Tutorial 5

- Overflow
- Structures
- Call stack and stack frames

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

- A variable of type `int` allocates 32 bits of memory.
- Able to represent negative and positive numbers, so roughly half of this range is negative and roughly half is positive.

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

INT_MIN	-2^{31}	-2147483648
INT_MAX	$2^{31}-1$	2147483647

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

Practice Problem 1: Checking for overflow

- Define the following C function:

```
// will_positive_overflow(a, b) takes non-negative
//   integers a and b and determines if (a+b) will
//   overflow.
// requires: a, b >= 0
```

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 2: A simple robot

Write a program which tracks the movement of Rob the robot.

This program reads in commands from I/O:

```
//   n INT   move the robot north INT
//   s INT     "   "     "   south INT
//   e INT     "   "     "   east  INT
//   w INT     "   "     "   west  INT
```

The following functions must be written, and are called from `main`:

- `move`: moves Rob a given distance in a given direction
- `print_location`: outputs the message `Rob is at (x,y)`

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:

```
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 0;
}
```

Example: Draw the call stack

```
1  int g(void){
2    int x = 2;
3    return x;
4  }
5
6  int f(int x){
7    int a = g();
8    return x * a;
9  }
10
11 int main(void){
12   int x = 4;
13   int a = 1;
14   ⇒ int y = f(x) + a;
15   return 0;
16 }
```

```
==========================
main:
  x: 4
  a: 1
  y: ?
  return address: OS
==========================
```

Example: Draw the call stack

```
1  int g(void){
2    int x = 2;
3    return x;
4  }
5
6  int f(int x){
7    ⇒ int a = g();
8    return x * a;
9  }
10
11 int main(void){
12   int x = 4;
13   int a = 1;
14   int y = f(x) + a;
15   return 0;
16 }
```

```
==========================
f:
  x: 4
  a: ?
  return address: main:14
==========================
main:
  x: 4
  a: 1
  y: ?
  return address: OS
==========================
```

Example: Draw the call stack

```
1  int g(void){
2    int x = 2;
3    ⇒ return x;
4  }
5
6  int f(int x){
7    int a = g();
8    return x * a;
9  }
10
11 int main(void){
12   int x = 4;
13   int a = 1;
14   int y = f(x) + a;
15   return 0;
16 }
```

```
==========================
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

```
1  int g(void){
2    int x = 2;
3    return x;
4  }
5
6  int f(int x){
7    int a = g();
8    ⇒ return x * a;
9  }
10
11 int main(void){
12   int x = 4;
13   int a = 1;
14   int y = f(x) + a;
15   return 0;
16 }
```

```
==========================
f:
  x: 4
  a: 2
  return address: main:14
==========================
main:
  x: 4
  a: 1
  y: ?
  return address: OS
==========================
```

Example: Draw the call stack

```
1  int g(void){
2    int x = 2;
3    return x;
4  }
5
6  int f(int x){
7    int a = g();
8    return x * a;
9  }
10
11 int main(void){
12   int x = 4;
13   int a = 1;
14   int y = f(x) + a;
15   ⇒ return 0;
16 }
```

```
==========================
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:

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

int f(struct posn bar) {
  return bar.x + 1;
}

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

Example: Structures on the call stack

```
1  struct posn {
2    int x;
3    int y;
4  };
5
6  int f(struct posn bar) {
7    return bar.x + 1;
8  }
9
10 int main(void) {
11   struct posn foo = {5, 6};
12   ⇒ int x = f(foo);
13   return 0;
14 }
```

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

Example: Structures on the call stack

```
1  struct posn {
2    int x;
3    int y;
4  };
5
6  int f(struct posn bar) {
7    ⇒ return bar.x + 1;
8  }
9
10 int main(void) {
11   struct posn foo = {5, 6};
12   int x = f(foo);
13   return 0;
14 }
```

```
==========================
f:
  bar:
      .x = 5
      .y = 6
  return address: main:12
==========================
main:
  foo:
      .x = 5
      .y = 6
  x: ?
  return address: OS
==========================
```

Example: Structures on the call stack

```
1  struct posn {
2    int x;
3    int y;
4  };
5
6  int f(struct posn bar) {
7    return bar.x + 1;
8  }
9
10 int main(void) {
11   struct posn foo = {5, 6};
12   int x = f(foo);
13   ⇒ return 0;
14 }
```

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