Introduction to Testing and Debugging

Note: these examples often invoke **undefined behaviour** so the results can vary depending on platform, optimisation compiler etc.

- 01_RoundingAndTruncation

  - Demonstrates how single and double precision floats can be inexact, and what this can do unexpectedly

  Info:

  Try the program with a few optimisation levels, as this can affect results

  The double version of pi is correct to about 15 dp, the float version to about 7dp

  C literals are double precision by default

  Summing a rather small number many times often doesn't give quite the expected result. This makes it risky as a loop condition as you can get 1 (or rarely more) fewer or extra loops depending on platform, optimisation levels etc

  Try:

  What we set as 0.01 might not come out that way, and what we do see depends on formatting. Try different numbers and compare the results. In particular, try setting n to a power of 2

- 02_CombiningBigAndSmall

  - This code prints a variety of values found by summing floats. In particular it sums small numbers onto a larger one

  Try:

  Adjust the number or range of values in the array

  Adjust the init value to see the effect of a large number on the accuracy of the final sum

  Check what difference it makes to change eg_float to a double. What sort of value for init then makes the error a similar size to the array elements?

-

- 03_Overflows

  - This code demonstrates the min and max values of integers and floats and what happens when they're exceeded

  Info:

  See e.g. http://www.cplusplus.com/reference/climits/

  C only defines the minimum length of integer types. Normal ints must be at least 2 bytes but are nearly always 4. There are also long ints, which (can be) longer than normal ints, and must be at least 4 bytes.

  Floating point numbers also have limits. The IEEE (I triple-E) standards define strictly what behaviour must be. The program 03-05_InfAndNanTable (compile with gcc {name} -o {outputname}) shows the required values of operations on Inf and NaN (see also 05) values.

  Note 1: Here FLT_MAX + 1 is still FLT_MAX because the finite Epsilon for a value this large is more than 1. Try finding the minimum needed to add to get inf

  Note 2: Here we are running into what are called denormal numbers. Usually the base part of the number is a number between 0.0 and 1.0 as in normal scientific notation. In denormal numbers this is a pure decimal. This allows the representation of smaller numbers, but at lower precision.

  Try:

  Try adjusting the number or range of values

  Check the values of LONG_MIN and LONG_MAX and compare them to INT

  There are also short ints SHRT_MIN at least 2 bytes and chars (1 byte)

  Since C99 there are long long ints which are at least 8 bytes. See if your system has LLONG_MIN and LLONG_MAX and if so, their sizes.

  There are also Unsigned ints. These don't have a sign so can be larger. Line 23 shows one gotcha with these types: what is it doing?

- 04_MixedMode

  - This code demonstrates the dangers of combining int and float in calculations

Info:

Try:

In general operands are promoted to the "biggest" type involved in a single operation (biggest here means that ints will be boosted to longs, but can also be boosted to floats). Using the order of operations (BODMAS, PEMDAS etc) work out what is happening. Note that the division operator "associates with" a number to its right. Note also that C doesn't guarantee the order things are evaluated

adjust the erroneous calculation on line 15 to be correct

- 05_NaNContagion

- This program explores NaN, how it arises and how it behaves

Info:

See also the 03-05_InfAndNanTable, the slides or the notes

Try:

Using the facts about NaN comparisons implement an isNaN function using the stub definition in the code. Test it with some numbers.

Modern C (post 99) has an isNaN function in math.h. Look it up and try it

- 06_UninitialisedErrors

- This program shows what can happen if you fail to initialise variables

Info:

This program does something a little subtle: it allocates an array, frees it, and then reallocates it. Depending on what has happened in the meantime, you may or may not happen to reallocate exactly the same block of memory.

Run the program, supplying one integer between 0 and 100.

The first array should be the numbers 1 to 15.

This memory is now freed.

Next is an array of whatever size you entered.

Last is another array.

Depending on the sizes, sometimes one or other array gets that chunk of memory we just freed, and contains what seems to be “data"

This sort of error is very hard to spot, and is a good reason to always initialise your variables!

Try:

Run the program a few times: try passing values of 15, 11, and 18 You probably find that which array contains the apparent data changes

Try also with optimisation on. You probably find the ``junk'' is more junky.

- 07_AccessErrors

- This program shows what happens if you access outside the bounds of an array

Info:

This program has two chunks. Compiler with -DARR or -DBUFF to get them

These sorts of errors are very unpredictable. The first part of this code (ARR) writes beyond it's array bounds. It randomly chooses to write to one of two arrays. On my system this alternates between an access violation, and overwriting the start of the next array.

The second part of the code (BUFF) tries to print a string missing a null-terminator. Usually you get the content of the string and then some junk, often unprintable characters shown as ``?''.

Try:

Adjust how far out of bounds the write goes.

- 08_BadPointer/BadAllocation

- This program shows what happens with failed Pointer or Allocations

Info:

In this example you'll have to ignore the error about integer to pointer conversion.

This code tries to access really bad memory, which is usually a seg-fault.

Try:

Change the pointer values: try NULL, try smaller numbers etc

- 09_Tarpit

  - This code has a logic bug which makes it very very slow.

  Info:

  If you compile and run, you'll see numbers printed. If it takes a very long time or no time at all, adjust the reps parameter until it takes 2-3 seconds

  The code is writing to a file, but accidentally opening it every iteration. You may notice the numbers go in pulses, when the writing actually flushes to disk.

  Use the command-line option `-q' to omit printing i at every step

  Try:

  Recompile the program with -DFIX and see how much faster it is without the unneeded work.