COMP1511 Week 2 Lecture 1
Control Flow

Quick notices

– Help sessions starting early! Keep an eye on course page
– Keep the feedback coming!

Feedback overview

- We like the lecture format, quality and in-person lectures
- We, on average, seem to like the pace
- We would like to see the slides up earlier
- Less typos!

Last week

- ✓ Went to tute/lab
- ✓ hello_world.c
- ✓ memory
- ✓ reading/writing to terminal
- ✓ arithmetic

This week

- ☐ control flow
- ☐ logical operators
- ☐ repetition

Input/Output recap

`printf()`

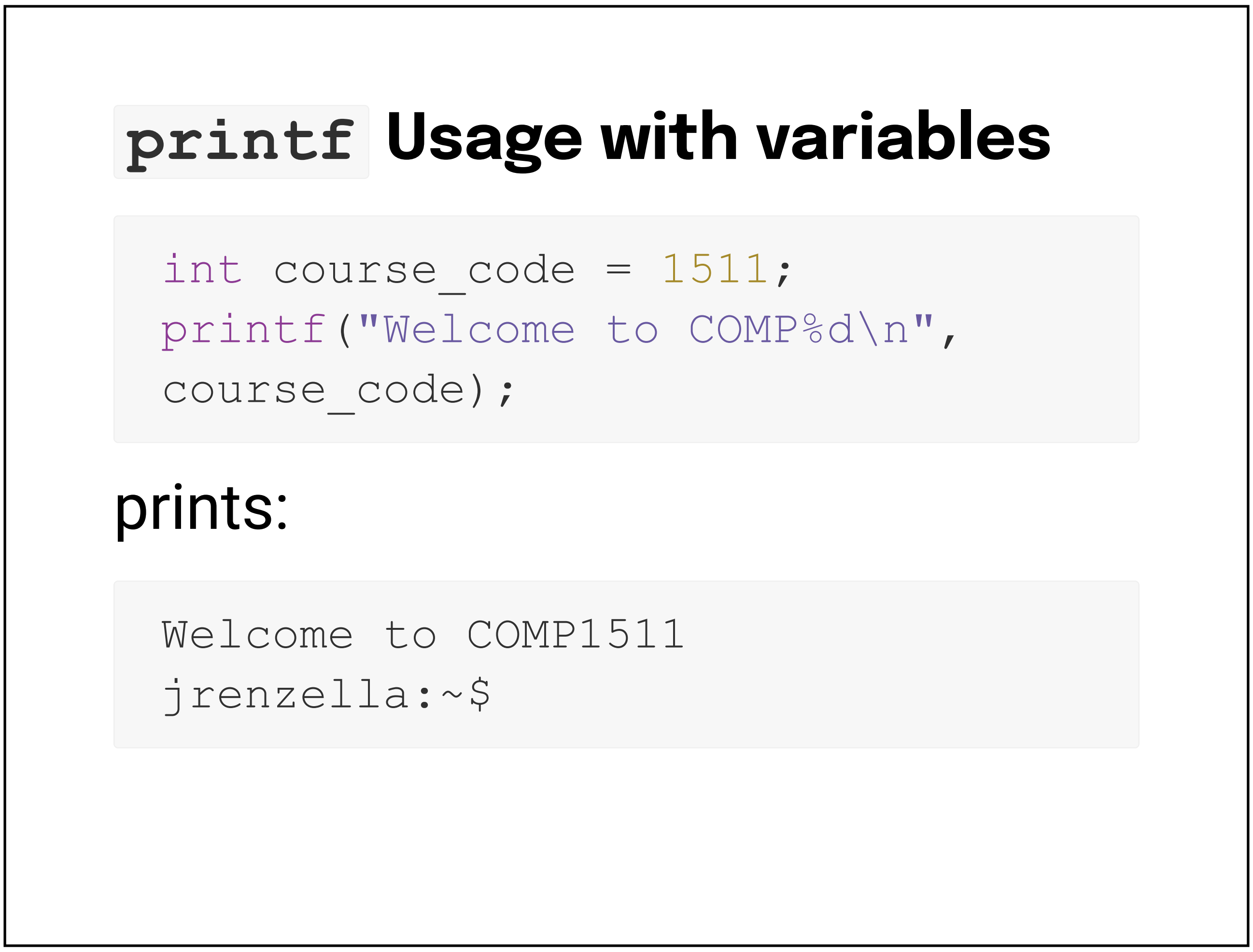
- Outputs text to terminal
- stands for *print formatted*
- Need to import `#include <stdio.h>` to use

What will this print out?

```
int course_code = 1511;
printf("Welcome to COMP%d\n",
course_code);
```

x: top

y: left



`printf` Usage with variables

```
int course_code = 1511;
printf("Welcome to COMP%d\n",
course_code);
```

prints:

```
Welcome to COMP1511
jrenzella:~$
```

Each format flag (%d) is mapped to a passed in variable at the end. Each variable must have a corresponding %d/%lf, etc.

newlines

`\n`

```
printf("Hello
world!")
```

```
Hello
world!jrenzella:~$
```

```
printf("Hello
world!\n")
```

```
Hello world!
jrenzella:~$
```

`scanf()`

- reads text from terminal (input)
- stands for *scan formatted*
- Need to import `#include <stdio.h>` to use

`scanf` usage

```
int age;
printf("Enter your age: ");
scanf("%d", &age);
```

^ reads an integer from the terminal and stores it in `age`.

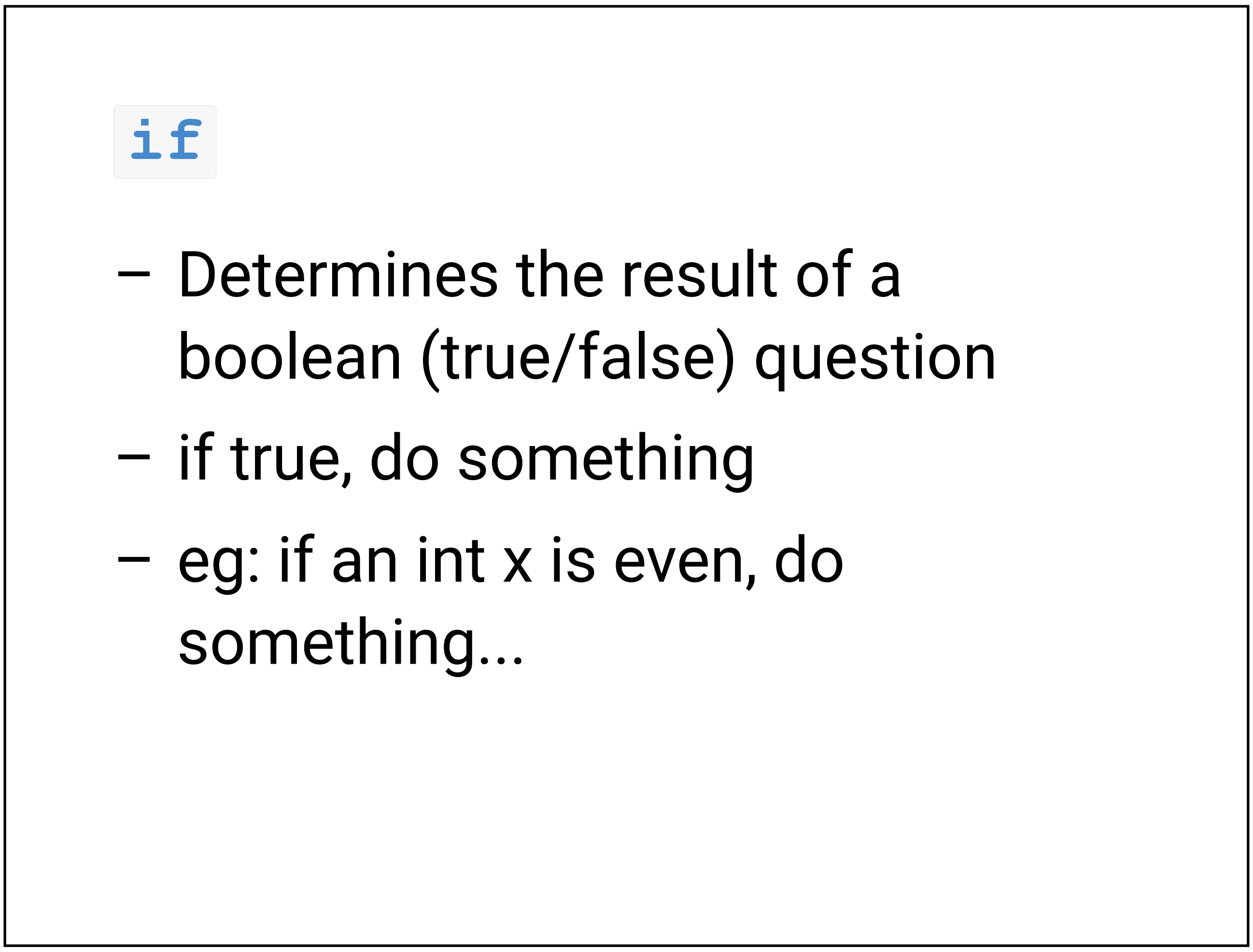
- `%d` tells scanf to look for a `decimal integer`.
- We need to use `&` before the variable, more on that in a few weeks...

Week 2 Lecture 1
Control Flow

Control Flow

– Sometimes we need to make decisions in our programs
– We can make our programs branch between sets of instructions
– To do this, we use the `if` statement.

Enter the `if` statement



`if`

- Determines the result of a boolean (true/false) question
- if true, do something
- eg: if an int x is even, do something...

x: left

y: top

Understanding true and false in C

`true` and `false` are integers in C

- `true` -> `1`
- `false` -> `0`
- later versions of C added `true` and `false` as synonyms *(need to* `#include <stdbool.h>` *to use these)*

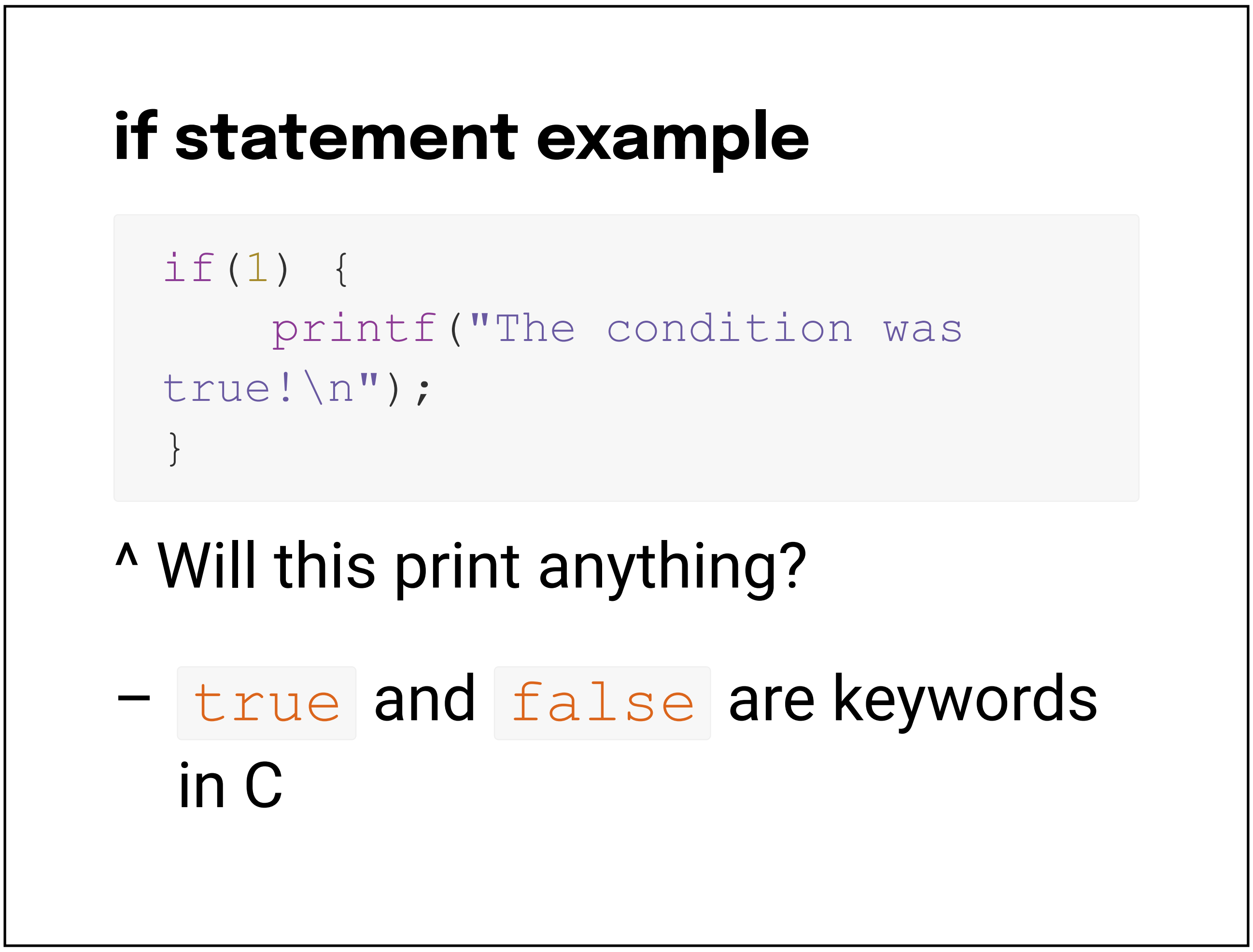
Added in 1999!

stdbool.h stands for standard boolean library. Gives us some these types. See https://en.wikibooks.org/wiki/C_Programming/stdbool.h for more info.

if statement syntax

```
if(<condition>) {
    do_something();
    do_something_else();
}
```

- `if` statement -> requires a condition, executes if true
- **`<condition>`** -> something that evaluates to true/false
- `{...}` -> everything inside will run if condition is true



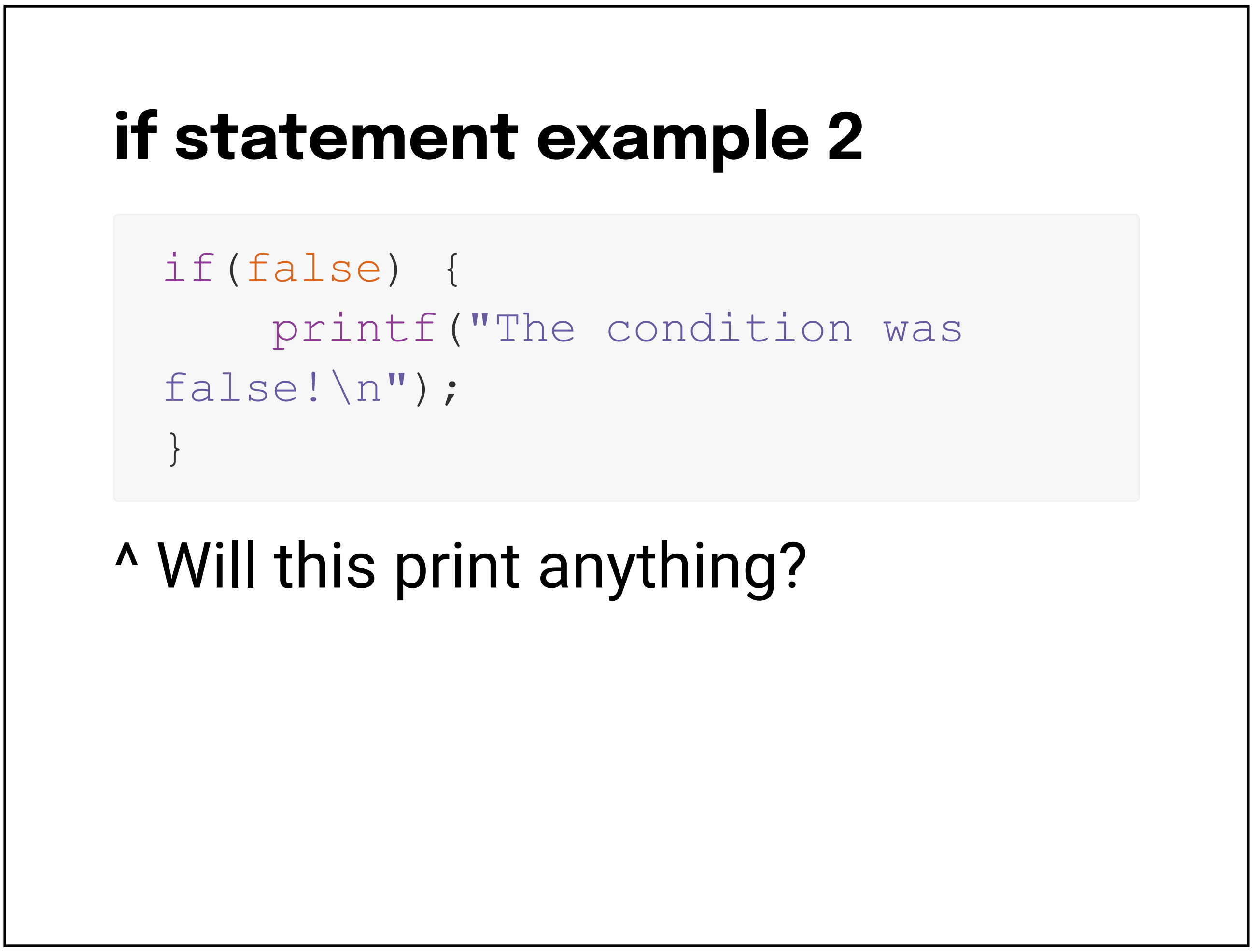
if statement example

```
if(1) {
    printf("The condition was
true!\n");
}
```

^ Will this print anything?

– `true` and `false` are keywords in C

Yes! `true` evaluates to true, so the printf statement executes.

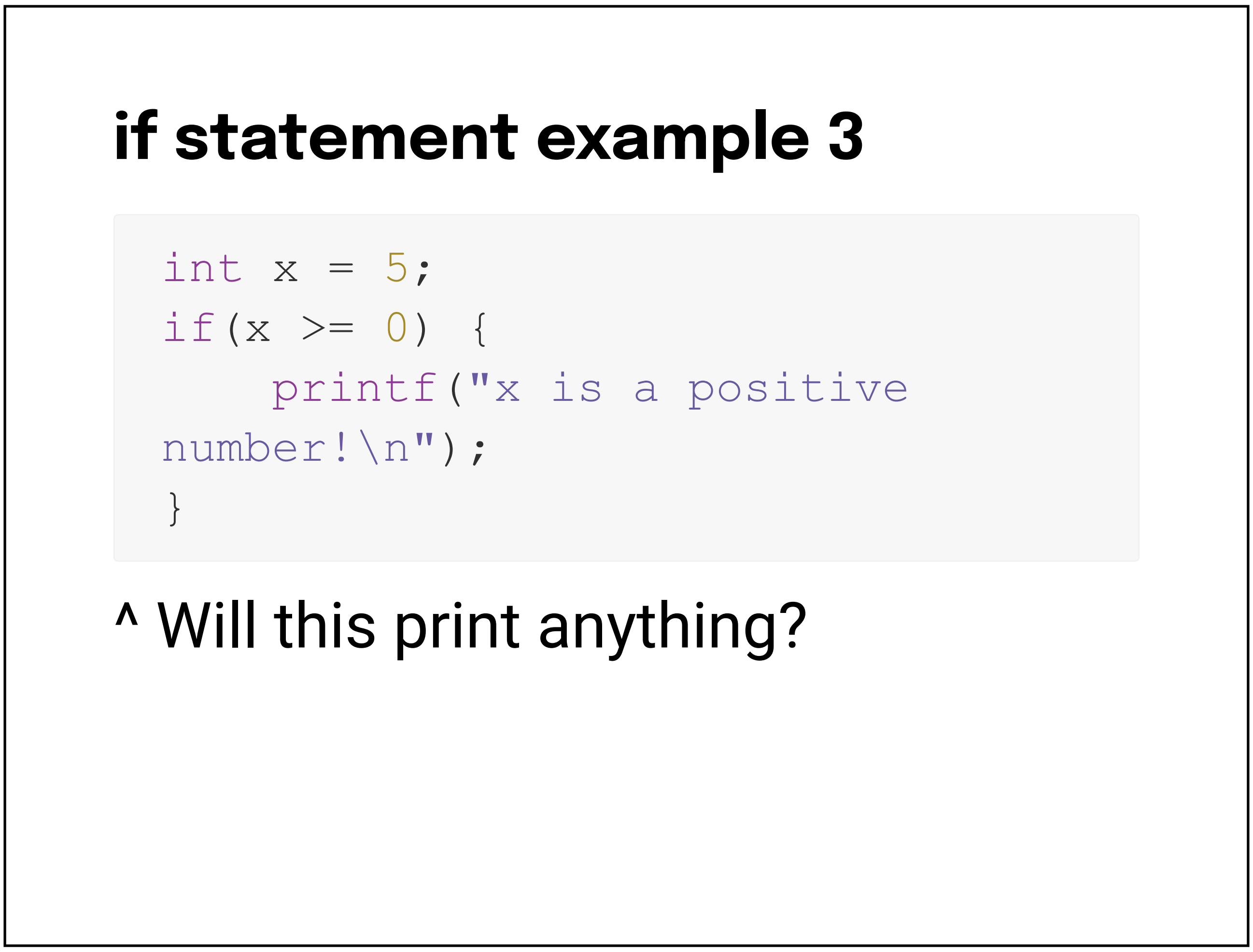


if statement example 2

```
if(false) {
    printf("The condition was
false!\n");
}
```

^ Will this print anything?

No, false evaluates to false, so the block does not run at all.



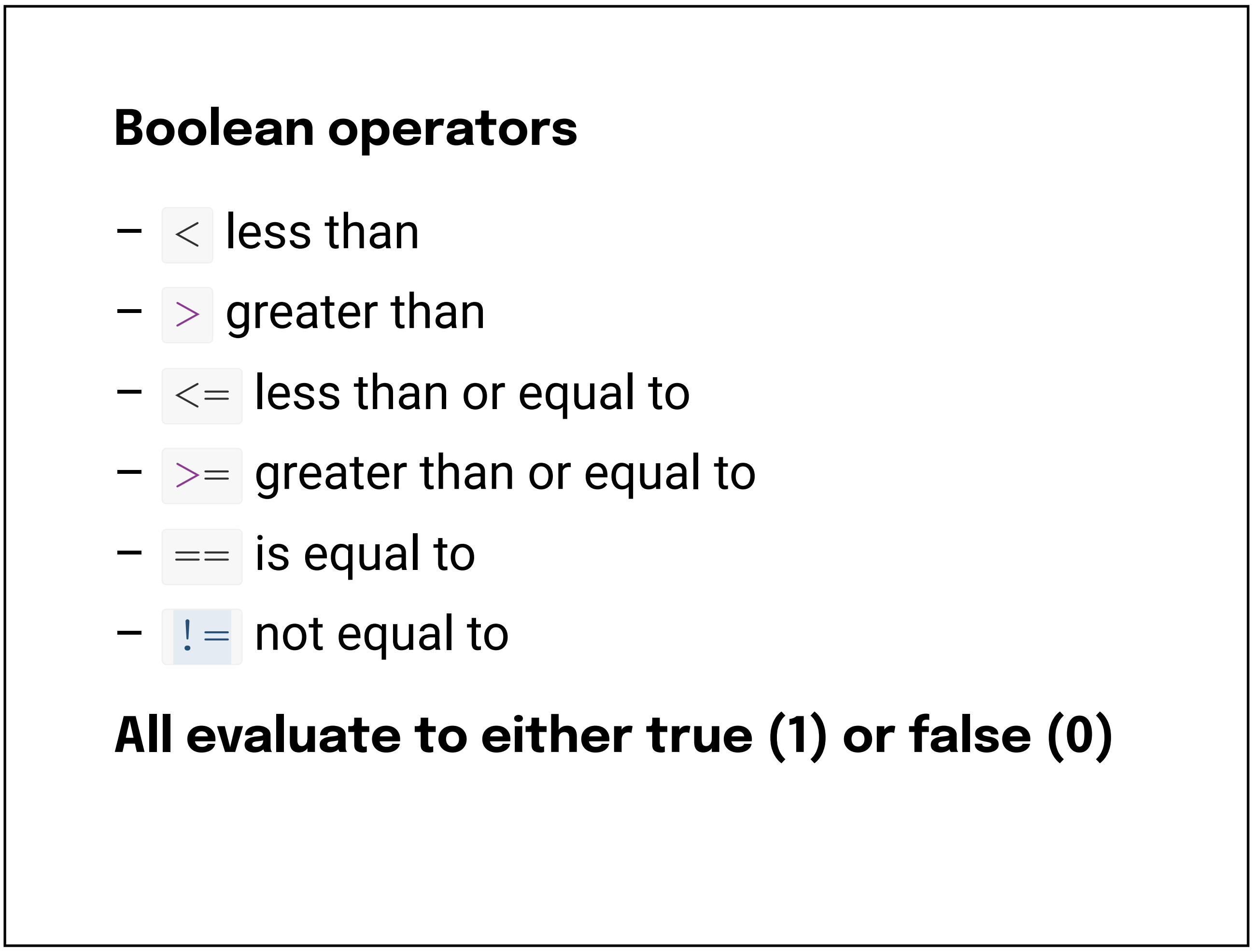
if statement example 3

```
int x = 5;
if(x >= 0) {
    printf("x is a positive
number!\n");
}
```

^ Will this print anything?

Yes, >= checks if the left is greater than or equal to the right, so in this case is 5 greater than or equal to 0, which is... true! true things run!

Wait what is `>=`?

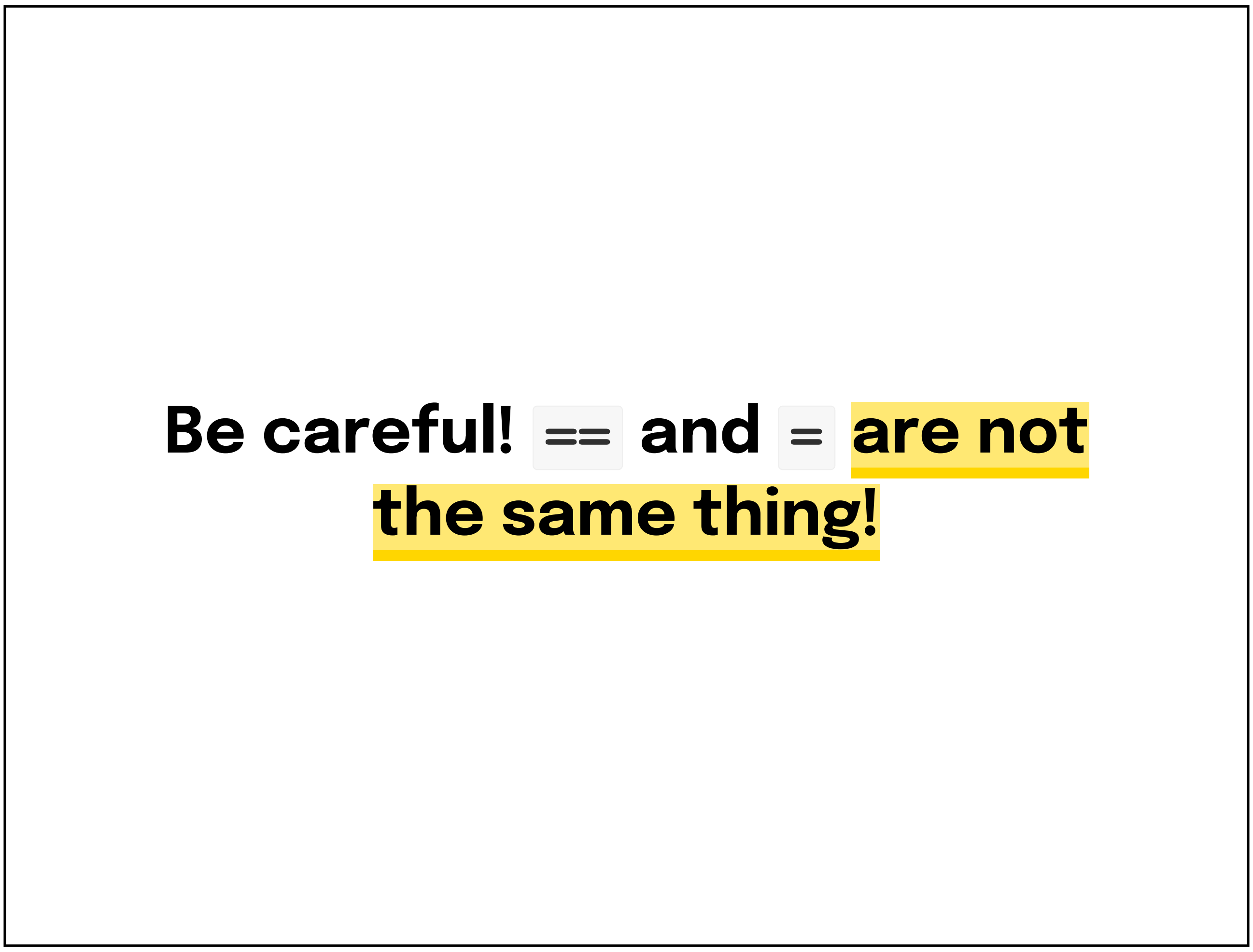


Boolean operators

- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal to
- `==` is equal to
- `!=` not equal to

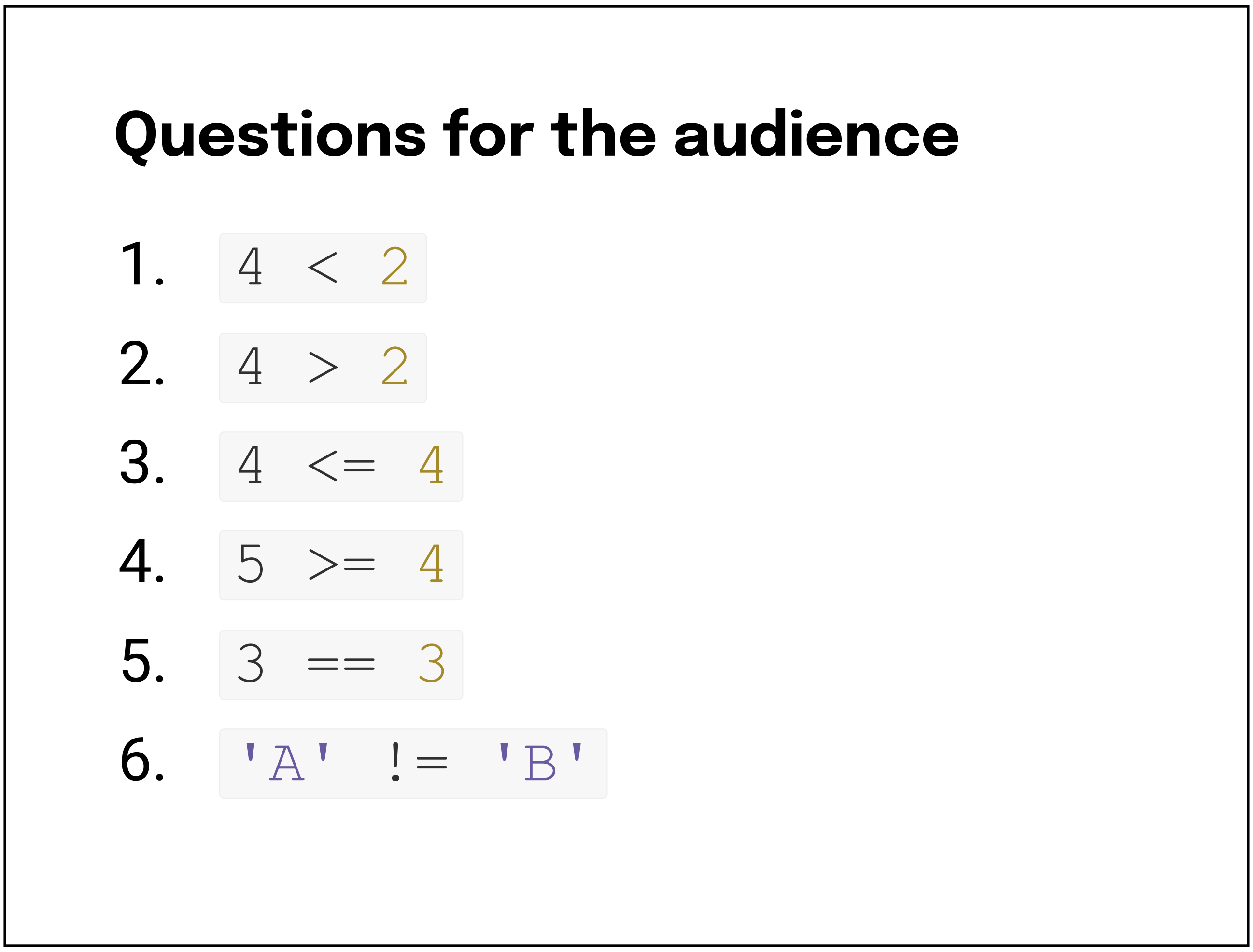
All evaluate to either true (1) or false (0)

They determine if the thing on the left is less than/greater than/equal to, etc the thing on the right.



Be careful! `==` and `=` are not the same thing!

They look the same, but are not... `=` is assignment, `==` is equivalence.



Questions for the audience

1. `4 < 2`
2. `4 > 2`
3. `4 <= 4`
4. `5 >= 4`
5. `3 == 3`
6. `'A' != 'B'`

1. false
2. true
3. true
4. true
5. true
6. true

Demo

More control flow

The `else` statement

- Sometimes we want to run a block of code if the `if` statement is false!
- the `else` statement **must** be associated with an `if` statement.
- it only runs if the condition evaluates to false

`else` statement syntax

```
if(<condition>) {
    do_something();
    do_something_else();
} else {
  do_if_false();
}
```

- Notice there is no condition, because one is not needed
- else is optional

`else` statement example

```c
int x = -5;
if(x > 0) {
    printf("x is positive\n");
} else {
    printf("x is negative\n");
}
```

chaining `if` statements

We can *chain* multiple if statements to check for multiple options

```
if(<condition>) {
    do_something();
    do_something_else();
} if (<second_condition>) {
    do_if_second_condition();
}
```

What if we want to check if two things are true?

Boolean operators

- `&&` -> **and** operator
- `||` -> **or** operator
- `!` -> **not** operator

putting it all together

```
int age = 15;
int drinking_age = 18;

if(age > 0 && age < 18) {
    // age is valid, but not legal
} else if (age > 18) {
    // legal age
} else {
    // invalid age!
}
```

Live coding

Repetition
Repetition
Repetition
Repetition

Why do we need to loop?

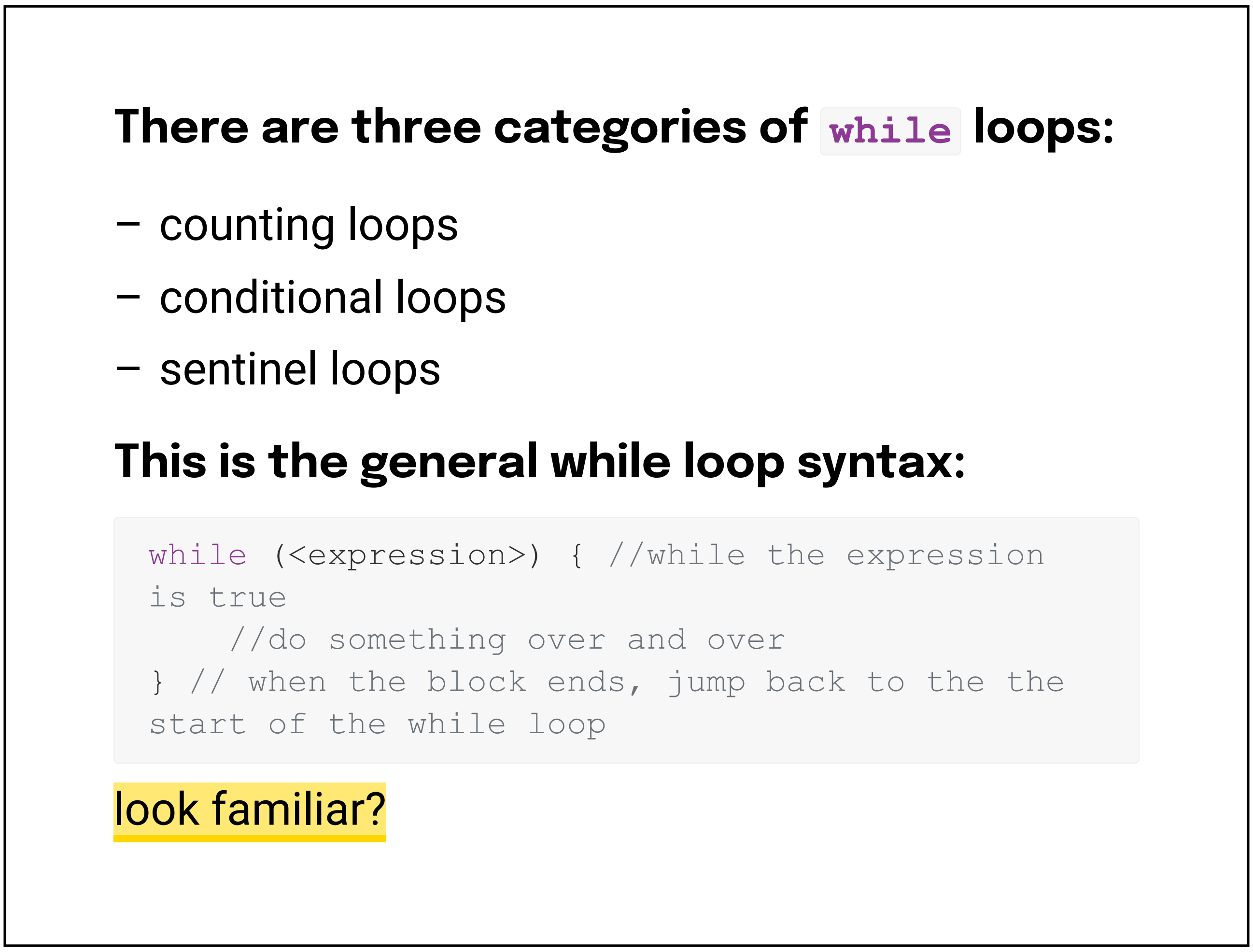
Programmers are lazy, we don't like repeating ourselves...

We can make computers do that for us!

What are some real world examples?

Enter the `while` statement

- Repetitive tasks shouldn't require repetitive code
- C starts at main and executes each line in sequence
- We can control that sequence



There are three categories of `while` loops:

– counting loops

– conditional loops

– sentinel loops

This is the general while loop syntax:

```
while (<expression>) { //while the expression
is true
    //do something over and over
} // when the block ends, jump back to the the
start of the while loop
```

look familiar?

Same syntax as `if` statements!

counting loops

- do something `n` amount of times (counting up to `n`)

```c
int number_of_lines = 5;
int i = 0;

while (I < number_of_lines) {
    printf("hey!\n");
    i = i + 1;
}
```

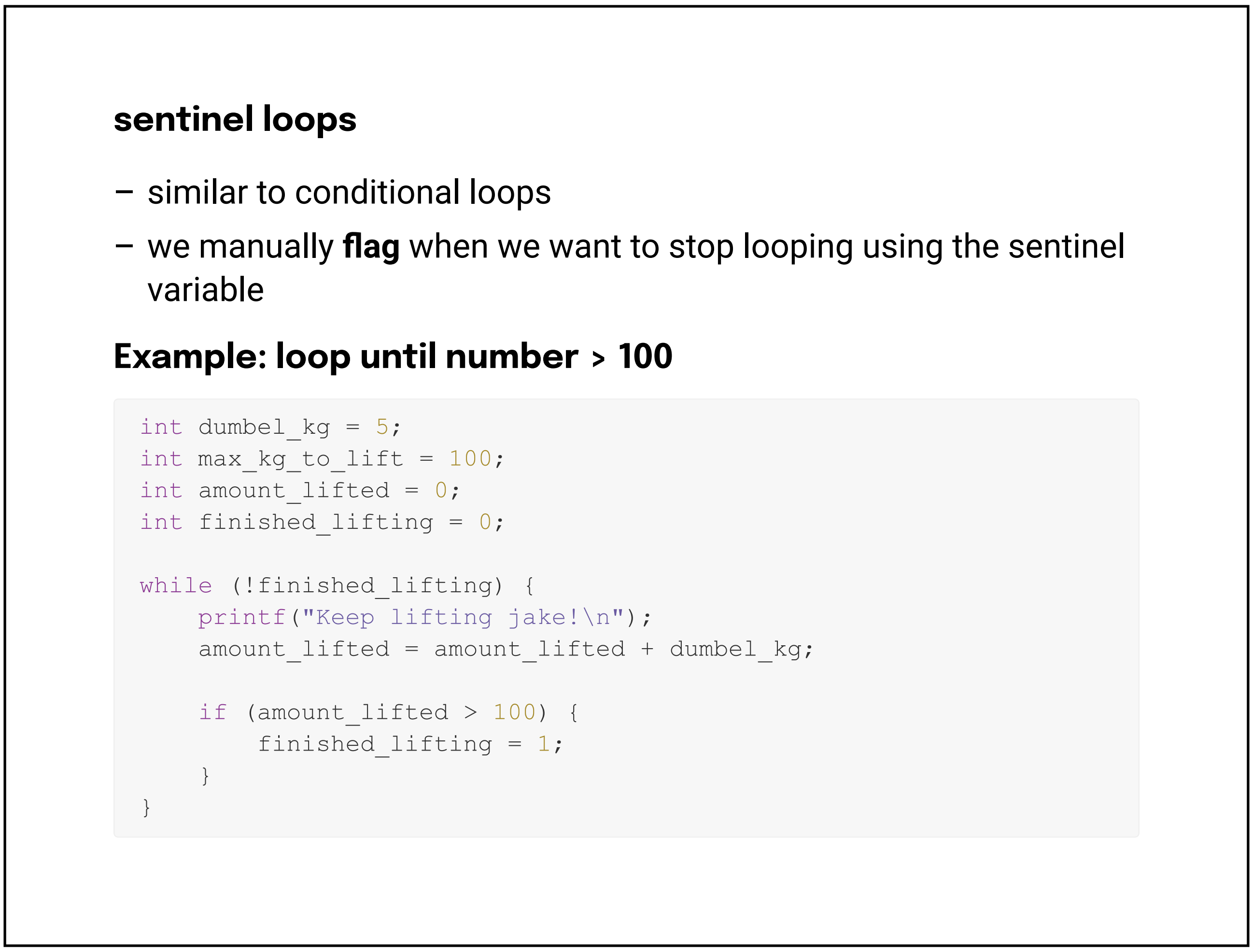
conditional loops

- do something until the condition is true
- we don't know how many times we will need to loop

Example: loop until number > 100

```
int dumbel_kg = 5;
int max_kg_to_lift = 100;
int amount_lifted = 0;

while (amount_lifted < 100) {
    printf("Keep lifting jake!\n");
    amount_lifted = amount_lifted + dumbel_kg;
}
```



sentinel loops

- similar to conditional loops
- we manually **flag** when we want to stop looping using the sentinel variable

Example: loop until number > 100

```c
int dumbel_kg = 5;
int max_kg_to_lift = 100;
int amount_lifted = 0;
int finished_lifting = 0;

while (!finished_lifting) {
    printf("Keep lifting jake!\n");
    amount_lifted = amount_lifted + dumbel_kg;

    if (amount_lifted > 100) {
        finished_lifting = 1;
    }
}
```

In this particular example, the conditional loop version is better, however sentinel loops give us more fine-grain control over when we want to stop looping.

Feedback