Week 3 - Friday

CS222

Last time

- What did we talk about last time?
- ASCII table
- **`printf()`** format strings
- Bitwise operators

Questions?

Project 1

Quotes

Unix is simple. It just takes a genius to understand its simplicity.

Dennis Ritchie

Precedence

- Operators in every programming language have precedence
- Some of them are evaluated before others
  - Just like order of operations in math
- ***** and **/** have higher precedence than **+** and **–**
  - **=** has a very lowest precedence
- I don't expect you to memorize them all, **but**
  - Know where to look them up
  - Don't write confusing code

Precedence table

Type	Operators	Associativity
Primary Expression	`()` `[]` `.` `->` *expr*`++` *expr*`--`	Left to right
Unary	`*` `&` `+` `-` `!` `~` `++`*expr* `--`*expr* `(`*typecast*`)` `sizeof`	Right to left
Binary	`*` `/` `%`	Left to right
	`+` `-`	
	`>>` `<<`	
	`<` `>` `<=` `>=`	
	`==` `!=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary	`?:`	Right to left
Assignment	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	Right to left
Comma	`,`	Left to right

Insane precedence example

- What happens here?
  - `x++ >> 5 == 4 % 12 & 3`
- It's also worth noting that precedence doesn't tell the whole story
- What about multiple assignments in a single line of code?
- C doesn't give you guarantees about what happens when
- The following could have different results on different compilers:

```
printf("%d %d", x++, (x + 5));
a[x] = x++;
x = x++;
```

Control flow

- Sequences of statements surrounded by braces are treated like a single statement with no value
  - Braces can be thrown in whenever you want
  - We used to say that "braces were optional" for one-line blocks, but this is the more accurate way to look at it
- An expression can always become a statement

```
int a = 150;
a; //legal in C, illegal in Java
```

Selection

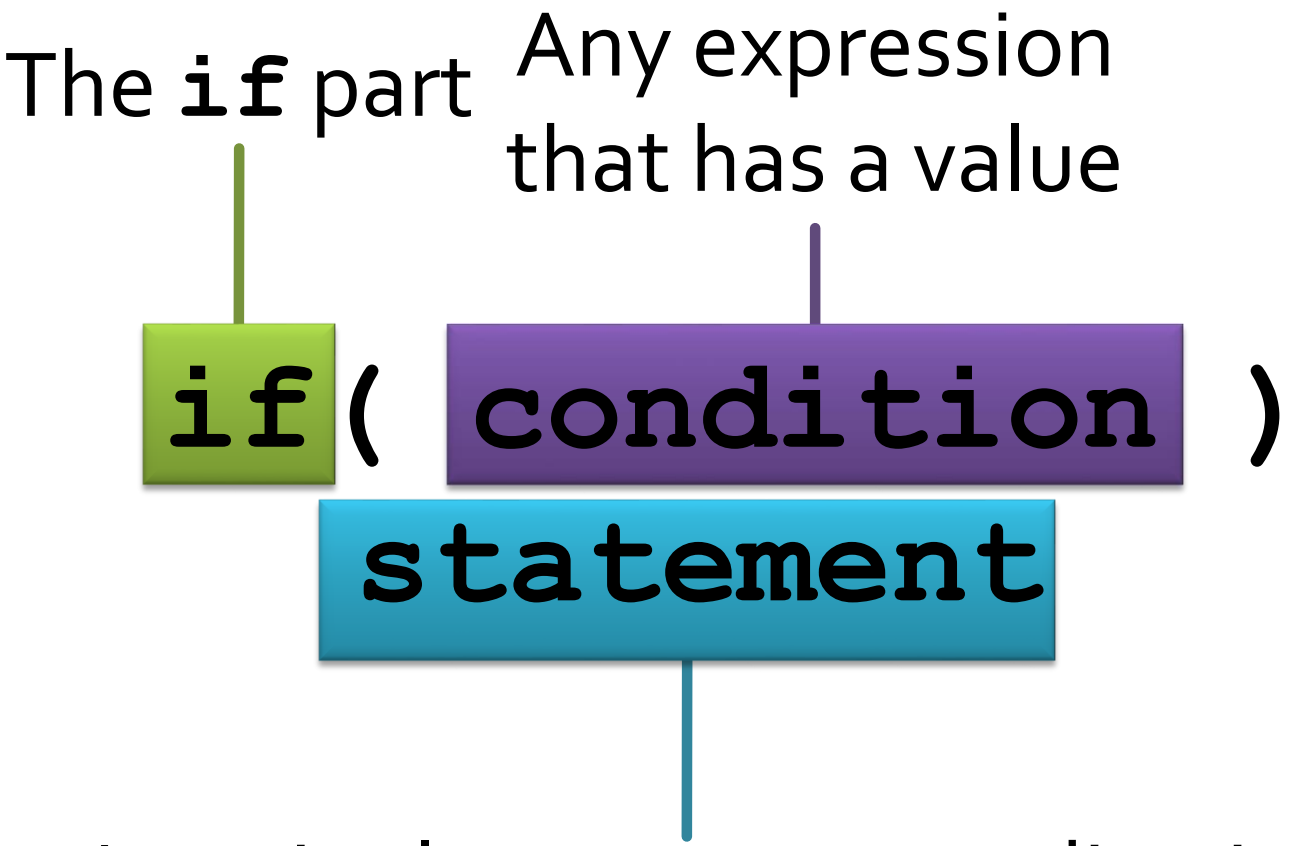
`if` statements

- Like Java, the body of an **`if`** statement will only execute if the condition is true
  - The condition is evaluated to an **int**
  - True means not zero

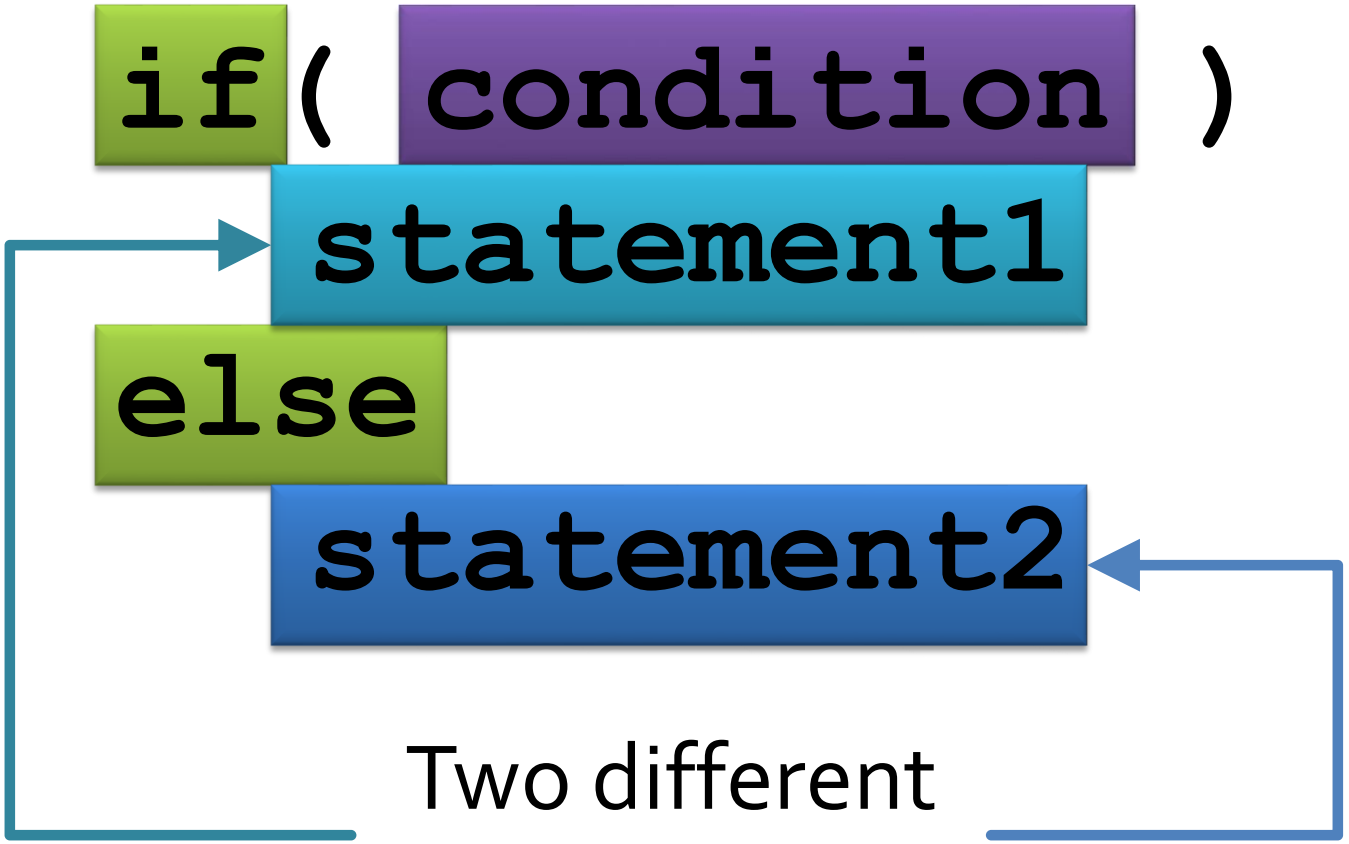
Sometimes this is natural and clear; at other times it can be cryptic.

- An **`else`** is used to mark code executed if the condition is false

Anatomy of an `if`



Anatomy of an `if-else`



Nesting

- We can nest **`if`** statements inside of other if statements, arbitrarily deep
- Just like Java, there is no such thing as an **`else if`** statement
- But, we can pretend there is because the entire **`if`** statement and the statement beneath it (and optionally a trailing **`else`**) is treated like a single statement

`switch` statements

- **`switch`** statements allow us to choose between many listed possibilities
- Execution will jump to the matching label or to **`default`** (if present) if none match
  - Labels must be constant (either literal values or **`#define`** constants)
- Execution will continue to fall through the labels until it reaches the end of the switch or hits a **`break`**
  - Don't leave out **`break`** statements unless you really mean to!

Anatomy of a switch statement

```
switch( data )
 {
        case constant1:
              statements1
        case constant2:
              statements2
         …
        case constantn:
              statementsn
        default:
              default statements
 }
```

Loops

Three loops

- C has three loops, just like Java
  - **`while`** loop
    - You don't know how many times you want to run
  - **`for`** loop
    - You know how many times you want to run
  - **`do-while`** loop
    - You want to run at least once
- Like **`if`** statements, the condition for them will be evaluated to an **`int`**, which is true as long as it is non-zero
  - All loops execute as long as the condition is true

`while` loop

- A **`while`** loop is the keyword while followed by a pair of parentheses
- Within the parentheses is a condition
- If the condition is true, the body of the loop will be executed
- At the end of the loop, the condition is checked again

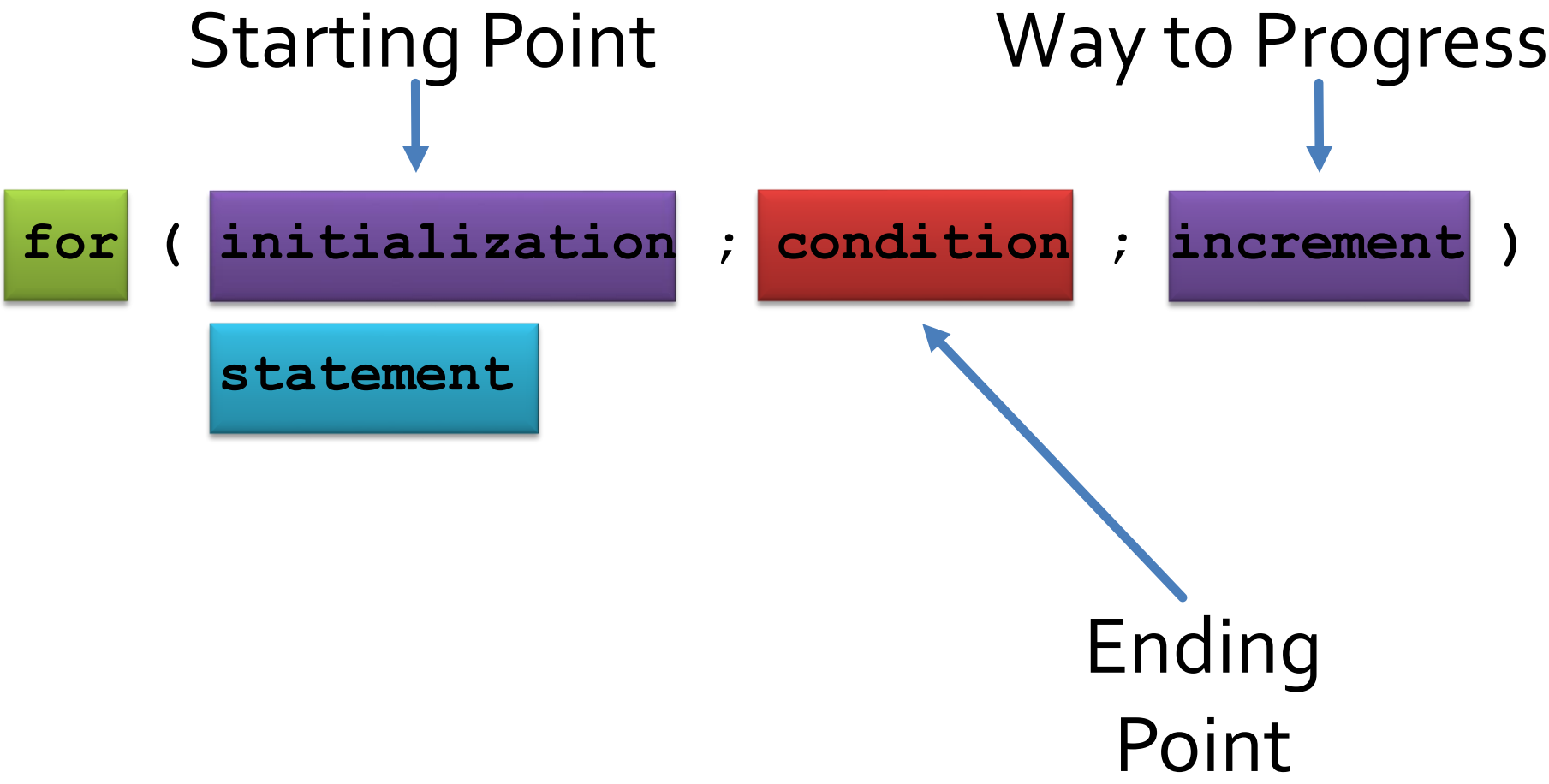
Anatomy of a `while` loop

```
while( condition )
    statement
```

`for` loop

- A **`for`** loop consists of three parts:
  - Initialization
  - Condition
  - Increment
- The initialization is run when the loop is reached
- If the condition is true, the body of the loop will be executed
- At the end of the loop, the increment will be executed and the condition checked again
  - If the condition is empty (nothing in it), it is considered true

Anatomy of a `for` loop



The comma operator

- C has a comma operator
- Expressions can be written and separated by commas
- Each will be evaluated, and the last one will give the value for the entire expression

```
int a = 10;
int b = 5;
int c = a, b, ++a, a + b; //16
```

Adding the comma to for

- Sometimes you want to do multiple things on each iteration
- Consider this code to reverse an array

```
int start;
int end;
int temp;
for( start = 0, end = length - 1; start < end;
     start++, end-- )
{
     temp = array[start];
     array[start] = array[end];
     array[end] = temp;
}
```

- You can even use a comma in the condition part, but it doesn't usually make sense

do-while loops

- As in Java, there are do-while loops which are useful only occasionally
- They work just like **`while`** loops except that that they are guaranteed to execute at least once
- Unlike a **`while`** loop, the condition isn't checked the first time you go into the loop
- Sometimes this is useful for getting input from the user
- Don't forget the semicolon at the end!

Anatomy of a do-while loop

```
do
    statement
while( condition );
```

Duff's device

- C has relatively relaxed syntax rules

```
int n = (count + 7) / 8;
switch (count % 8)
{
        case 0: do {  *to++ = *from++;
        case 7:       *to++ = *from++;
        case 6:       *to++ = *from++;
        case 5:       *to++ = *from++;
        case 4:       *to++ = *from++;
        case 3:       *to++ = *from++;
        case 2:       *to++ = *from++;
        case 1:       *to++ = *from++;

        } while (--n > 0);
}
```

- What the hell is that?!

Common Loop Errors

Infinite loops

- Loops can go on forever if you aren't careful

```
int n = 40;
int i = 1;

while( i <= 40 )
{
  printf("%d", i);
  //supposed to print all the numbers
  //less than 40, but i never increases
}
```

Infinite `for` loops

- Infinite **`for`** loops are unusual, but possible:

```
for( ; ; )
  printf("Hey!");
```

- This situation is more likely:

```
int i;
for(i = 0; i < 10; i++ )
{
  printf("%d", i);
  //lots of other code
  i--; //whoops, maybe changed from while?
}
```

(Almost) infinite loops

- Overflow and underflow will make some badly written loops **eventually** terminate

```
int i;

for( i = 1; i <= 40; i-- )
//whoops, should have been i++
  printf("%d", i);
```

Fencepost errors

- Being off by one is a very common loop error

```
int i;

for( i = 1; i < 40; i++ )
//runs 39 times
  printf("%d", i);
```

Skipping loops entirely

- If the condition isn't **`true`** to begin with, the loop will just be skipped

```
int i;
for( i = 1; i >= 40; i++ )
//oops, should be <=
   printf("%d", i);
```

Misplaced semicolon

- A misplaced semicolon can cause an empty loop body to be executed

```
int i;
for( i = 1; i <= 40; i++ );
//semicolon is wrong
{
  printf("%d", i);
}
```

- Everything looks good, loop even terminates
- But, only one number will be printed: 41
- Misplaced semicolon usually makes a **`while`** loop infinite

Lab 3

Upcoming

Next time…

- Systems programming concepts
- Functions and prototypes

Reminders

- Read LPI chapter 3 and K&R chapter 4
- **Project 1 due tonight by midnight!**