Control Flow

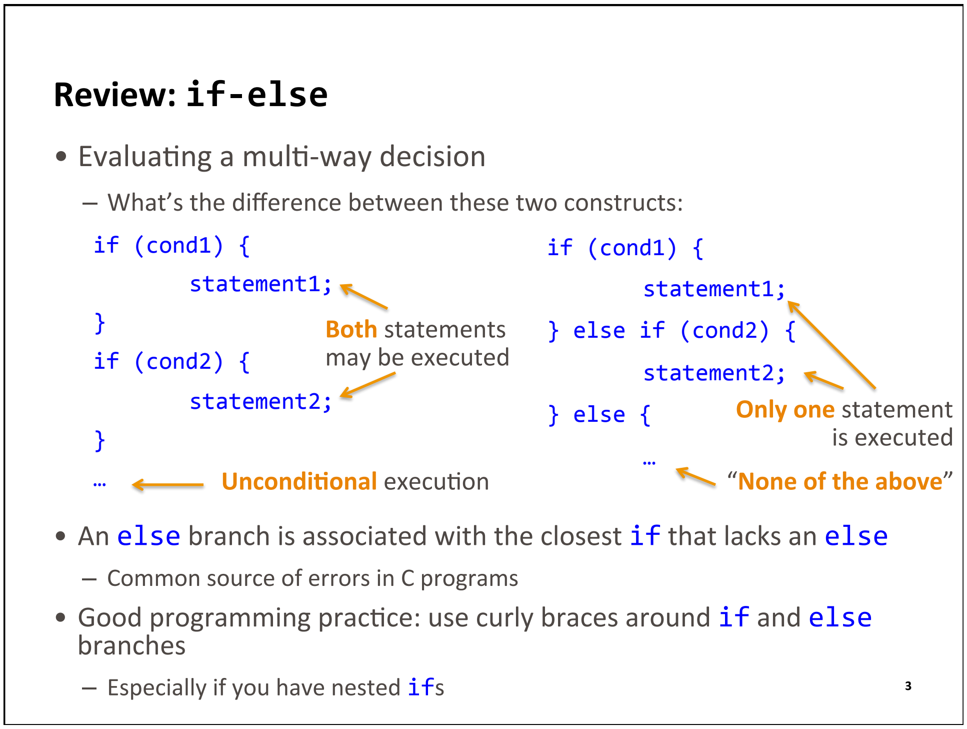
ENEE 140

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

http://ter.ps/enee140

Today's Lecture

- Where we've been
  - Scalar data types (`int, long, float, double, char`)
  - Basic control flow (`while` and `if`)
  - Functions
  - Random number generation
  - Arrays and strings
  - Variable scope
  - Header and source files
- Where we're going today
  - Other control flow statements
  - Project 2 Q&A
- Where we're going next
  - File Input/Output



Review: if-else

- Evaluating a multi-way decision
  - What's the difference between these two constructs:

```
if (cond1) {
        statement1;
}
if (cond2) {
        statement2;
}
…
```

Both statements may be executed

Unconditional execution

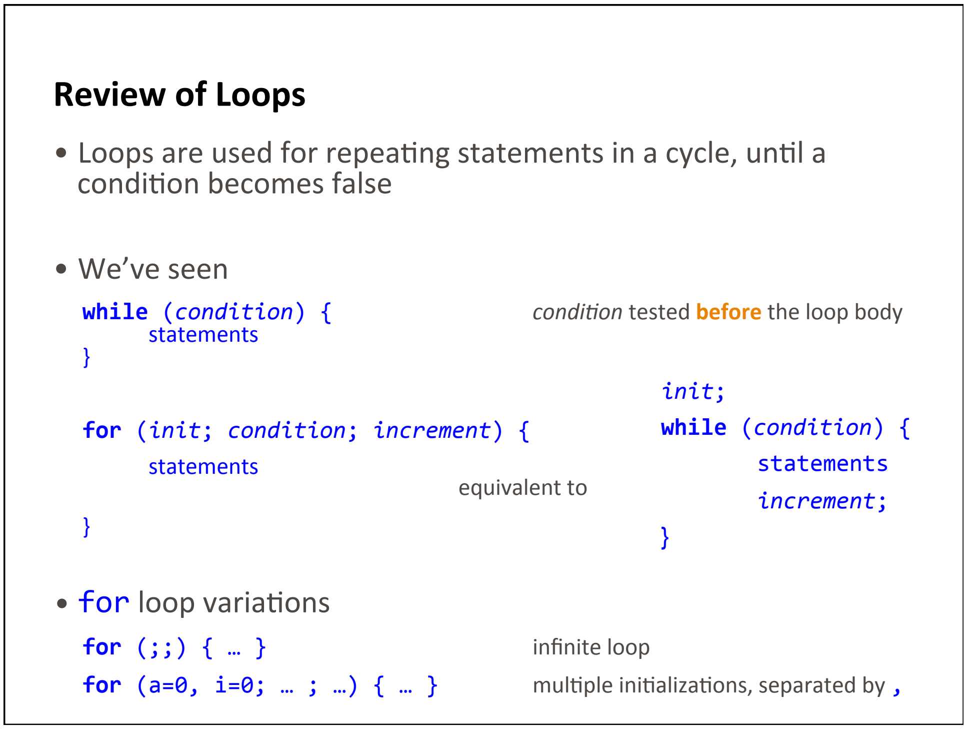
```
if (cond1) {
        statement1;
} else if (cond2) {
        statement2;
} else {
        …
```

Only one statement is executed

"**None of the above**"

- An `else` branch is associated with the closest `if` that lacks an `else`
  - Common source of errors in C programs
- Good programming practice: use curly braces around `if` and `else` branches
  - Especially if you have nested `if`s

3



Review of Loops

- Loops are used for repeating statements in a cycle, until a condition becomes false

- We've seen

```
while (condition) {
        statements
}
```

condition tested **before** the loop body

```
for (init; condition; increment) {
        statements
}
```

equivalent to

```
init;
while (condition) {
        statements
        increment;
}
```

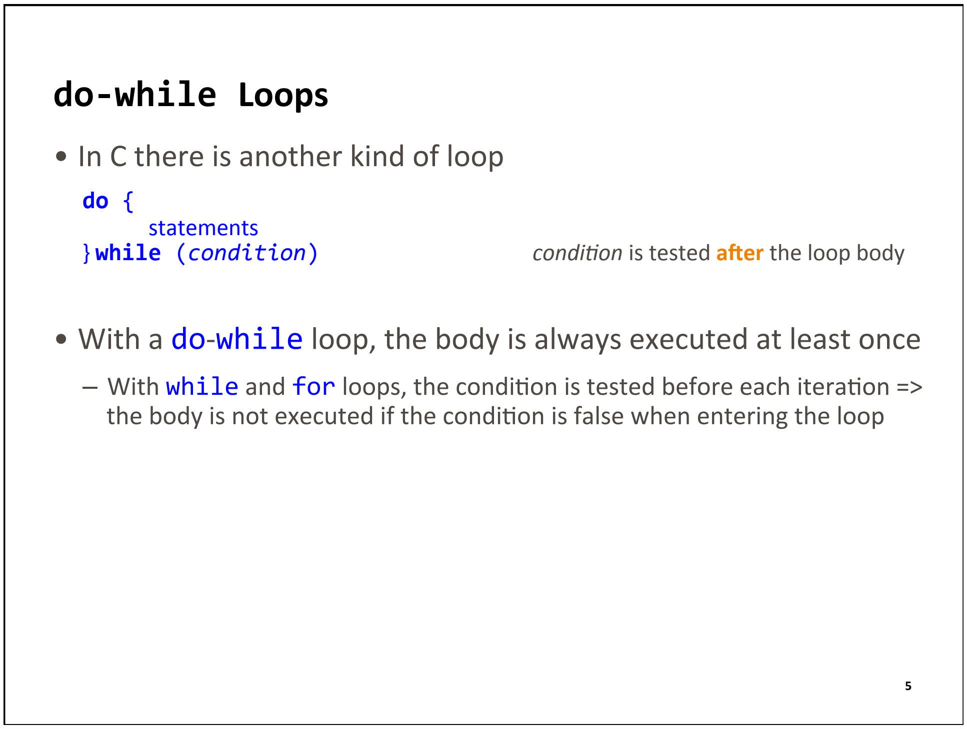
- `for` loop variations

```
for (;;) { … }
```

infinite loop

```
for (a=0, i=0; … ; …) { … }
```

multiple initializations, separated by `,`



do-while Loops

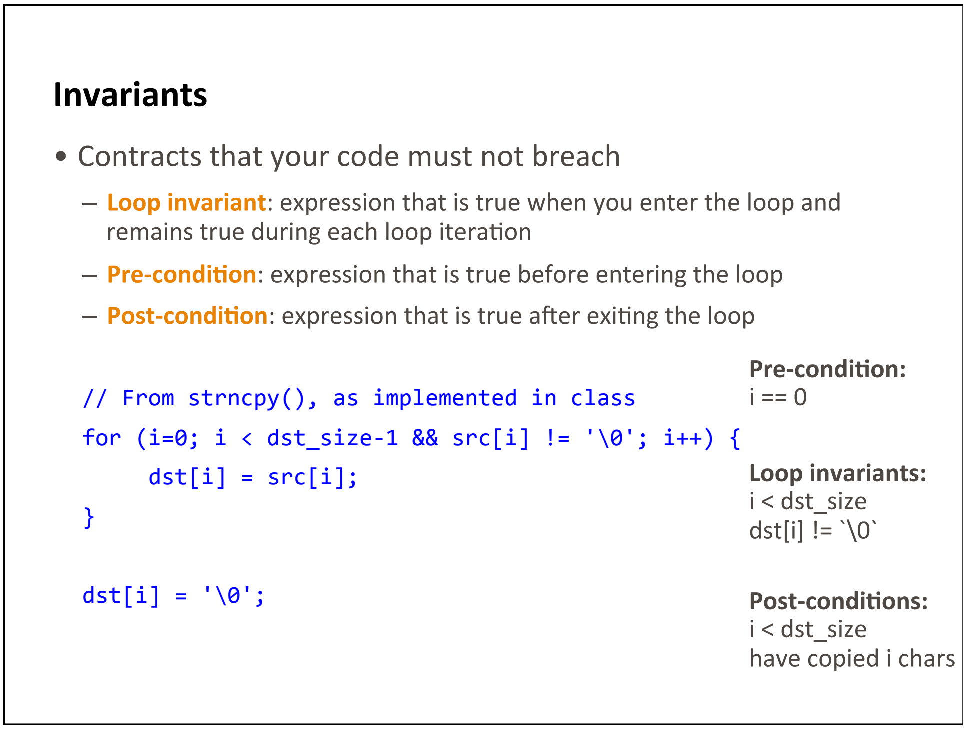
- In C there is another kind of loop

```
do {
     statements
}while (condition)
```

condition is tested **after** the loop body

- With a do-while loop, the body is always executed at least once
  - With while and for loops, the condition is tested before each iteration => the body is not executed if the condition is false when entering the loop

5



Invariants

- Contracts that your code must not breach
  - **Loop invariant**: expression that is true when you enter the loop and remains true during each loop iteration
  - **Pre-condition**: expression that is true before entering the loop
  - **Post-condition**: expression that is true after exiting the loop

```
// From strncpy(), as implemented in class
for (i=0; i < dst_size-1 && src[i] != '\0'; i++) {
     dst[i] = src[i];
}

dst[i] = '\0';
```

Pre-condition:
i == 0

Loop invariants:
i < dst_size
dst[i] != `\0`

Post-conditions:
i < dst_size
have copied i chars

Invariants and Defensive Programming

- Asserting invariants

```
#include <assert.h>
assert(condition);
```

exits the program if *condition* is false

  - Use `assert()` liberally
  - Assertions allow you to diagnose mistakes in your program
  - They also reveal your program’s invariants to other programmers who review your code

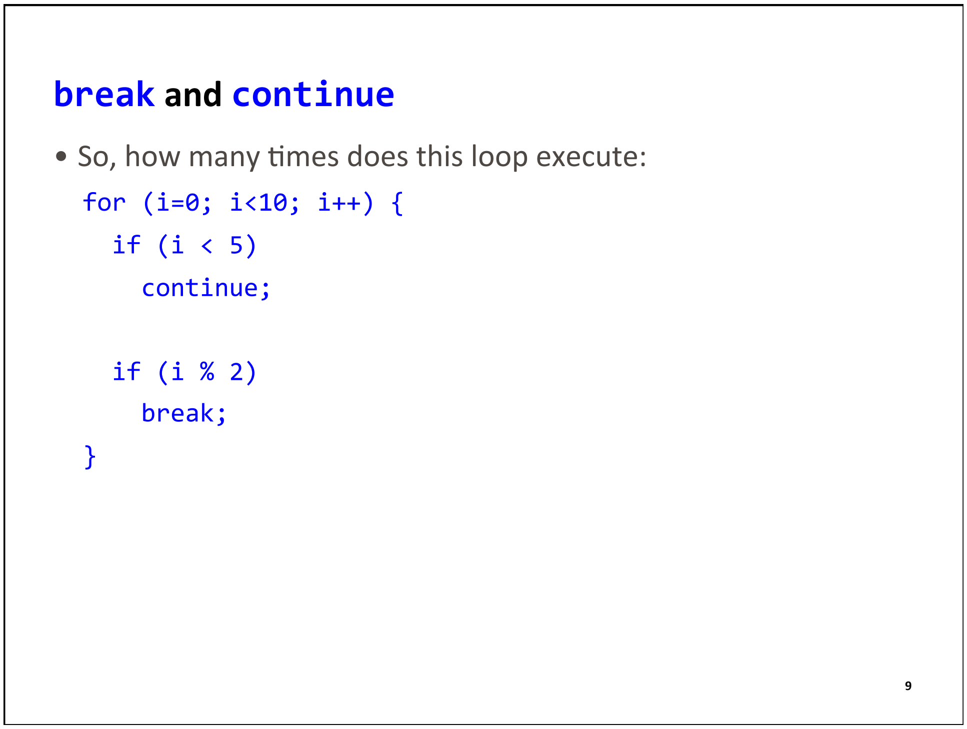
```
for (i=0; i < dst_size-1 && src[i] != '\0'; i++) {
     dst[i] = src[i];
     assert (dst[i] != `\0`);
}

assert (i < dst_size);
dst[i] = '\0’;
```

Early Loop Exit

- `break` and `continue`
  - `break` causes the innermost loop or `switch` statement (described next) to exit
  - `continue` skips over the remaining statements in the loop body and starts the next iteration

```
for (x=1; x<10; x++) {
     if (x == 5)
            break;                // exit the loop
     …
}
…
```

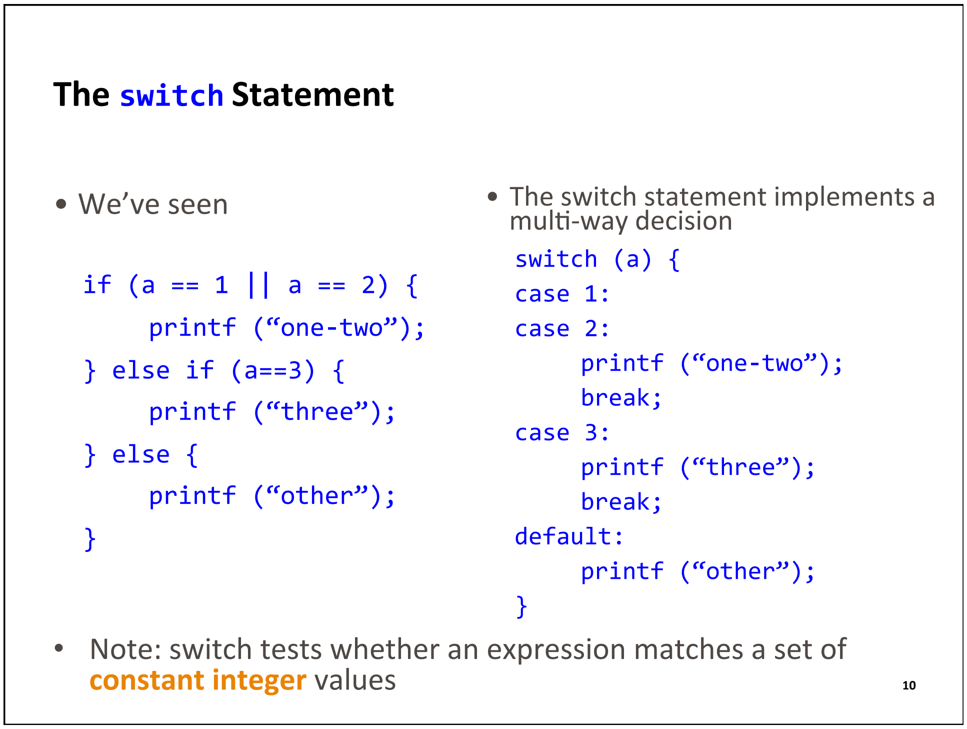
- `goto label`
  - Jumps to a label that can be placed anywhere in the code
  - `goto` makes it difficult to reason about invariants => DO NOT USE!!
  - The only accepted modern usage of `goto` is to break out of nested loops



break and continue

- So, how many times does this loop execute:

```
for (i=0; i<10; i++) {
  if (i < 5)
    continue;

  if (i % 2)
    break;
}
```



The switch Statement

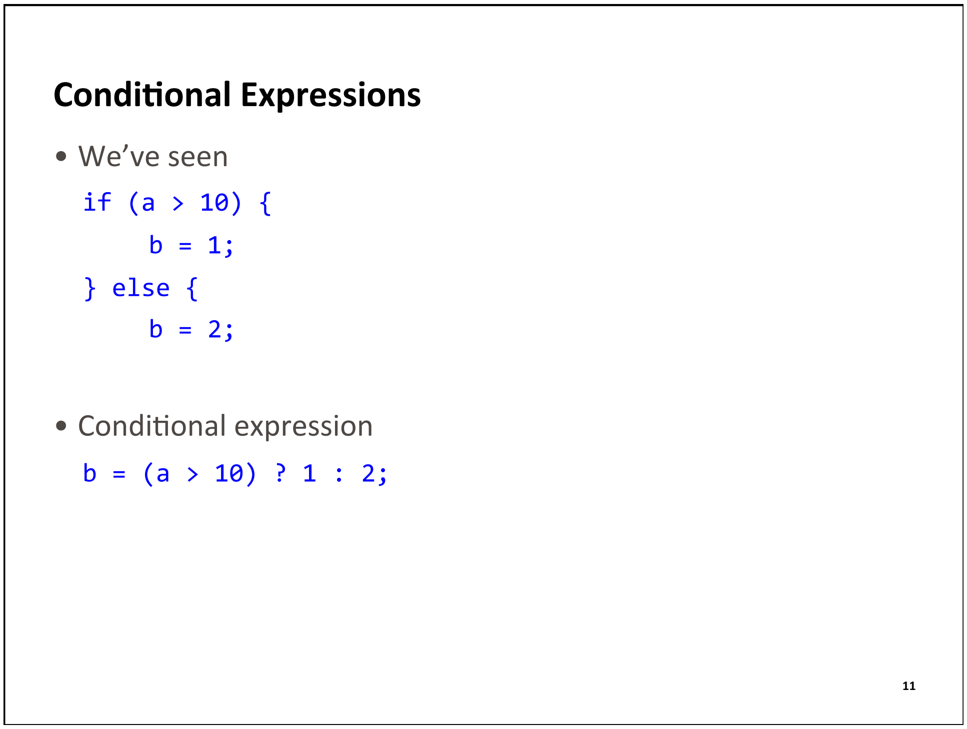
- We've seen

```
if (a == 1 || a == 2) {
     printf (“one-two”);
} else if (a==3) {
     printf (“three”);
} else {
     printf (“other”);
}
```

- The switch statement implements a multi-way decision

```
switch (a) {
case 1:
case 2:
     printf (“one-two”);
     break;
case 3:
     printf (“three”);
     break;
default:
     printf (“other”);
}
```

- Note: switch tests whether an expression matches a set of **constant integer** values



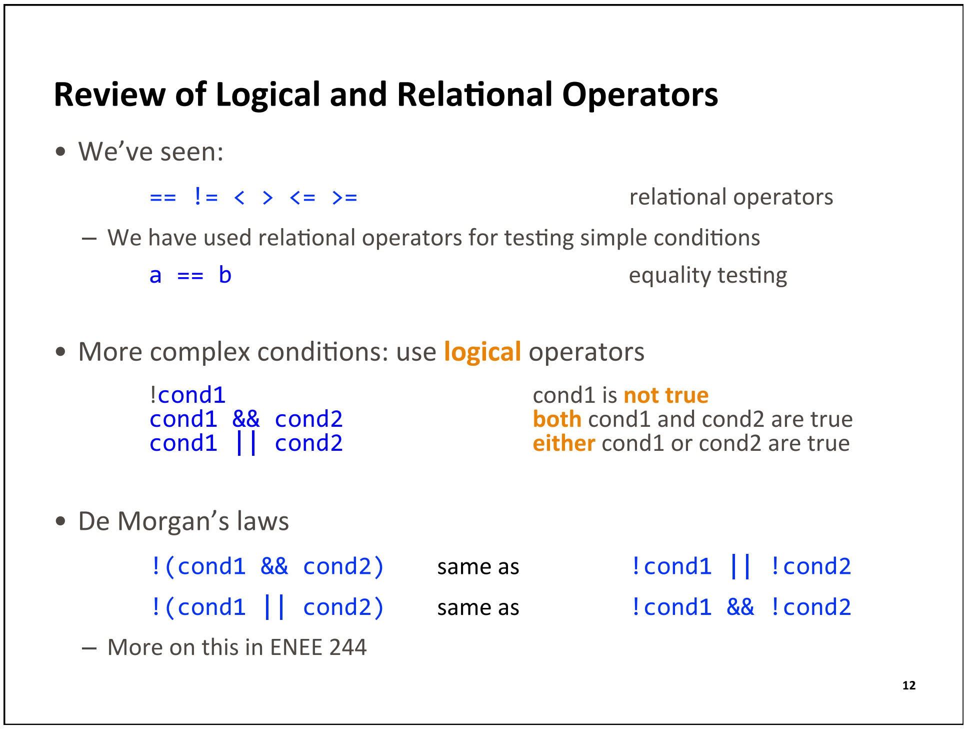
Conditional Expressions

- We've seen

```
if (a > 10) {
     b = 1;
} else {
     b = 2;
```

- Conditional expression

```
b = (a > 10) ? 1 : 2;
```



Review of Logical and Relational Operators

- We've seen:

`== != < > <= >=`	relational operators

  - We have used relational operators for testing simple conditions

`a == b`	equality testing

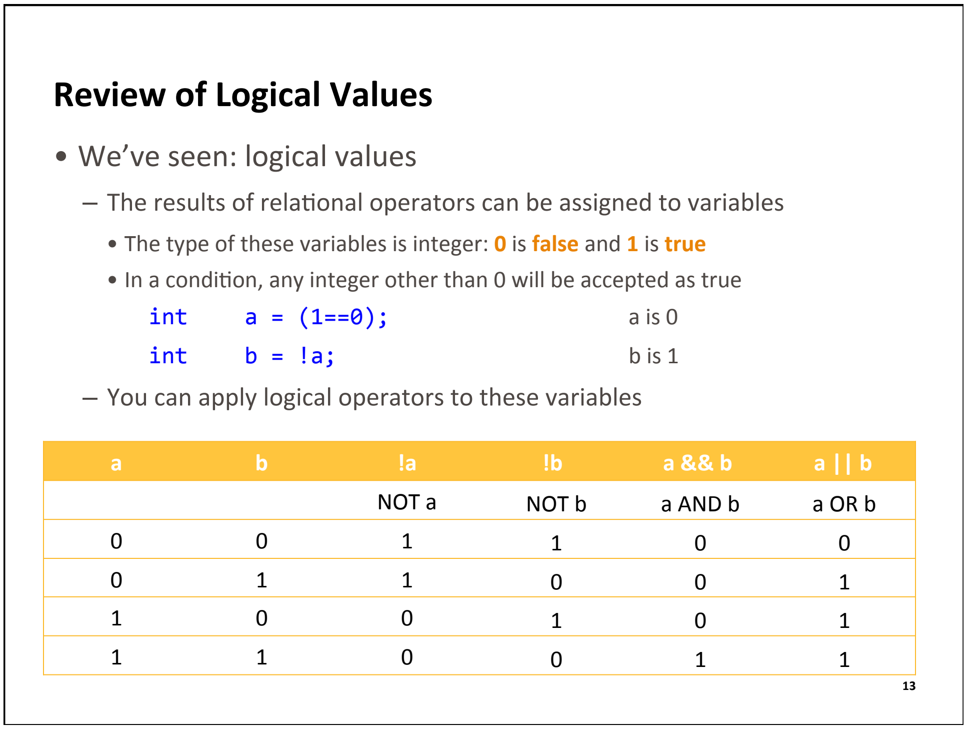
- More complex conditions: use **logical** operators

`!cond1`	cond1 is **not true**
`cond1 && cond2`	**both** cond1 and cond2 are true
`cond1 || cond2`	**either** cond1 or cond2 are true

- De Morgan's laws

`!(cond1 && cond2)`	same as	`!cond1 || !cond2`
`!(cond1 || cond2)`	same as	`!cond1 && !cond2`

  - More on this in ENEE 244



Review of Logical Values

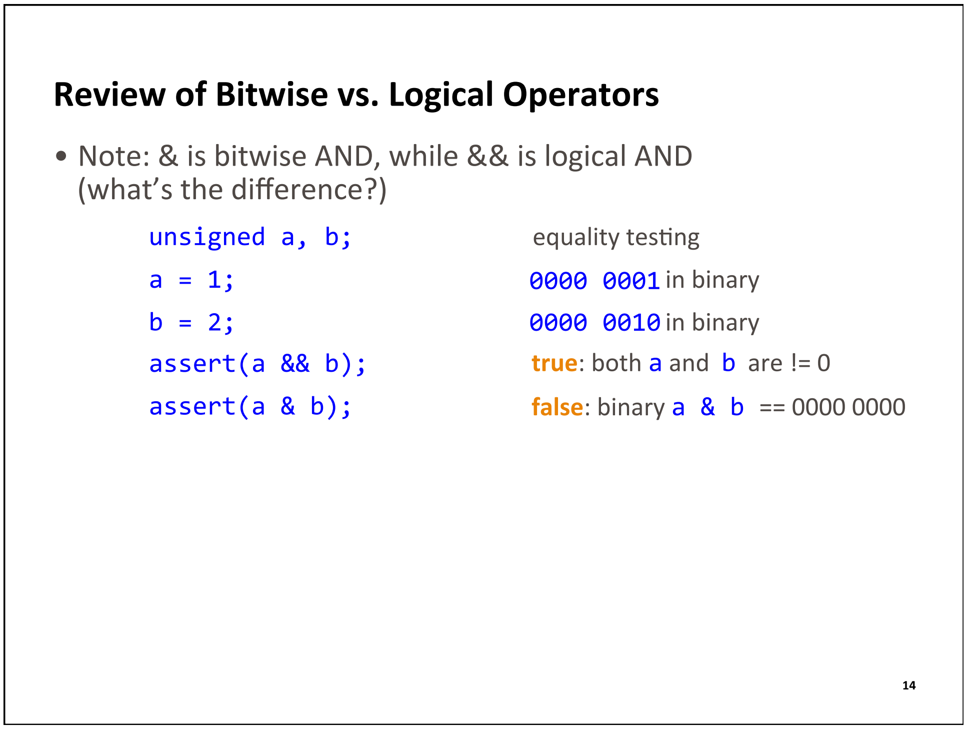
- We’ve seen: logical values
  - The results of relational operators can be assigned to variables
    - The type of these variables is integer: **0** is **false** and **1** is **true**
    - In a condition, any integer other than 0 will be accepted as true

```
int    a = (1==0);            a is 0
int    b = !a;                b is 1
```

  - You can apply logical operators to these variables

a	b	!a	!b	a && b	a || b
		NOT a	NOT b	a AND b	a OR b
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

13

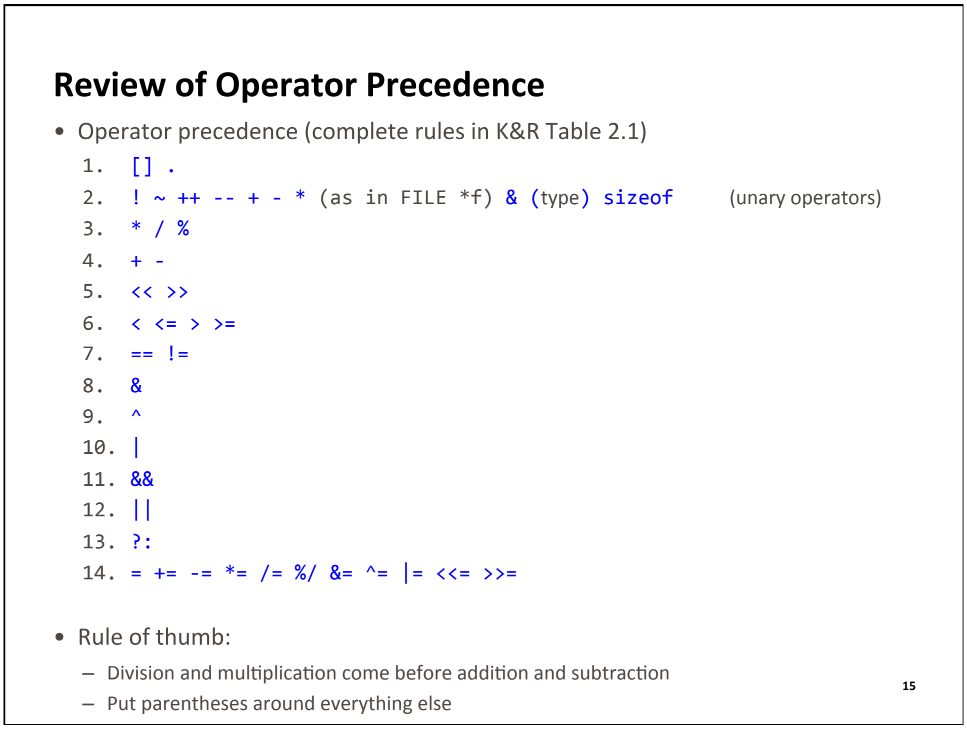


Review of Bitwise vs. Logical Operators

- Note: & is bitwise AND, while && is logical AND (what’s the difference?)

```
unsigned a, b;              equality testing
a = 1;                      0000 0001 in binary
b = 2;                      0000 0010 in binary
assert(a && b);             true: both a and b are != 0
assert(a & b);              false: binary a & b == 0000 0000
```

14

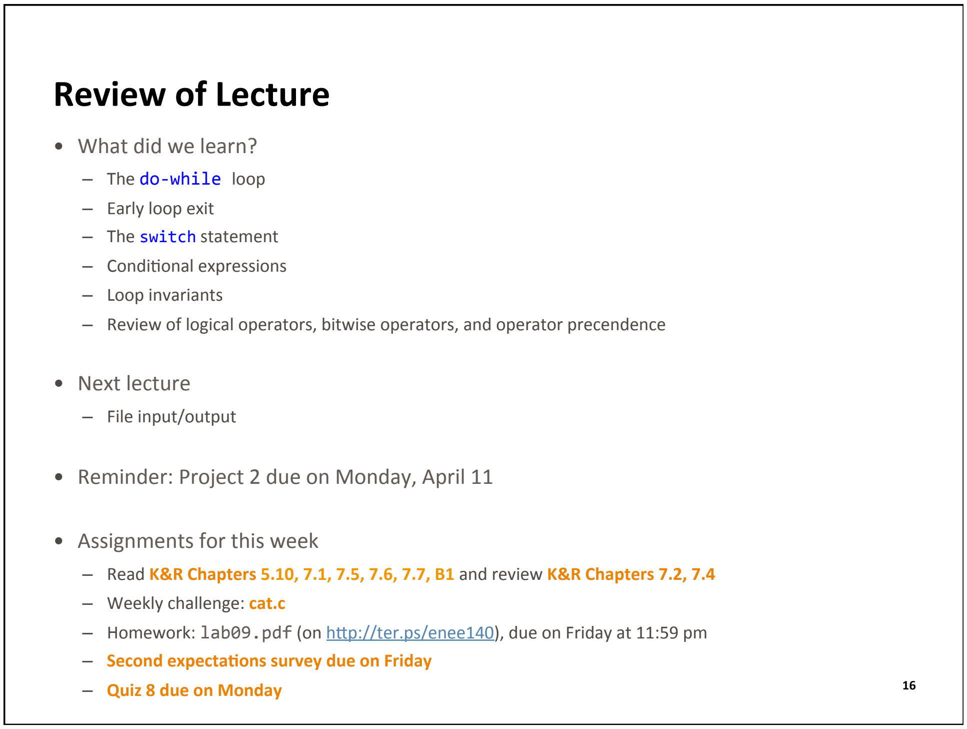


Review of Operator Precedence

- Operator precedence (complete rules in K&R Table 2.1)

```
1.  [] .
2.  ! ~ ++ -- + - * (as in FILE *f) & (type) sizeof      (unary operators)
3.  * / %
4.  + -
5.  << >>
6.  < <= > >=
7.  == !=
8.  &
9.  ^
10. |
11. &&
12. ||
13. ?:
14. = += -= *= /= %/ &= ^= |= <<= >>=
```

- Rule of thumb:
  - Division and multiplication come before addition and subtraction
  - Put parentheses around everything else



Review of Lecture

- What did we learn?
  - The `do-while` loop
  - Early loop exit
  - The `switch` statement
  - Conditional expressions
  - Loop invariants
  - Review of logical operators, bitwise operators, and operator precendence

- Next lecture
  - File input/output

- Reminder: Project 2 due on Monday, April 11

- Assignments for this week
  - Read **K&R Chapters 5.10, 7.1, 7.5, 7.6, 7.7, B1** and review **K&R Chapters 7.2, 7.4**
  - Weekly challenge: **cat.c**
  - Homework: `lab09.pdf` (on http://ter.ps/enee140), due on Friday at 11:59 pm
  - **Second expectations survey due on Friday**
  - **Quiz 8 due on Monday**