

Exercise One of Two – integer overflow (65 points)

➔ 1. (7.5 points) If a variable counting seconds is stored in a signed **long** 32-bit integer, how many **days** will it take until that integer overflows? (to one decimal place

➔ 2. (2.5 points) What are the maximum and minimum values that can be stored in a **short** 16-bit signed integer?

➔ 3. (5+5 points) Give examples of two **short** 16-bit signed integers that when added together would cause overflow.

[compile integerOverflow.c and play with it to see 16-bit overflow]

Binary Search Bug

FYI Here is an animation of a binary search which walks through the C code:

<https://www.cs.usfca.edu/~galles/visualization/Search.html>

[note: numeric keypad input does not work, use the keyboard's top row]

[search: for the value found at index position 20]

[controls at the bottom allow you to adjust animation speed and stepping]

The following line of code was used for a long time in many standard programming libraries to find the middle point in a range of array indices for a binary search:

```
mid = (low + high) / 2;    // find mid-point in a range of values
```

Your task is to identify and fix the bug in that calculation of mid.

→ 4. (10 points) What is potentially wrong with the `(low + high) / 2` calculation to find the middle point? Under what conditions would the calculation go wrong?

For a demonstration of the problem, compile `MidBugTest.c` found in this week's zip file.

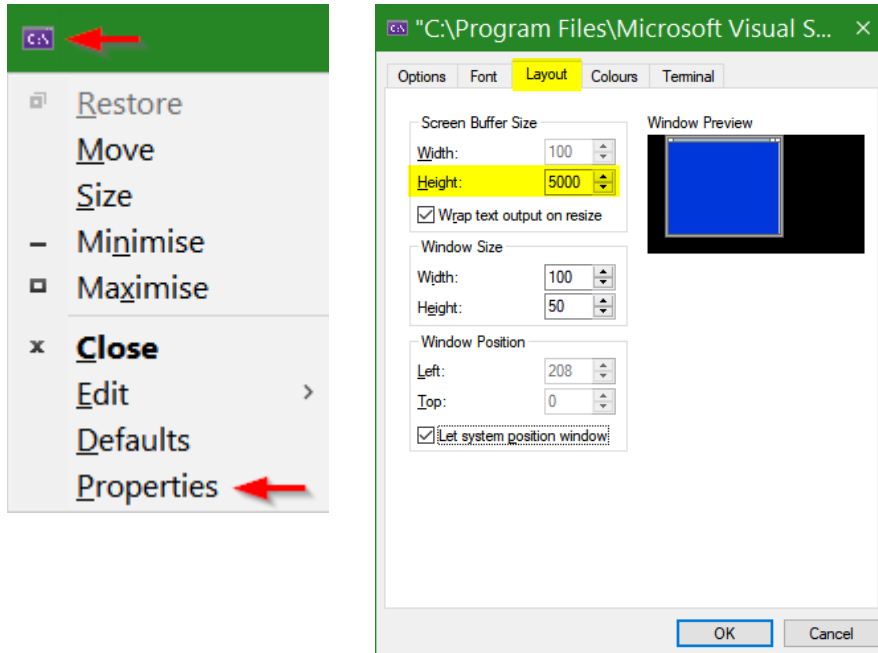
→ 5. (10 points) *See the notes below.*

REWRITE the `mid = (low + high) / 2` calculation to prevent overflow

C source code to test your new formula, `MidBugTest.c` is found in this week's zip file. The code demonstrates the bug using 8-bit integers to keep the test within a reasonable range. (The bit width of the integer is irrelevant to the problem.)

Change the `mid =` line of code to your new formula, compile, and run.
See the comments at the top of that source file.

To view the all output, change the defaults in the console:



Constraints

It must be assumed that `mid`, `low`, and `high` are all variables of the same numeric data type; an arithmetic operation on two variables of the same type always returns a value of that type. A good programmer is always mindful of the possibility of overflow regardless of the capacity of the data type.

Given that `mid` is used in a binary search as an array index, the data type will likely be `int`; that `high` and `low` would also be the same data type as `mid` is a standard programming practice to be assumed. A good programmer would clearly indicate, in comments, whether different data types or bit widths were being mixed. And that is not relevant in the case of this bug.

Casting is computationally expensive and merely avoids the problem, i.e. bad programming and overflow, rather than *solving* the problem. Even though an `int` is signed, cheating by casting `((unsigned int)low + high)` is unwise because the programmer before you may already have declared the three variables as unsigned to increase capacity and avoid the problem until they get another job or retire. Casting an intermediate result to a higher capacity data type will work reliably only if you cast from `int` to `long long`. That is the good news.

The bad news is you are *avoiding* the bug, not solving it. Using the widest bit width is always much slower to process than the default `int` width. Your children will be faced with the same problem when a 64-bit

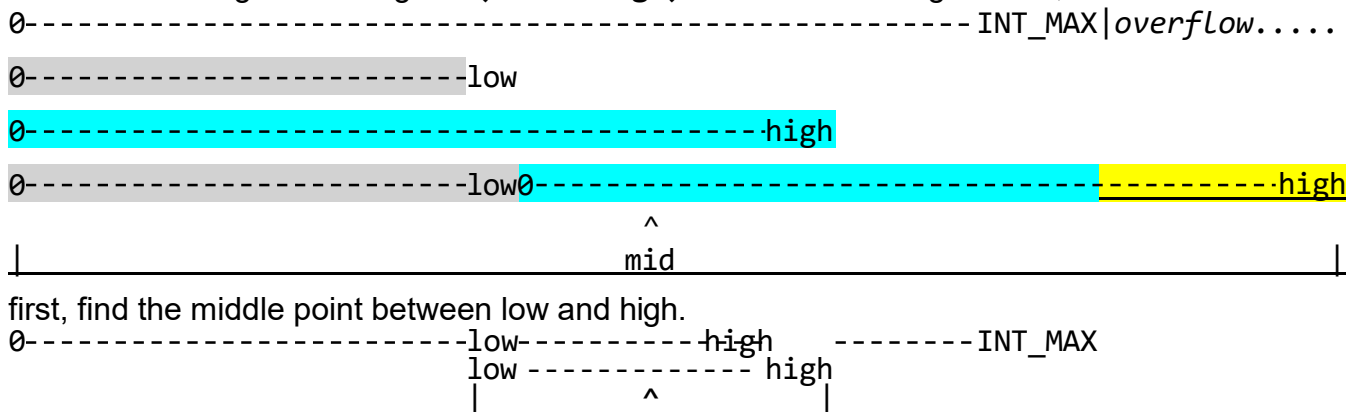
sized `int` becomes the default. You do not want them fixing this by casting to `int128_t` because your grandchildren will inherit the problem. When will it end?

Even if a casted solution works today, casting behaviour has [changed over time](#) and could change on different platforms' compilers so casting is not guaranteed to work tomorrow or even later today; it is also very inefficient compared to a change in the arithmetic. [Occam's Razor](#) applies literally and metaphorically here.

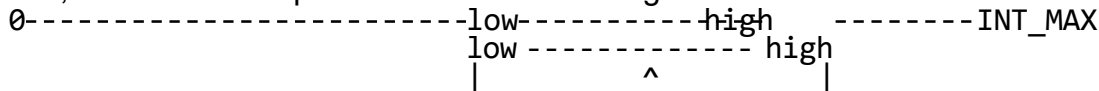
The problem here is not about the data type, its size, sign, or bit width. The challenge is to *rewrite the arithmetic* so that *intermediate values* in the calculation *cannot* overflow. In this instance, `(low + high)` is an intermediate value that the system must resolve to divide its sum by 2. The highest risk case must be assumed here: that both variables are the same data type and therefore subject to overflow when summed together.

Hint:

Instead of finding the average of `(low + high) / 2` as in the bug version,



first, find the middle point between low and high.



Consider using Excel to generate a [Trace Table](#) of expected values. The Trace Table technique is also known as a walkthrough.

→ 6. (25 points) Write a 250+ word “reflection”(similar to a workshop in your programming class) describing the steps you used to develop and test your solution to the binary search bug. This is much more than just about the coding change itself. **To earn the full points**, what were the details of your process from problem analysis to solution implementation? Note that the minimum word count gets more or less average marks depending on the quality of content. To go above average, see the marking rubric in the course Announcements.

When you are done, or just done in, read this: <https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html> “Nearly All Binary Searches and Mergesorts are Broken” — *only one of the suggested solutions in this article works all the time*. Your code should always be portable across platforms meaning it must work in all cases without significant additional runtime overhead.

Without a LOT of programming, most code has the potential to overflow. We must always consider the theoretical risk and design our code to avoid it on principal. When it is not practical or possible to do that,

we then consider the overflow risk relative to the practical use of the variable. A Boeing Dreamliner had almost no risk of running continuously for > 248.55 days when all its electrical power would have shut down. However small the practical risk of it happening, if it did the consequence is unconscionable. The probability of Risk \times impact of Consequence = a constant. A very low risk with a very big consequence (plane crash) is similar to a very high risk with a very small consequence (almost certainly losing the cost of a lottery ticket).

Perhaps it is why the programmers who wrote the binary search thought, "we'll never cause overflow with (high + low) because no one will ever have an array that big." (that big is \geq **21845** or 2/3 of the MAX_INT of a 16-bit integer). That was likely true in the 1980s. At that time, the practical risk of overflow was likely close to zero. Avoiding it by using inefficient 32-bit integers or additional arithmetic was likely considered not worth the runtime penalty on the slow computers of the day. When the default `int` size increased to 32 from 16-bit, then they were back to "no one will ever have an array that big." That kind of thinking had to stop before the consequences started piling up.

Exercise Two of Two – Numbering Systems and Conversions (35 points)

HTML colour codes are expressed in three hexadecimal values like this #0088FF which can represent 16,777,216 colours. # indicates the start of hex digit pairs which read as # 00 88 FF pronounced "hex, zero-zero, eight-eight, Fox-Fox." (three hex values)

Hex digits are 0 – 9 A B C D E F (Able, Baker, Charlie, Dog, Easy, Fox)
for decimal 0 – 9 10 11 12 13 14 15 values in a single hex digit.

All of these are equivalent:

$16 \times 16 \times 16 \times 16 \times 16 \times 16$ (6 hex digits or three hex values, #FFFFFF)


or $256 \times 256 \times 256$ (three byte values)

or $2^8 \times 2^8 \times 2^8$ (three 8 bit values)

or 2^{24} (one 24 bit value)

or 16,777,216 possible decimal values.

Each hex digit pair represents a range of three colours: **Red #FF0000**, **Green #00FF00**, and **Blue #0000FF** known as RGB values.

See <https://www.rapidtables.com/convert/color/index.html> to convert colour codes between HEX and RGB. To further explore colours, go to <https://www.hexcolortool.com/>. The Windows calculator ( + R "calc") has a very handy Programmer function available from the hamburger button.

Roses are **#FF0000**,

Violets are **#0000FF**,

We use hex codes,

And RGB too.

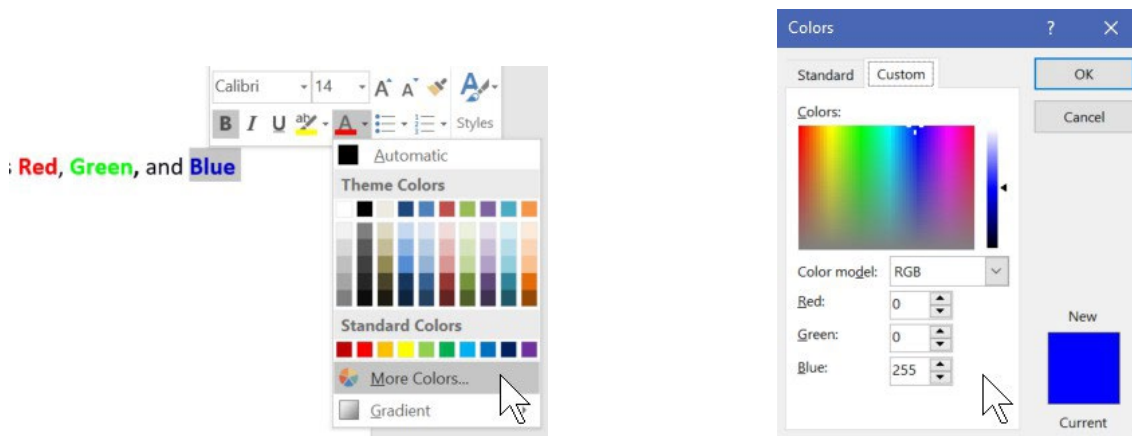
→ 7. (10 points) What is the hex value for these colours?

See the answer document for RGB values to translate.

Red decimal	Green decimal	Blue decimal	Hex triplet	What Colour?
<i>nnn</i>	<i>nnn</i>	<i>nnn</i>	#0x0x0x	

In Microsoft Word, you can edit a font's colour by adjusting its **Red**, **Green**, and **Blue** values. However, Word uses decimal instead of hex values where each colour value can be any decimal number from 0 to 255 (same range as a hex pair 00 – FF). Using the following Hexadecimal values, determine what colour would be produced in Word.

First convert each HEX pair to decimal and then use Font Color / More Colors...Custom to adjust the RGB decimal values:



→ 8. (10 points) Fill in this chart as per the column headings

See the answer document for RGB values to translate.

6 digit Hex code	Red decimal value (0-255)	Green decimal value (0-255)	Blue decimal value (0-255)	Describe the Final Colour and change the cell's background colour, i.e. R-click and see MS Word 'Shading', to match the values for RGB
#0x0x0x	nnn	nnn	nnn	