Exercise One of Two – **integer overflow** (80 points)

➔ 1. (5) A `long` int variable counts time in seconds.
If it contains a value equal to your student number *in reverse sequence*,
how many **days** did it take until that value was reached? (rounded to two decimal places)
e.g. student number 987 654 321 seconds = 11,431.18 days

➔ 2. (20 points) Using your student number *in reverse sequence* as the number of seconds,
how much **time** would that represent in whole numbers
of years, then the remaining days, then hours, then minutes, then seconds?

??	??	??	??	??
YEARS	**DAYS**	**HOURS**	**MINUTES**	**SECONDS**

Most programmers would use Excel as a tool to calculate the answer…
After *n* years, how many days remain? After *n* days, how many hours remain? etc.
(Leap years are usually ignored for general time calcs; use a 365 day year.)
e.g. student number 987654321 contains 987,654,321 seconds in total which is
31 YEARS 116 DAYS 4 HOURS 25 MINUTES 21 SECONDS

➔ 3. (5 points) What is the smallest positive value in a 16-bit signed integer [short] that, when multiplied by 2, would cause overflow?

➔ 4. (5 points) What is the largest negative value in a 16-bit signed integer [short] that, when multiplied by 2, 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`.

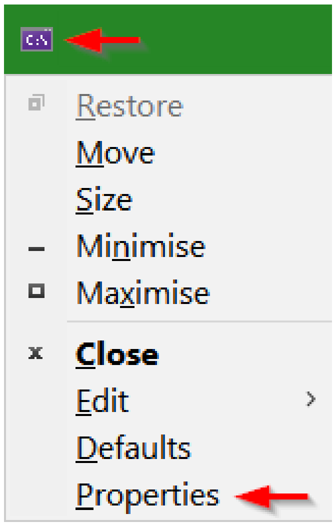
➔ 5. (10 points) What is potentially wrong with the **`(low + high) / 2`** calculation to find the middle point? Under what conditions would the calculation go wrong?
For a demonstration of the problem, compile **`MidBugTest.c`** found in this week's zip file.

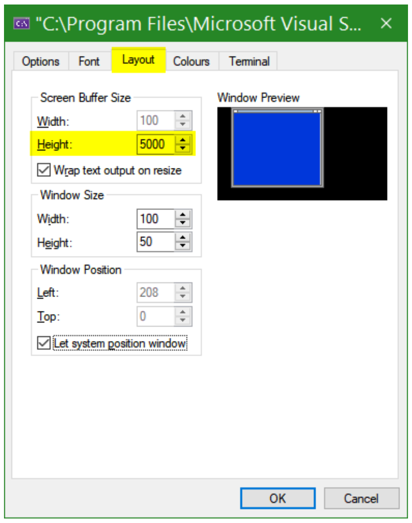
➔ 6. (10 points) *See the notes below.*
REWRITE the `mid = (low + high) / 2` calculation to prevent overflow

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 code and comments at the top of that source file.

To view all terminal output, change the console's defaults:




"C:\Program Files\Microsoft Visual S...

Options | Font | Layout | Colours | Terminal

Screen Buffer Size
Width: 100
Height: 5000
[✓] Wrap text output on resize

Window Size
Width: 100
Height: 50

Window Position
Left: 208
Top: 0
[✓] Let system position window

Window Preview

OK Cancel

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 data 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.

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.

> 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 to a wider bit width is computationally expensive and merely avoids the problem 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 variables' data types 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 works 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 their children will inherit the problem.
>
> 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 here.
>
> Microsoft's C compiler has built-in overflow protection for intermediate values: the result of 8-bit terms in an intermediate formula are stored as 16-bit, 16-bit as 32-bit, 32-bit as 64-bit. It is safe but inefficient.
> **But all C compilers do not do this. Overflow protection should never be assumed.**

Hint:

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

```
0------------------------------------------------------INT_MAX|overflow.....

0-------------------------low

0------------------------------------------high

0-------------------------low0------------------------------------------high
                                    ^
|                                  mid                                       |
```

first, find the middle point between low and high.

```
0-------------------------low--------------high--------INT_MAX
                          low--------------high
                          |         ^         |
```

Consider using Excel to generate a Trace Table of expected values. The Trace Table technique is also known as a walkthrough.

➔ 7. (25 points) You explained what was wrong with the (low + high) / 2 calculation above in 5. and offerred a solution in 6. Now, **how did you get from 5. to 6?** Describe the **steps used to develop and test your solution** to the binary search bug. *(N.B. it is worth 25 points. This is one time when a stream of consciousness answer can be a good thing.)*

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 × 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 what you spent on a lottery ticket).
>
> Perhaps it is why the programmers who wrote the binary search thought, "no one will ever encounter overflow with (high + low) because no one will ever have an array that big." (that big is >= ½ +1 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 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." But that just assumes and pretends that today's problem won't happen until tomorrow.

Exercise Two of Two – **Numbering Systems and Conversions** (20 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:

16×16 × 16×16 × 16×16	(6 hex digits or three hex values, #FFFFFF)
or 256 × 256 × 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 an RGB value.

Roses are **#FF0000**, Violets are **#0000FF**, We use hex codes, And RGB too.

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 (⊞ "calc" *or* ⊞ + R "calc") has a very handy Programmer function available from the hamburger button.

➔ 8. (5 points) What is the hex value for these decimal colours?

Red decimal	Green decimal	Blue decimal	**Hex triplet**

11	*22*	*123*	**#0B167B**

See the answer document for the RGB value to translate into a Hex triplet.

➔ 9. (15 points) What are the decimal values for the hex triplet?
Change the row's Font or Shading colour to match the RBG value.

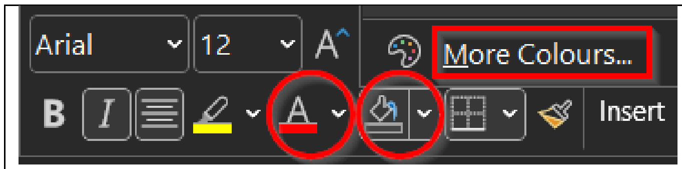
In Microsoft Word, you can edit the font or Shading/Fill colour by adjusting its **Red**, **Green**, and **Blue** values. Change either the three RGB decimal values from 0 to 255 (same range as a hex pair 00 – FF) or simply change the Hex triplet in the same way HTML colours are specified. (see below)

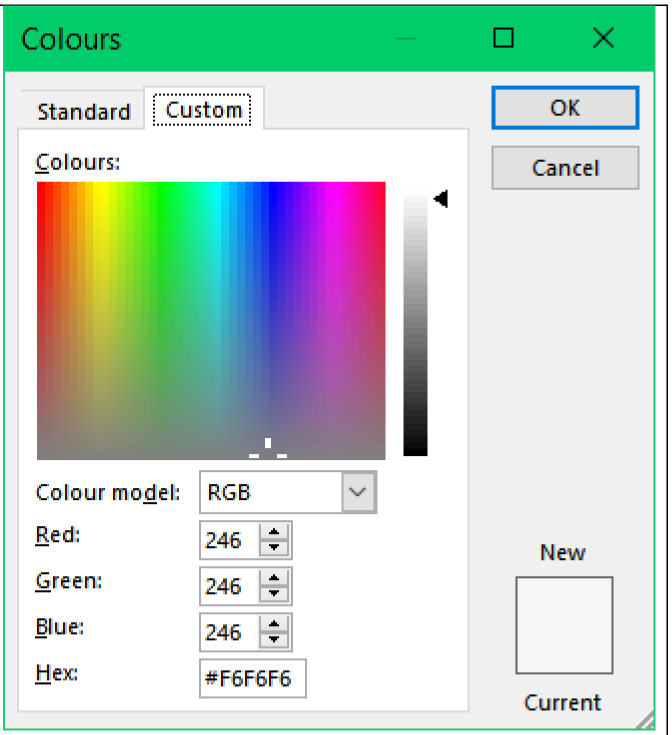
Hex triplet	**Red decimal**	**Green decimal**	**Blue decimal**
#0B167B	**11**	**22**	**123**

← this row's Font colour matches Hex/RGB

See the answer document for the Hex triplet to translate into an RGB value.

Select table row, select Font | Shading, More Colours… to adjust values:





Colour model: RGB
Red: 246
Green: 246
Blue: 246
Hex: #F6F6F6