

# CSC 347 - Concepts of Programming Languages

## Undefined Behavior

Instructor: Stefan Mitsch



## Learning Objectives

- ❓ What should happen if the language does not specify the meaning of some construct?
- ❓ What should happen when `x` is holding 8 bits, `x=7` and then `x = x+1` ?
  - Identify undefined behavior in C



# Datatype Bounds

- Natural numbers are only partly useful as integer arithmetic semantics

$$\frac{}{\langle n, \xi \rangle \Downarrow \langle n, \xi \rangle} \text{ (Num)}$$

- Add integer bounds

$$\frac{-2^{63} \leq n \leq 2^{63} - 1}{\langle n, \xi \rangle \Downarrow \langle n, \xi \rangle} \text{ (Num)}$$

- What to do with overflow?

- Error: 
$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle \quad \neg(-2^{63} \leq v_1 + v_2 \leq 2^{63} - 1)}{\langle e_1 + e_2, \xi_0 \rangle \Downarrow \langle \text{error}, \xi_2 \rangle} \text{ (AddOutOfBounds)}$$
- Wrap: 
$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle \quad 2^{63} - 1 < v_1 + v_2}{\langle e_1 + e_2, \xi_0 \rangle \Downarrow \langle -2^{63} + ((v_1 + v_2) \bmod 2^{63}), \xi_2 \rangle} \text{ (AddOverflow), } \dots$$



## Under and Overflow

```
1 #include <stdio.h>
2
3 int isMinValue (int x) {
4     return (x-1) > x;
5 }
6 int main () {
7     int i = -2000000000;
8     while (!isMinValue(i))
9         i--;
10    printf ("Min value is %d\n", i);
11 }
```

```
1 $ gcc -O1 undefined.c && ./a.out
2 Min value is -2147483648
```

```
1 $ gcc -O2 undefined.c && ./a.out
2 ^C #infinite loop
3
```



## Order of Operations

- Recall [rule for sequential composition](#)
- Rule define an order of evaluating subexpressions

$$\frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle e_1 ; e_2, \xi_0 \rangle \Downarrow \langle v_2, \xi_2 \rangle} (;) \qquad \frac{\langle e_1, \xi_0 \rangle \Downarrow \langle v_1, \xi_1 \rangle \quad \langle e_2, \xi_1 \rangle \Downarrow \langle v_2, \xi_2 \rangle}{\langle e_1 + e_2, \xi_0 \rangle \Downarrow \langle v_1 + v_2, \xi_2 \rangle} (\text{Add})$$

- by chaining the stores
- $\xi_0$  is used in evaluating  $e_1$  and produces  $\xi_1$
- $\xi_1$  is used in evaluating  $e_2$  and produces  $\xi_2$



# Undefined Order of Operations

```
1 #include <stdio.h>
2 int count = 0;
3
4 int f() { return ++count; }
5
6 int g() { return ++count >= 2 ? 5 : 3; }
7
8 int h(int a, int b) { return a+b; }
9
10 int main() {
11     int x = h(f(), g());
12     printf ("%d\n", x);
13
14     int y = 3;
15     y = (y += 1) + (y = y*y);
16     printf ("%d\n", y);
17 }
```

```
1 $ clang -Wall undefined.c
2 undefined.c:13:21: warning:
3 unsequenced modification and
4 access to 'y' [-Wunsequenced]
5     y = (y += 1) + (y = y*y);
6                 ~~~~      ^
7 1 warning generated.
8 $ ./a.out
9 6
10 20
```

- `a=f(), b=g(), a+b`
- `y=y+1; y=y+y*y`



# Undefined Order of Operations

```
1 #include <stdio.h>
2 int count = 0;
3 int f() { return ++count; }
4
5 int g() { return ++count >= 2 ? 5 : 3; }
6
7 int h(int a, int b) { return a+b; }
8
9 int main() {
10     int x = h(f(), g());
11     printf ("%d\n", x);
12
13     int y = 3;
14     y = (y += 1) + (y = y*y);
15     printf ("%d\n", y);
16 }
```

```
1 $ gcc -Wall -O3 undefined.c
2 undefined.c:13:21: warning:
3 unsequenced modification and
4 access to 'y' [-Wunsequenced]
5     y = (y += 1) + (y = y*y);
6         ^
7 $ ./a.out
8 5
9 32
```

- `b=g(), a=f(), a+b`
- `y=y+1; y=y*y; y=y+y;`



# Compiler Optimizations

- For undefined executions, the compiler can do what it likes
- This can lead to some surprising compiler optimizations
- [C null pointer optimization 1](#)



## Summary

- Undefined behavior resolved in the compiler (compiler decides what is the meaning of undefined programs)
- Undefined behavior often presents security risks
- [More examples of undefined behavior](#)
- [More undefined behavior](#)
- Formal semantics can be analyzed for contradictions and completeness