time()` is expressed in terms of the number of seconds since midnight 1970-01-01 (called the "epoch"). When using 32-bit integers, this clock overflows on 2038-01-19 at 3:14am. Hence, the script generated date stamps up to 19138-01-19 and then made the underlying internal clock integer value overflow.

Surprise 5: The script jumped back 1847 years. At the point of overflow, adding 86400 brings the two's complement integer value into the negative range. The resulting date is as far before the epoch as 2038-01-19 is after the epoch: 1901-12-14. Unfortunately, the date stamp formatting statement does not provide a leading zero, hence the generated date at that point is 191-12-14.

Surprise 6: ...and then forward 1711 years. From 191-12-14 the script ran nicely through the years until 199-12-31. At that point it jumped forward to 1910-01-01. From there, everything went "normal" until 1969-12-31.

Surprise 7: A fixed point is reached — because of a language feature. One should expect that the internal clock integer values would change back into the positive range now, calendars through the year 2038 be produced again, and the whole thing starting over at 1901. But this is not what happened. Instead the script began generating date stamps for only 1970-01-01 over and over. The reason is the integer handling of Perl: it has none. Instead, all arithmetic variables are in fact implemented as double (64-bit floating point). When calling an operating system procedure which takes a 32-bit integer argument, Perl converts the double value into `int`, ensuring that no overflow occurs. Concretely: At 1969-12-31, the Perl variable holds a value just under 2^{32} and all further additions of another 86400 let Perl convert the result into the maximum 32-bit integer value, namely $2^{32} - 1$. The internal clock time value and hence the generated date stamp do not change any further.

Surprise 8: the non-distinction of `int` and `unsigned int` is relevant. But, hey, if this is so, why did the script jump back to 1901 in the first place? Because our Perl 5.003 treats the target integer value as an unsigned value, but the Unix library call interprets it as signed.

Summary: The following is the list of the years, for which the script generated (complete or partial) calendars as described above. Note that the calendars printed for the years 19100–19138 and 191-199 have incorrect weekdays, as the year printed is not the one used for weekday computation: 1999 19100 19101 19102 19103 19104 19105 19106 19107 19108 19109 19110 19111 19112 19113 19114 19115 19116 19117 19118 19119 19120 19121 19122 19123 19124 19125 19126 19127 19128 19129 19130 19131 19132 19133 19134 19135 19136 19137 19138 191 192 193 194 195 196 197 198 199 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1970 1970 1970 1970 1970 1970 1970 1970 1970 1970 1970 1970 1970 . . .

Epilogue: Is the infinite loop really infinite? One might ask: Wouldn't (in principle) the floating point value in the Perl variable overflow at some point, hence making the script leave its apparent fixed point? The answer is no. Once the floating point value becomes sufficiently large, the addition of 86400 will not change it at all. This happens long before the floating point overflow.

3 Conclusion

We don't have the slightest idea what will happen in the next few years.