Review of CMSC 11

What do you remember?

Indentation Rules (1)

- Used to avoid misreading program code
- Should be kept in mind especially when nesting statements
- Not really required by C syntax rules but are generally followed to provide clearer code

Indentation Rules (2)

Bad practice of indentation

```
int example(int a, int b)
{
int x;
      if(a<b)
      x=a;
      else
            x=b;
            x*=.5f;
      return x;
}
```

Good practice of indentation

```
int example(int a, int b)
{
      int x;
      if(a<b)
            x=a;
      else
            x=b;
      x*=.5f;
      return x;
}
```

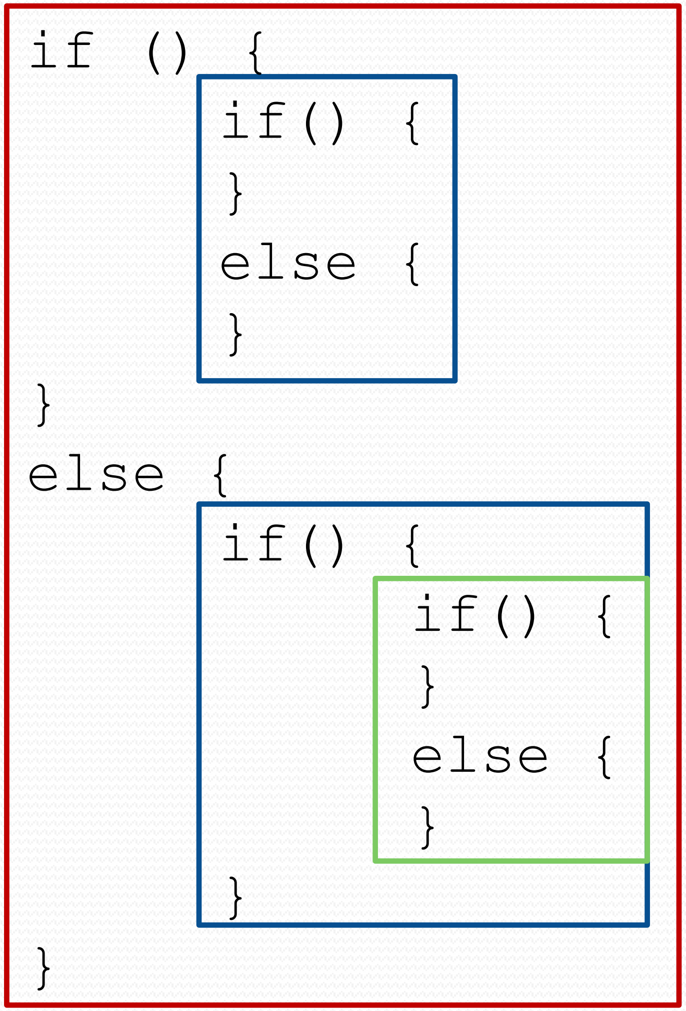
Indentation Rules (3)

1. Indent statements that are dependent on previous statements.
   - For example, the statements in an if-clause are dependent on the if statement, therefore these statements should be indented

```
if (a < b) {
    x = a;
} else {
    x = b;
}
```

Indentation Rules (4)

2. Align else statements with corresponding if statements.



```
if () {
    if() {
    }
    else {
    }
}
else {
    if() {
        if() {
        }
        else {
        }
    }
}
```

Indentation Rules (5)

3. Place the opening brace of a code block on a separate line and indent the statements in the code block relative to the opening brace.

```
if (...)
{
     //code block
}
```

Indentation Rules (6)

4. Place the closing brace of a code block on a separate line and align in with its corresponding opening brace. Mark the end of a code block with a comment.

```
if (...)
{
     //code block for if
} //end if
else
{
     //code block for else
} //end else
```

Indentation Rules (7)

5. Align all code dependent on the same (previous) statement on the same level.

```
if (...) {
     x = y;
     y = y + 10;
     ...
}
```

6. Further indent nested statements according to the previous rules.

Indentation Rules (8)

7. Surround operators with a whitespace.
8. Code only one definition or statement on a single line.
9. Make comments meaningful at the block level. Comments should not parrot the code.

* Source: Computer Science: A Structured Programming Approach Using C by Behrouz A. Forouzan & Richard F. Gilberg

Negative Logic

- Any expression that begins with a NOT (!) or have a NOT within
- Can be difficult to interpret
- Complement the negative logical expression

Original Statement	Complemented Statement
`if (x <= 0)`	`if (x > 0)`
`if (x != 5)`	`if (x == 5)`
`if (!(x <= 0 || !flag))`	`if (x > 0 && flag)`

Rules for Selection Statements

1. Code positive statements whenever possible.
2. Code the normal/expected condition first.
3. Code the most probable conditions first.
   - Important, especially in multi-way selection
   - The sooner a condition is met the sooner the program can skip the conditions of other cases, and the more efficient the program

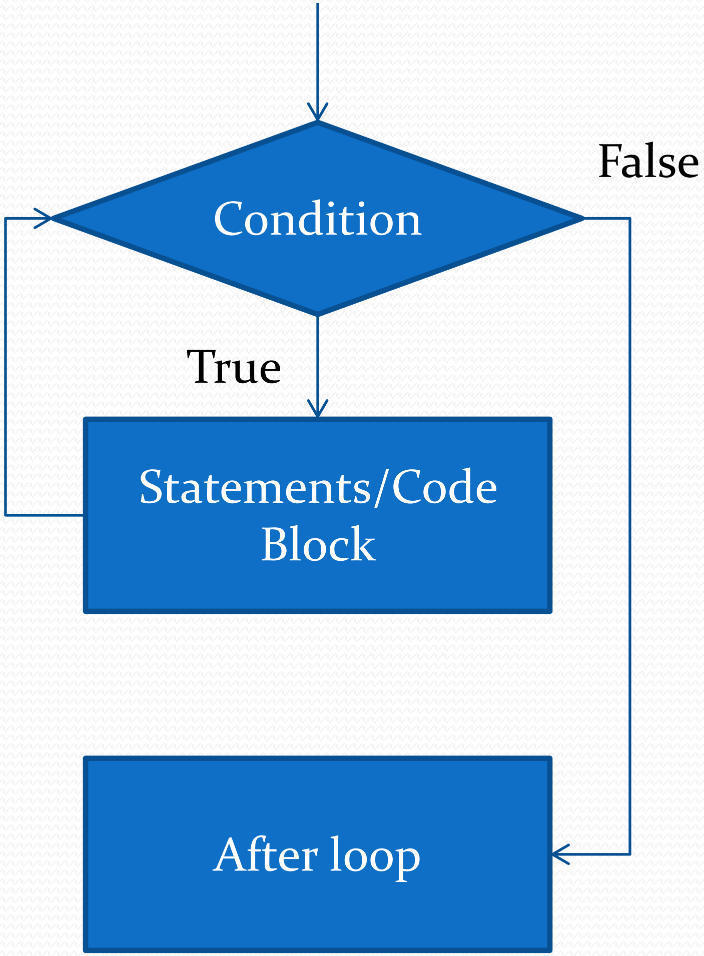
Iterative Statements

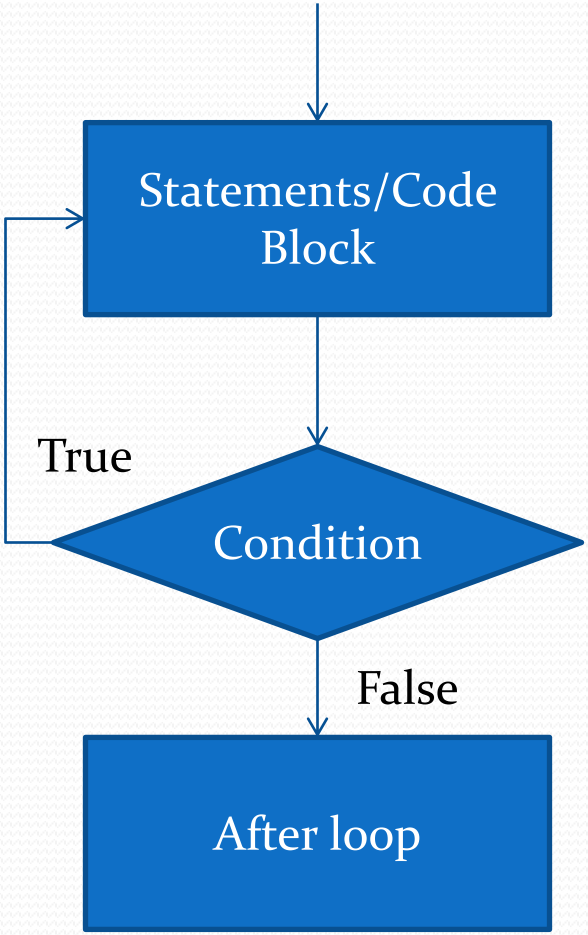
Loops and other things.

Loops

- A statement or group of statements are executed repeatedly
- To end repetition, use a loop control expression
- End the loop when the loop control expression evaluates to false
- When to test the loop control expression?
  - Pretest loop – test loop control expression before
  - Posttest loop – test loop control expression after; statements/code block executed at least once

Flowchart for Pretest & Posttest Loops





Processes in a Loop

1. Initialization
   - Process that needs to be done before execution of the loop body
   - Set values of key variables used within the loop
2. Condition (expressed as the loop control expression)
3. Increment/Update
   - Process that changes the truth value of the loop control expression over time to false
   - Must be present to avoid an infinite loop

Kinds of Loops

1. Event-Controlled
   - Loop termination dependent on a certain event
   - Once the event happens, the loop terminates
2. Counter-Controlled
   - Used when number of times to repeat is known beforehand
   - Initialize, test, and update the counter
   - Update can be increment (counting up) or decrement (counting down)
   - Number of times does not need to be a constant

Loop Statements in C

- `while`
- `for`
- `do-while`

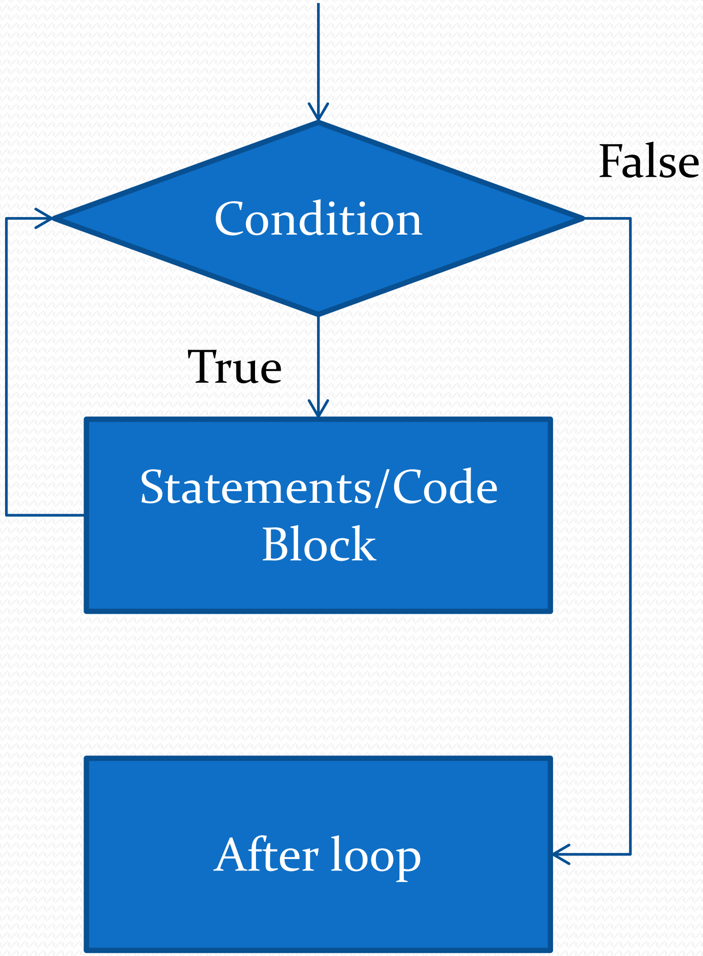
`while` Loop

- Pre-test loop
- Terminates when loop control expression is FALSE, continues otherwise
- Usually used for event-controlled loops
- Syntax:

```
while (loop control expression)
{
    statement;
}
```

- As with if-else statements, the statement under the loop can be replaced by a code block

Flowchart for `while` loop



`for` loop (1)

- Pretest loop
- Uses three expressions: initialization statements, limit-text expression, and updating expression
- Used mostly as a counter-controlled loop but can also be used as an event-controlled loop
- Syntax:

```
for (initialization; condition; update)
{
    statement;
}
```

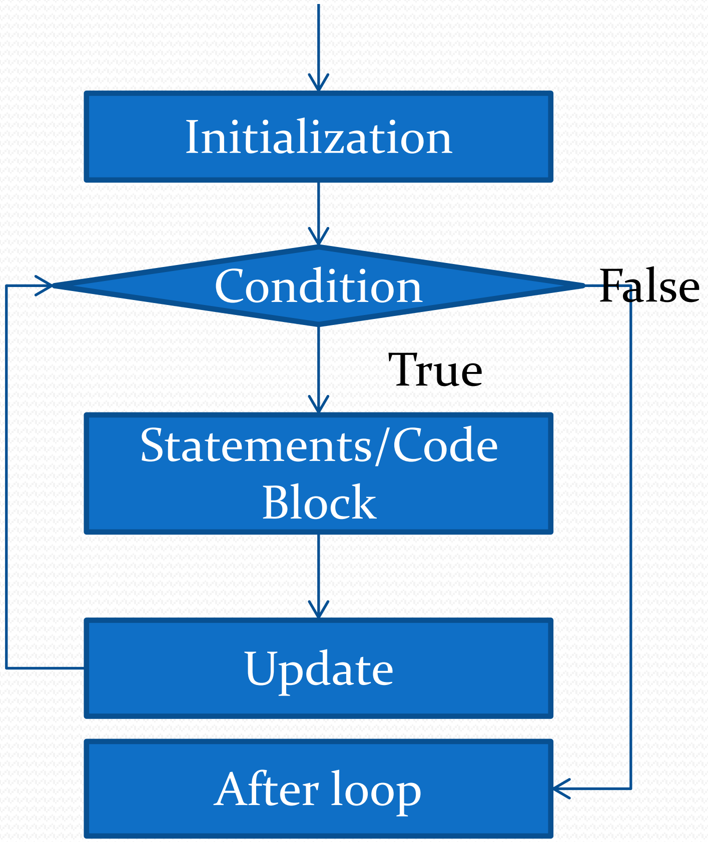
- Statement can also be a code block

for loop (2)

- Any of the three expressions may be null and controlled within the loop
- For example, what does the following statement result in?

```
for( ; ; )
{
        printf(“Hello world!”);
}
```

Flowchart for `for` loop


Initialization

Condition

False

True

Statements/Code Block

Update

After loop

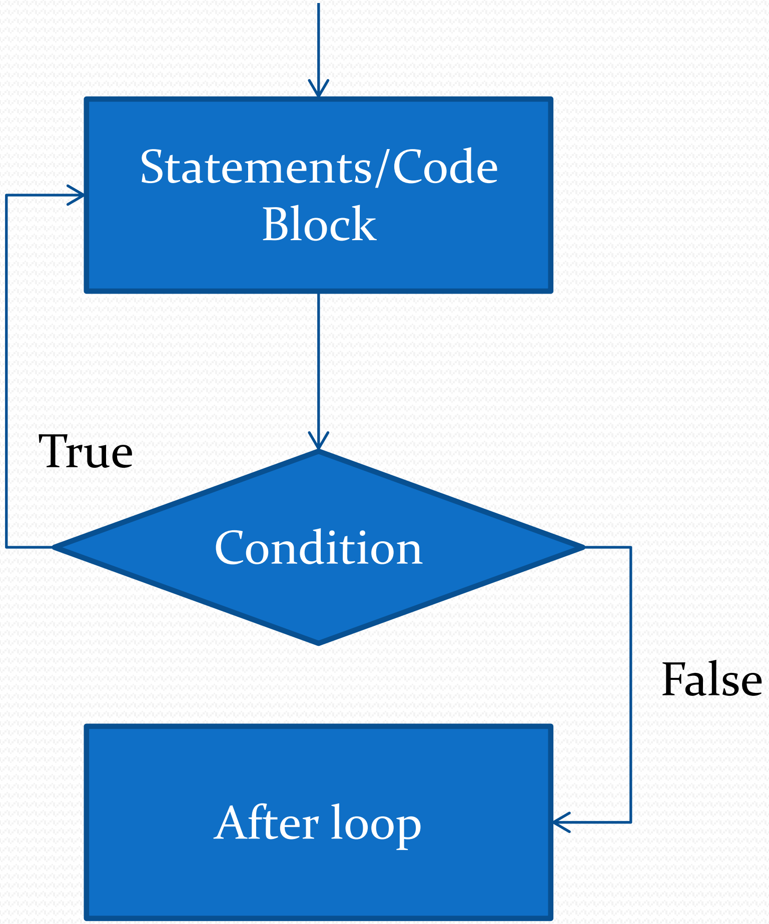
`do-while` loop

- Post-test loop
- Also uses a loop control expression but tests it after the execution of the loop body
- Syntax:

```
    do {
        //statements
    } while (loop control expression);
```

- Note that a do-while statement **ends with a semicolon (;)**.

Flowchart for `do-while` loop



Comma Expressions

- Complex expression
- Two expressions separated by a comma
- Most often used in `for` statements
- Evaluated left to right
- Example:

```
for(sum=0, i=1; i<=20; i++) {
        //statements
}
```