Tutorial 5

- Overflow
- Data Types
- Structures
- Call stack.
- Stack frames.
- Debugging Tips
Integer Overflow: Introduction

- 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."
Integer Overflow: Explanation

- A variable of type `int` allocates 32 bits of memory.
- We want to be able to represent negative and positive numbers, so roughly half of this range is negative and roughly half is positive.
- Using this logic, Integers range from $-2^{31}$ to $2^{31} - 1$ or $-2,147,483,648$ to $2,147,483,647$
- This is why we have other data types.
Integer Overflow Example

The following function can overflow for large values of \(a\) and \(b\).

```cpp
// 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 \texttt{INT\_MAX}, the result of computing \((a + b)\) is undetermined.

Practice: On seashell, implement the \texttt{find\_mid} function that would fix the issue above
# Data Types

<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?

- Structures are examples of compound data.

- Compound Data Example:

  Name: John Doe
  Date of Birth: 01.01.01
  Student ID: 0101010010
Structures in Racket vs. C

Structures in C are similar to structures in Racket:

Racket:

```
(struct posn (x y))
(define p (posn 1 2))
(define a (posn-x p))
(define b (posn-y p))
```

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.
Practice Problem: Structure – Collinear

On Seashell, we have given you the structure posn:

```c
struct posn {
    int x;
    int y;
};
```

Implement the function `collinear(a, b, c)` which consumes three posn parameters and returns `true` if a, b and c are collinear, otherwise returns `false`. 
Call Stack and Stack Frames

Suppose the function `main` calls `f`, then `f` calls `g`, and `g` calls `h`.

As the program jumps from function to function, we need to remember the history of the return addresses, as well as all the parameters and local variables. This history is known as the `call stack`.

The entries that are pushed onto the call stack are known as `stack frames`.
Each function call creates a new stack frame that contains the following:

```
// ==================================
// <function name>:
//   <parameter one>: <value>
//   <parameter two>: <value>
//   ...
//   <local variable one>: ( <value> || ??? )
//   <local variable two>: ( <value> || ??? )
//   ...
// return address: <caller function name>:<line>
// ==================================
```

If a question asks you to draw the call stack, then **for each** stack frame pushed onto the call stack you must provide the above.
For example:

```c
int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```
Example: Draw the call stack

```c
int g(void) {
    int x = 2;
    return x;
}

int f(int x) {
    int a = g();
    return x * a;
}

int main(void) {
    int x = 4;
    int a = 1;
    ⇒ int y = f(x) + a;
    return address: OS
}
```

==========
main:
  x: 4
  a: 1
  y: ?
  return address: OS
==========
Example: Draw the call stack

```c
int g(void)
{
    int x = 2;
    return x;
}

int f(int x)
{
    int a = g();
    return x * a;
}

int main(void)
{
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

f:
- x: 4
- a: ?
- return address: main:14

main:
- x: 4
- a: 1
- y: ?
- return address: OS
Example: Draw the call stack

```c
int g(void){
    int x = 2;
    ⇒ return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

---

```
g:
    x: 2
    return address: f:7

f:
    x: 4
    a: ?
    return address: main:14

main:
    x: 4
    a: 1
    y: ?
    return address: OS
```
Example: Draw the call stack

```c
int g(void) {
    int x = 2;
    return x;
}

int f(int x) {
    int a = g();
    return x * a;
}

int main(void) {
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

f:
- x: 4
- a: 2
- return address: main:14

main:
- x: 4
- a: 1
- y: ?
- return address: OS
Example: Draw the call stack

```c
int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    ⇒ int y = f(x) + a;
    return address: OS
    main:
        x: 4
        a: 1
        y: 9
    return address: OS
```
Structures on the Stack

- Structures are just groups of regular variables.
- Also stored on the stack in the order fields are declared.
- Passed by value (one reason pointers are useful).
For Example:

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    int x = baz(bar);
    return 0;
}
```
Example: Structures on the call stack

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    ⇒ int x = baz(bar);
    return 0;
}
```

main:
    bar:
        .x = 5
        .y = 6
    x: ?
    return address: OS

=========================
Example: Structures on the call stack

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    ⇒ return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    int x = baz(bar);
    return 0;
}
```

baz:
    qux:
        .x = 5
        .y = 6
    return address: main:12

main:
    bar:
        .x = 5
        .y = 6
    x: ?
    return address: OS
Example: Structures on the call stack

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    int x = baz(bar);
    return 0;
}
```

```
main:
    bar:
        .x = 5
        .y = 6
    x: 6
    return address: OS

=========================
```
Debugging Tips

- Use trace statements:
  - Print out the values of variables.
  - Print out statements to show control flow.

- Automate:
  - Write your own tests!

- Simplify:
  - Comment out parts that aren’t a likely cause.
  - Remove components until you isolate the problem.
  - Writing modular code helps immensely.