Tutorial 4
The goal of this tutorial is to reinforce the following materials:

- Overflow
- Data Types
- Structures

Integer Overflow

- Any variable in C takes up a certain amount of memory (bits).
- This limits the range of values that can be represented.
- Any time you try to go past this limit it is called an “overflow”

- A variable of type int allocates 32 bits of memory.
- We want to be able to represent negative numbers as well as positive numbers, so half of this range is negative and half is positive.
- Using this logic, Integers range from $-2^{31}$ to $2^{31} - 1$, or $-2147483648$ to $2147483647$
Overflow

As an **INT** it is impossible to represent outside of the range of:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INT_MIN</strong></td>
<td>$-2^{31}$</td>
<td>$-2147482648$</td>
</tr>
<tr>
<td><strong>INT_MAX</strong></td>
<td>$2^{31} - 1$</td>
<td>$2147482647$</td>
</tr>
</tbody>
</table>

which is why we have **other** data-types

---

**Integer Overflow Example**

The following function can overflow for large values of **a** and **b**.

```c
// find_mid(low, high) returns the middle integer between
// two boundaries, low and high, inclusively
// [round down to the whole integer]
// requires: 0 <= low <= high
int find_mid(int low, int high){
    return (low + high)/2;
}
```

Even though it can never **return** a number larger than **INT_MAX**, the result of computing (**a** + **b**) is undetermined.

**Practice**: On seashell, implement the **find_mid** function that would fix the issue above

---

**Practice Problem: Overflow**

The function **not_overflow_add(a, b)** returns true if adding non-negative integers **a** and **b** will not cause overflow, otherwise, returns false.

For example,

```c
not_overflow_add(1, 0);  // => true
not_overflow_add(INT_MAX, 1); // => false
```
<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
<th>Printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer (numbers)</td>
<td>%d</td>
</tr>
<tr>
<td>char</td>
<td>Characters</td>
<td>%c</td>
</tr>
<tr>
<td>float</td>
<td>Floating Point decimal numbers</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>Double precision floating value</td>
<td>%f</td>
</tr>
</tbody>
</table>

What is a Structure?

- Example:
  - Name: Coolio McCool
  - Date of Birth: 01.01.01
  - Student ID: 0101010010

Structures in Racket vs. C

Structures in C are similar to structures in Racket:

**Racket:**

```racket
(struct posn (x y))
#:transparent
(define p (posn 1 2))
(define a (posn-x p))
(define b (posn-y p))
```

**C:**

```c
struct posn {
    int x;
    int y;
};
const struct posn p = {1,2};
const int a = p.x;
const int b = p.y;
```

Racket generates selector functions when you define a structure. Instead of selector functions, C has a `structure operator (.)` which selects the value of the requested field.
#include "cs136.h"

struct mystruct{
    int number;
    char letter;
};

int main(){
    struct mystruct s = {5,'a'};
    printf ("my struct %d, %c \n", s.number, s.letter);
    s.number = 6;
    s.letter = 'b';
    printf ("my struct %d, %c \n", s.number, s.letter);
    return 0;
}

Practice Problem: Structure – collinear

On Seashell, we have given you the structure posn:

struct posn {
    int x;
    int y;
};

Implement the function collinear(a, b, c) which consumes three posns and returns true if a, b and c are collinear, otherwise returns false.