HIGH-LEVEL PROGRAMMING I

Order of operand evaluation

by Prasanna Ghali

References

- C's side effects are explained here
- C's sequence points are explained here
- Order of operand evaluation are explained here

What are Side Effects?

- *Side effect* is change in state of program's execution state and is achieved by modifying C/C++ entities with memory storage
- Following *side effect operators* modify associated operand
  - Assignment, compound assignment, prefix increment and decrement, and postfix increment and decrement
  - `c = a++ * --b; // three changes`

Side Effects: Examples (1/2)

Expression	Side effect? (Y or N)
`++n`	
`-x`	
`!b`	
`m = n`	
`m += n`	
`m + n`	
`m - n`	
`printf("%d %d", m, n)`	

Side Effects: Examples (2/2)

Expression	Side effect (Y or N)
`++n`	yes
`-x`	no
`!b`	no
`m = n`	yes
`m += n`	yes
`m + n`	no
`m - n`	no
`printf("%d %d", m, n)`	yes (**stdout** is modified)

Sequence Points (1/5)

- Evaluation of expression may produce *side effects*
- What are values of `a` and `b` after evaluation of following statement?

```
int a = 10, b;
b = a++ + ++a;
```

- Undefined behavior – `a` could be `11` or `12` and `b` could be `21` or `22`
- Why does statement generate undefined behavior?

Sequence Points (2/5)

- At specific points during execution, known as ***sequence points***, all side effects of previous evaluations are complete, and no side effects of subsequent evaluations have yet taken place
- In simpler terms: sequence point is location in program text where the previous dust has settled (that is, all operations have been executed) before new operations are executed

Sequence Points (3/5)

- We should be aware of following sequence points
  - Between evaluations of 1st and 2nd operands for logical AND, logical OR, and comma operators
  - Between evaluations of 1st operand of `?:` operator and whichever of 2nd and 3rd operands is evaluated
  - Controlling expression of `if` and `switch` statement
  - Loop condition of `while` or `do` statement
  - Each of 3 expressions of `for` statement

Sequence Points (4/5)

- What are values of **a** and **b** after evaluation of following statement?

```
int a = 10, b;
b = a++ + ++a;
```

- Undefined behavior – **a** could be 11 or 12 and **b** could be 21 or 22
- Why? There are two problems:
  - Increment/decrement operators can only guarantee increment/decrement will be complete *after* next sequence point
  - Order of operand evaluation is unspecified – can't say whether **a++** or **++a** is evaluated first

Sequence Points (5/5)

- Another example of undefined behavior:

```
int a = 4, b;
// a modified twice between sequence points
b = a * a++; // dangerous code
```

- Depending on whether left or right operand of operator `*` is evaluated first, `b` could have value of either `16` or `20`
- GCC will issue a warning with `-Wall` option:

```
warning: operation on 'a' may be undefined
```

Order of Evaluation

- Consider expressions that use multiple operators and are composed of multiple *subexpressions*
  - `x = foo() + boo()`
  - `r = w * x + y * z`
  - `r = x++ + ++x`
  - `x = coo(++x) + doo(x)`
- In general, assume order of operand evaluation is *unspecified*

Order of Evaluation: Example 1

- Undefined behavior

```
void func(int x, int y);

void foo(int i) {
  func(i++, i);
}
```

- Compliant code

```
void func(int x, int y);

void foo(int i) {
  func(i, i+1);
  i++;
}
```

Order of Evaluation: Example 2

- Undefined behavior

```
int i = 5;
i = i++;
```

  - Subexpression **`i++`** causes side effect while **`i`** is also referenced elsewhere in same expression
  - No way to know if reference will happen before or after side effect

- Compliant code

```
int i = 5;
i = i + 1;
```

Order of Evaluation: Example 3

- What is printed to standard output?
  - ***Order of operand evaluation is unspecified!!!***

```
int boo(void) {
  printf("I'm boo\n");
  return 10;
}

int foo(void) {
  printf("I'm foo\n");
  return 20;
}

int main(void) {
  printf("%d and %d\n", boo(), foo());
  return 0;
}
```

Could be either:

```
I'm boo
I'm foo
10 and 20
```

OR

```
I'm foo
I'm boo
10 and 20
```

Order of Evaluation: Example 4

- Taking advantage of order of operand evaluation
  - There's special "short-circuiting" behavior for logical AND (and also for logical OR) operator
  - If left operand determines value and result for expression, then right operand is not evaluated

```
/*
continue to read characters from standard input until
there are no more characters to be read or until a
newline is encountered
*/
while ((ch = getchar()) != EOF && ch != '\n') {
  // some useful code here
}
```

Summary

- What is side effect?
- What is undefined behavior and when does it arise?
- What is sequence point?
- What is order of operand evaluation mean?
- Which operators specify order of operand evaluation
- How to write compliant code