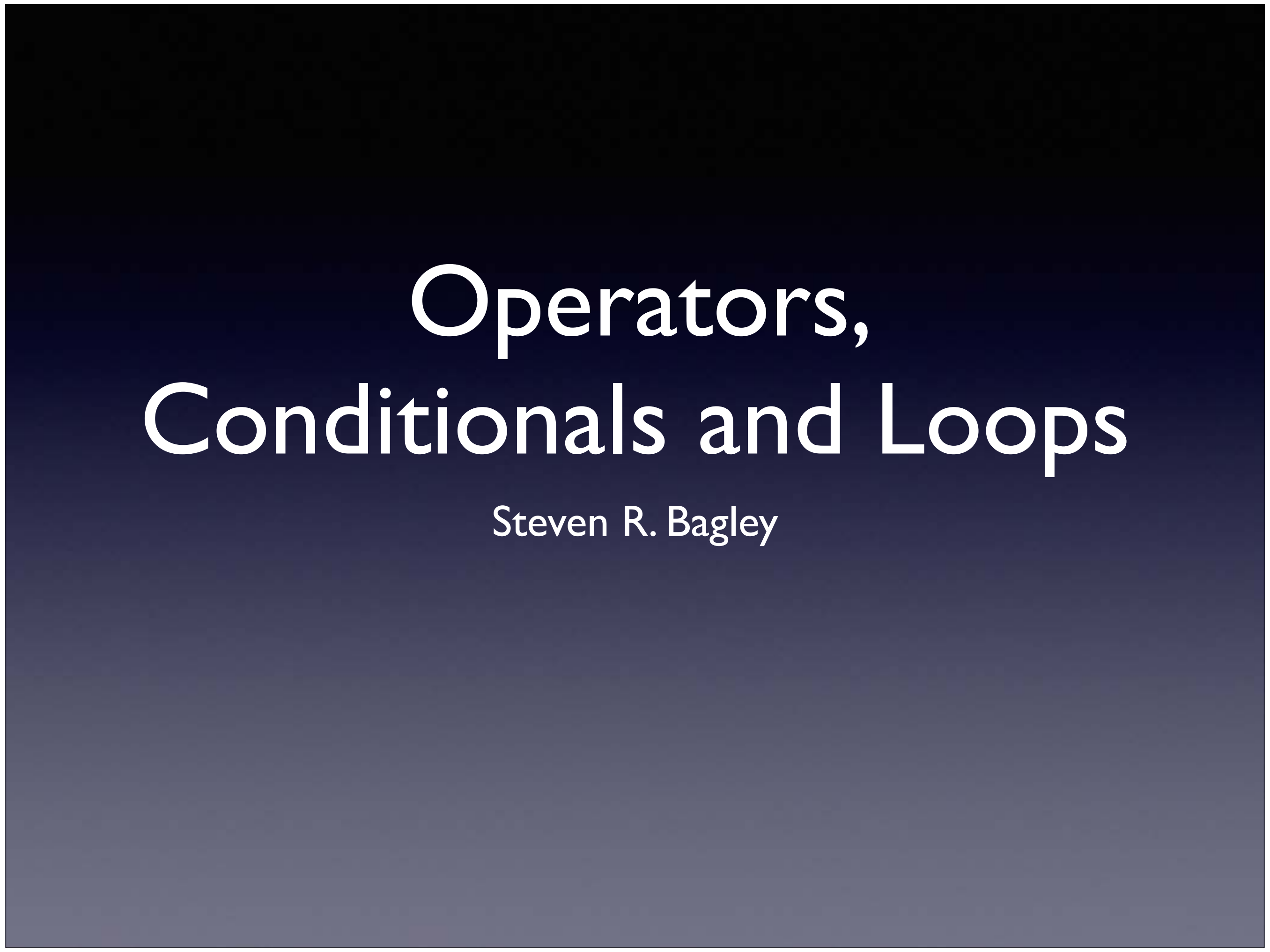
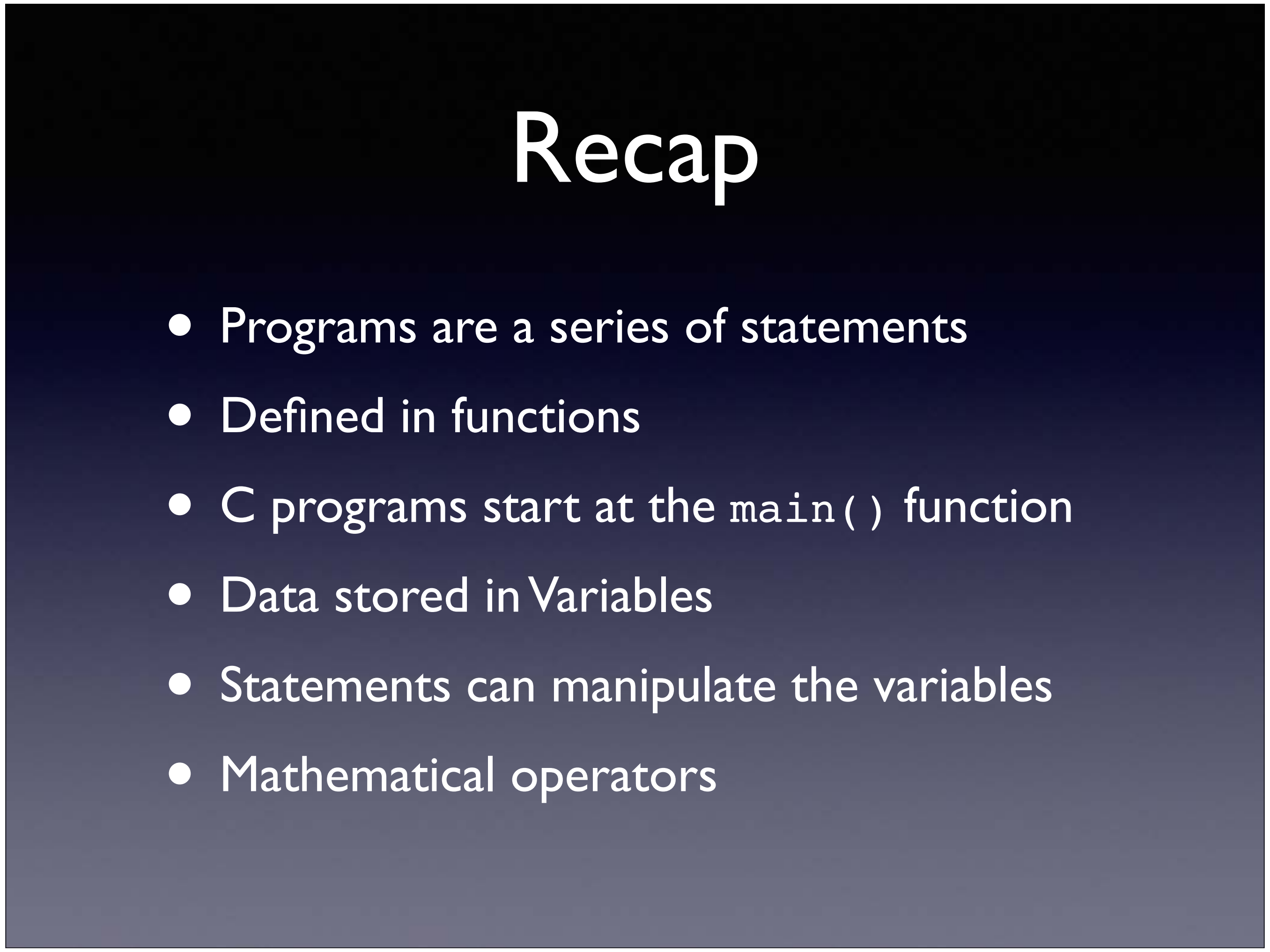


Operators, Conditionals and Loops

Steven R. Bagley

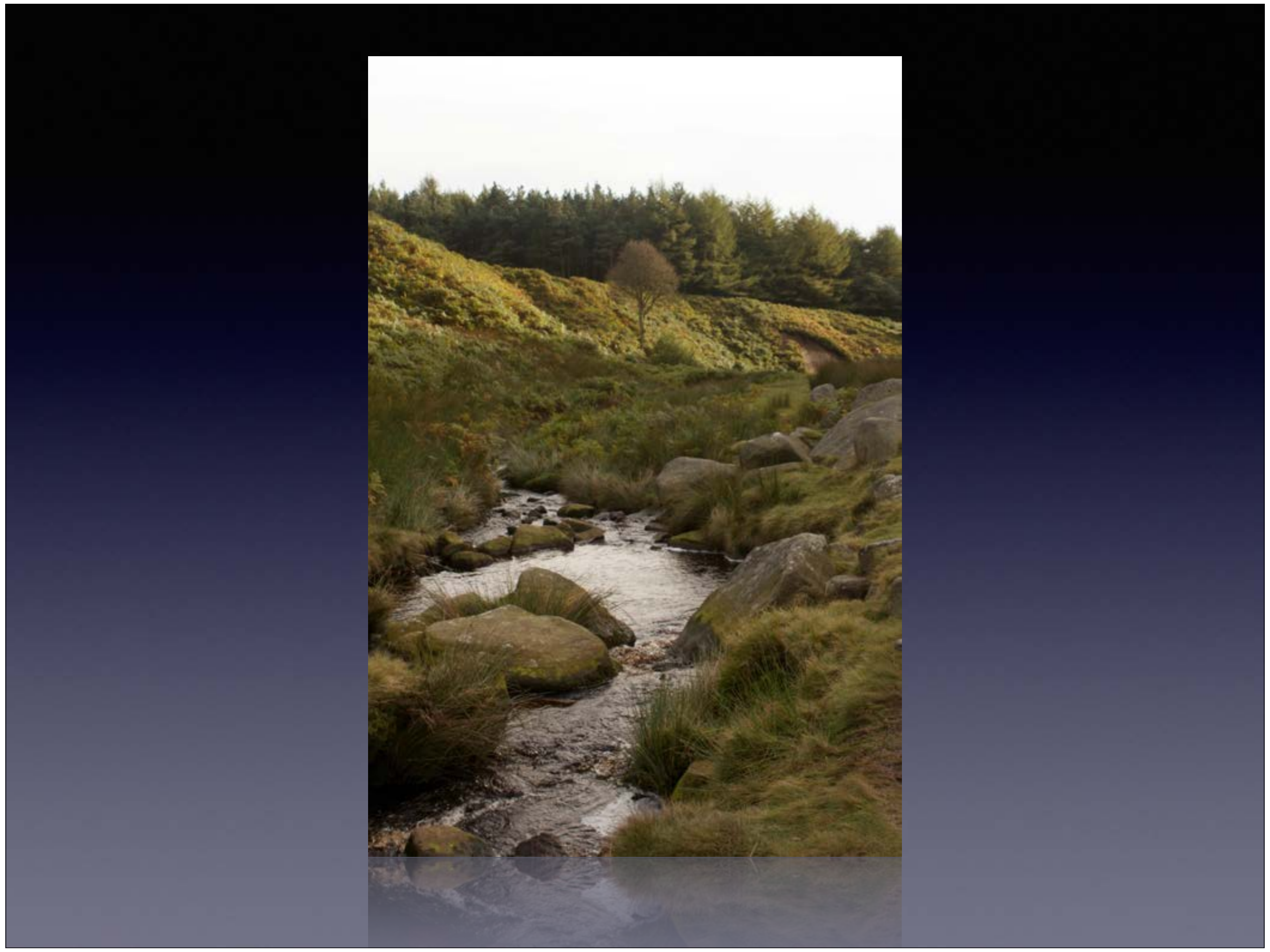


Recap

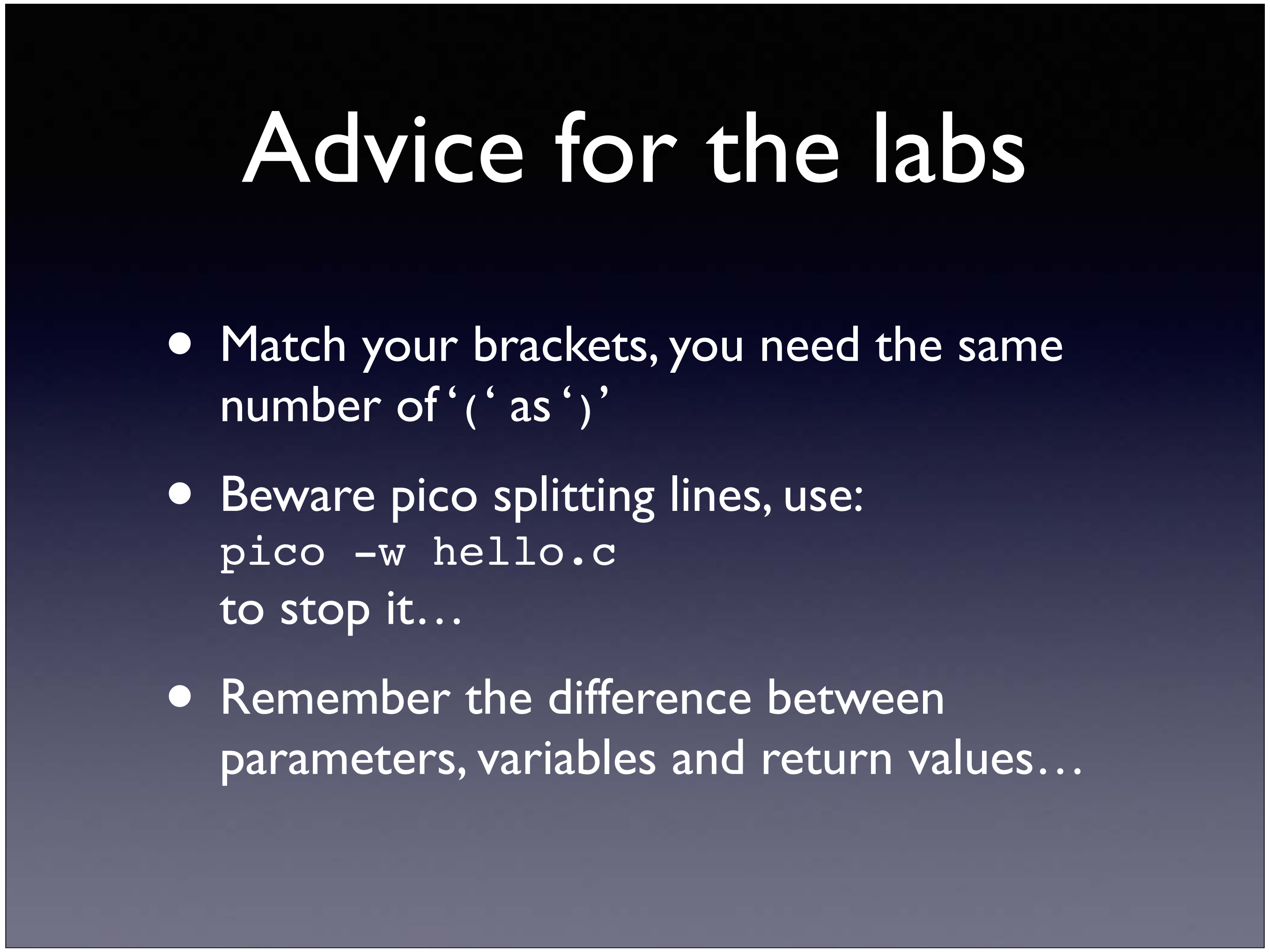
- Programs are a series of statements
- Defined in functions
- C programs start at the `main()` function
- Data stored in Variables
- Statements can manipulate the variables
- Mathematical operators

Advice for the labs

- Statements executed strictly in order…
- End every statement with a semicolon ‘`;`’
- Compile early and often
- Don’t expect to get it right first time
- Declare things before you use them
- All programs will have a `main()` etc.



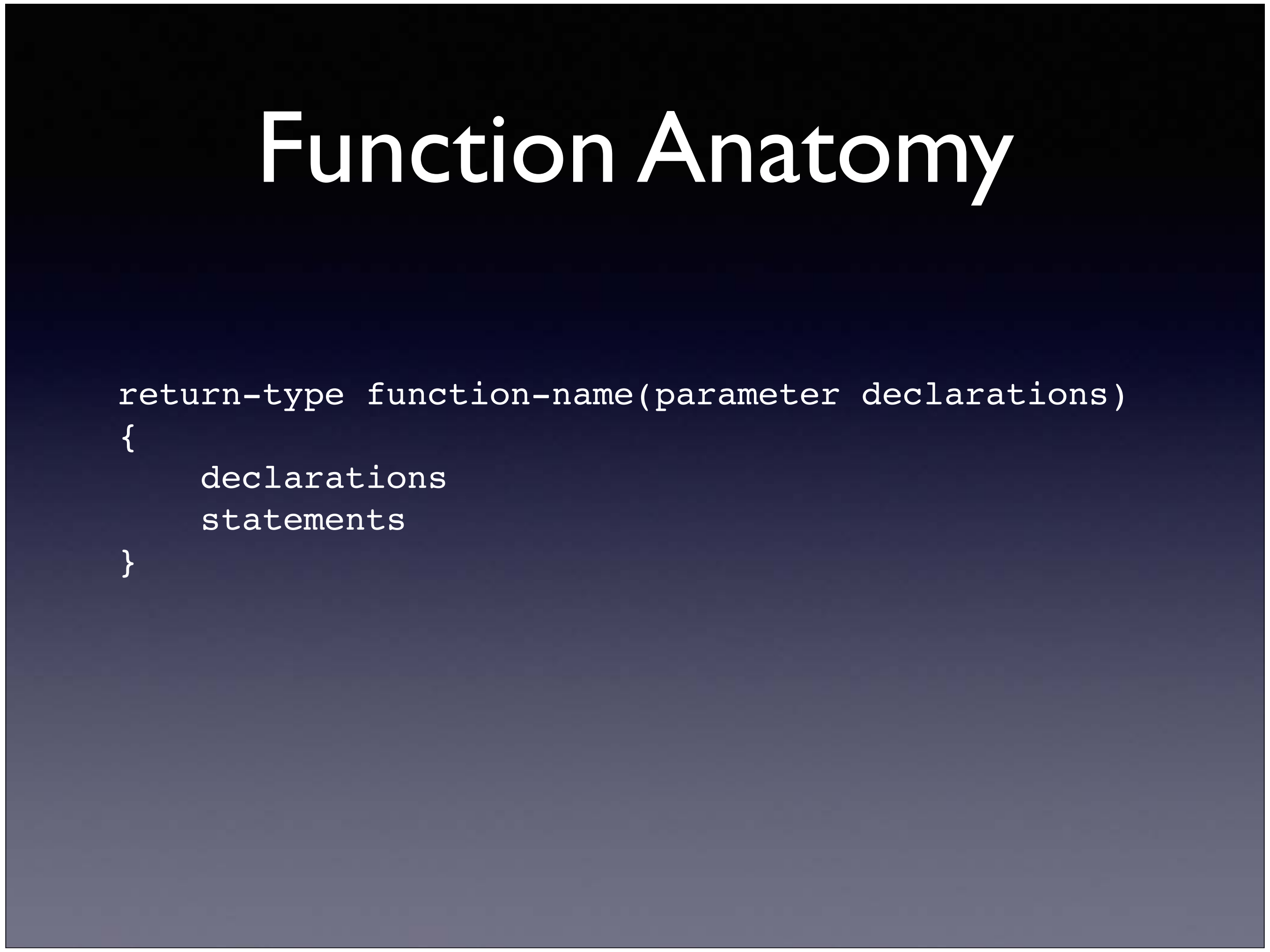
Compile early and compile often -- build the program up piece-by-piece,
step-by-step. Each step should move you closer to the working program, and compile!



Advice for the labs

- Match your brackets, you need the same number of ‘(‘ as ‘)’
- Beware pico splitting lines, use:
  ```
  pico -w hello.c
  ```
  to stop it…
- Remember the difference between parameters, variables and return values…

Or investigate other editors…



Function Anatomy

```
return-type function-name(parameter declarations)
{
    declarations
    statements
}
```

All statements end with a semicolon in C -- forget them and you’ll get compile errors (and not sensible ones)

Function Anatomy

- *Return type* — states the type of the value returned from the function (if any)
- *Function name* — `main` in this case
- *Parameter declarations* — that the function uses to perform its task (if any), also have type
- { … } — groups statements together
- *Declarations* — declare any variables used in the function
- *Statements* — what the program does (almost always ended by a semicolon)
- Return to the issue of type later

Variable Anatomy

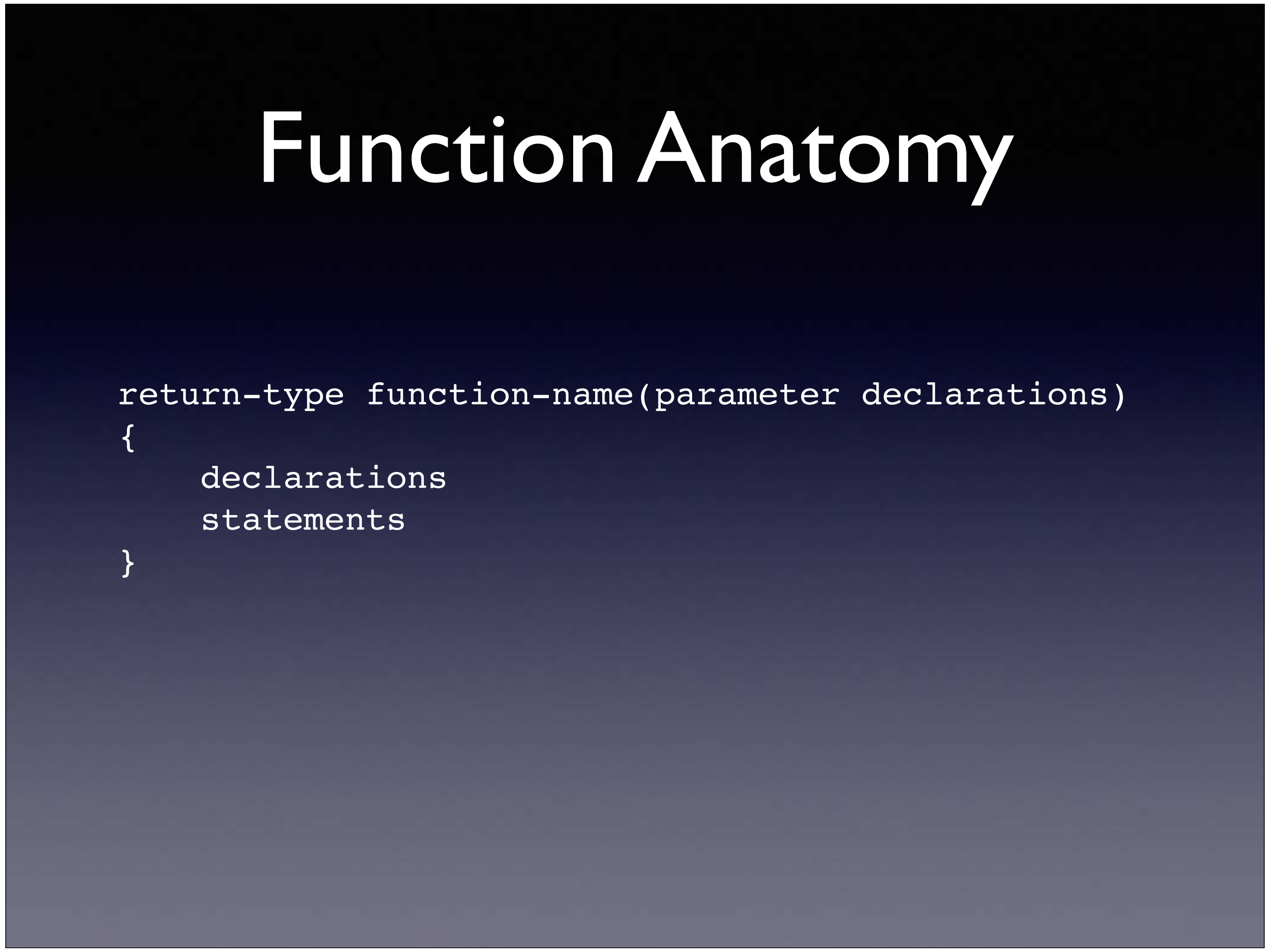
- Must be declared
  `double celsius;`
- Can assign a value to the variable
  `celsius = 25.0;`
- Value can be read by just using its name
  ```
  printf(“Temperature is %f”,
  celsius);
  ```
- Scope

Values

- Can be a *literal* value (e.g. `42, 23.5, ‘a’`)
- Read from a variable (by giving the name)
- Calculated (`x + 1`, `x*x + y*y`)
- Returned from functions (`getchar()`)

Parameters

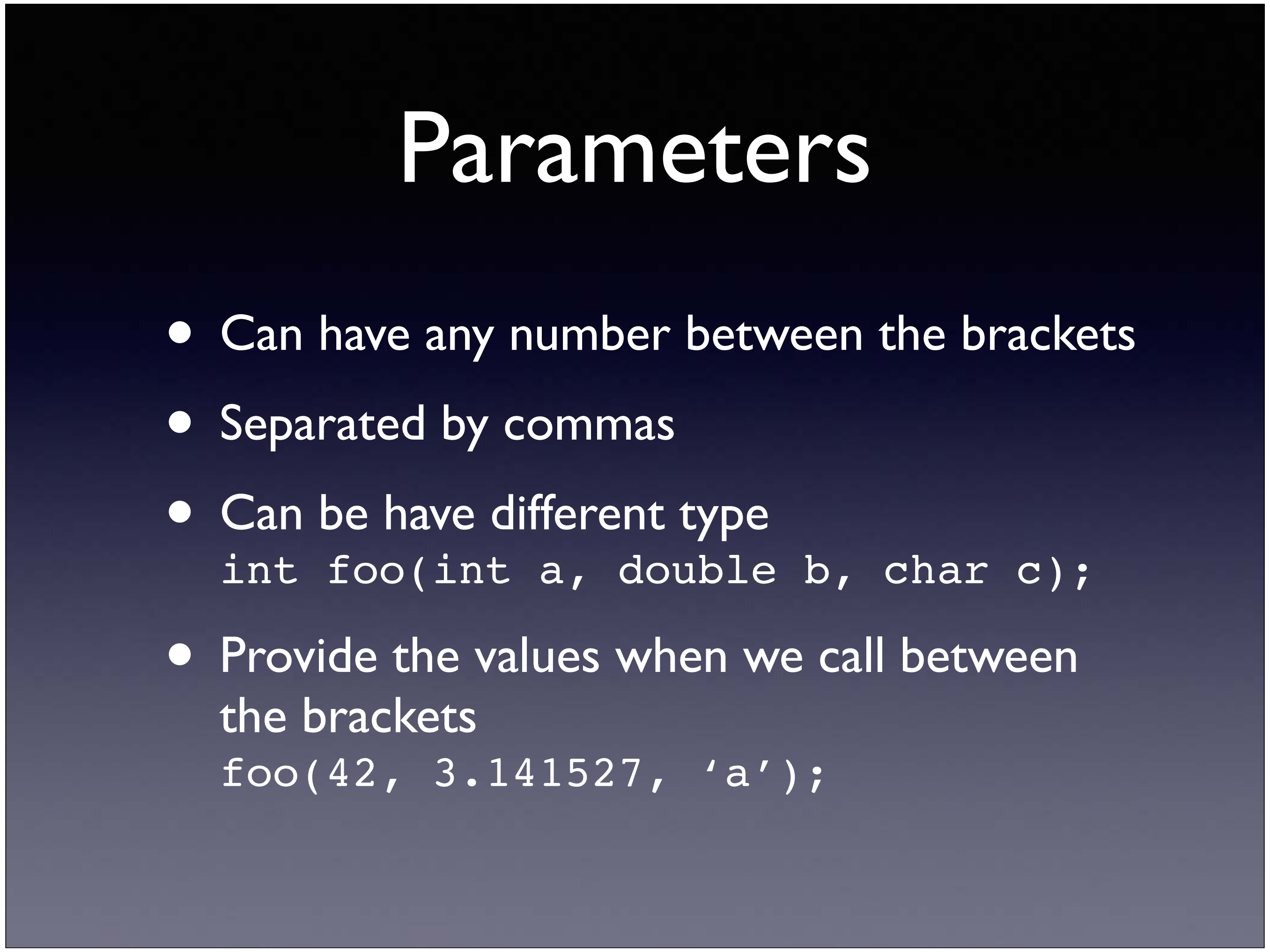
- Inside the function, act like local variables
- But think of them as having been pre-initialized with some value
- Value *passed in* when function called
- Declared *separately* from variables in the type signature



Function Anatomy

```
return-type function-name(parameter declarations)
{
    declarations
    statements
}
```

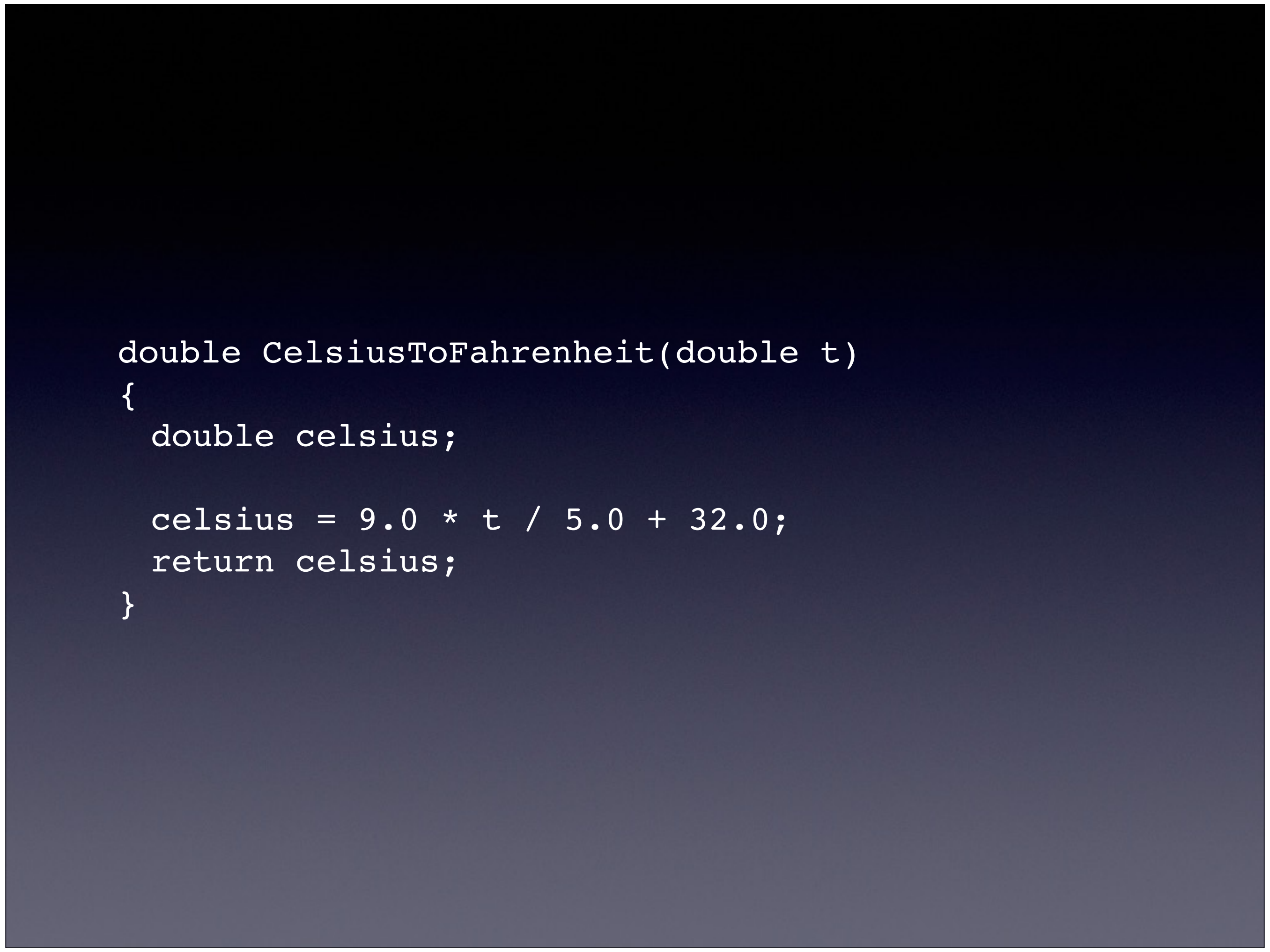
All statements end with a semicolon in C -- forget them and you'll get compile errors (and not sensible ones)



Parameters

- Can have any number between the brackets
- Separated by commas
- Can be have different type
  ```
  int foo(int a, double b, char c);
  ```
- Provide the values when we call between the brackets
  ```
  foo(42, 3.141527, ‘a’);
  ```

No need for the names of parameters when calling -- that happens automatically
Need to provide all parameters -- no default values

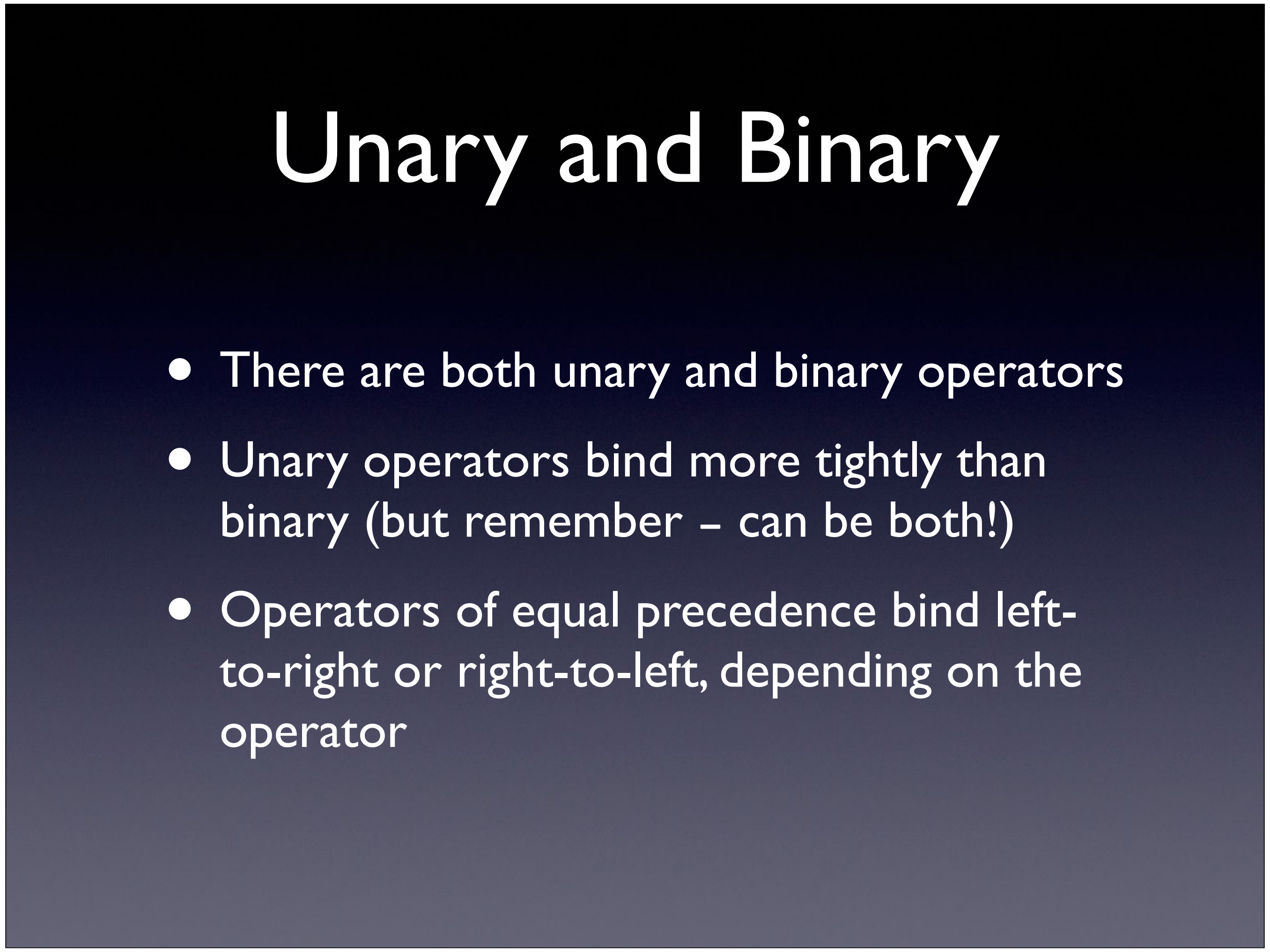


```
double CelsiusToFahrenheit(double t)
{
  double celsius;

  celsius = 9.0 * t / 5.0 + 32.0;
  return celsius;
}
```

remember values have types...

Mathematical Operators

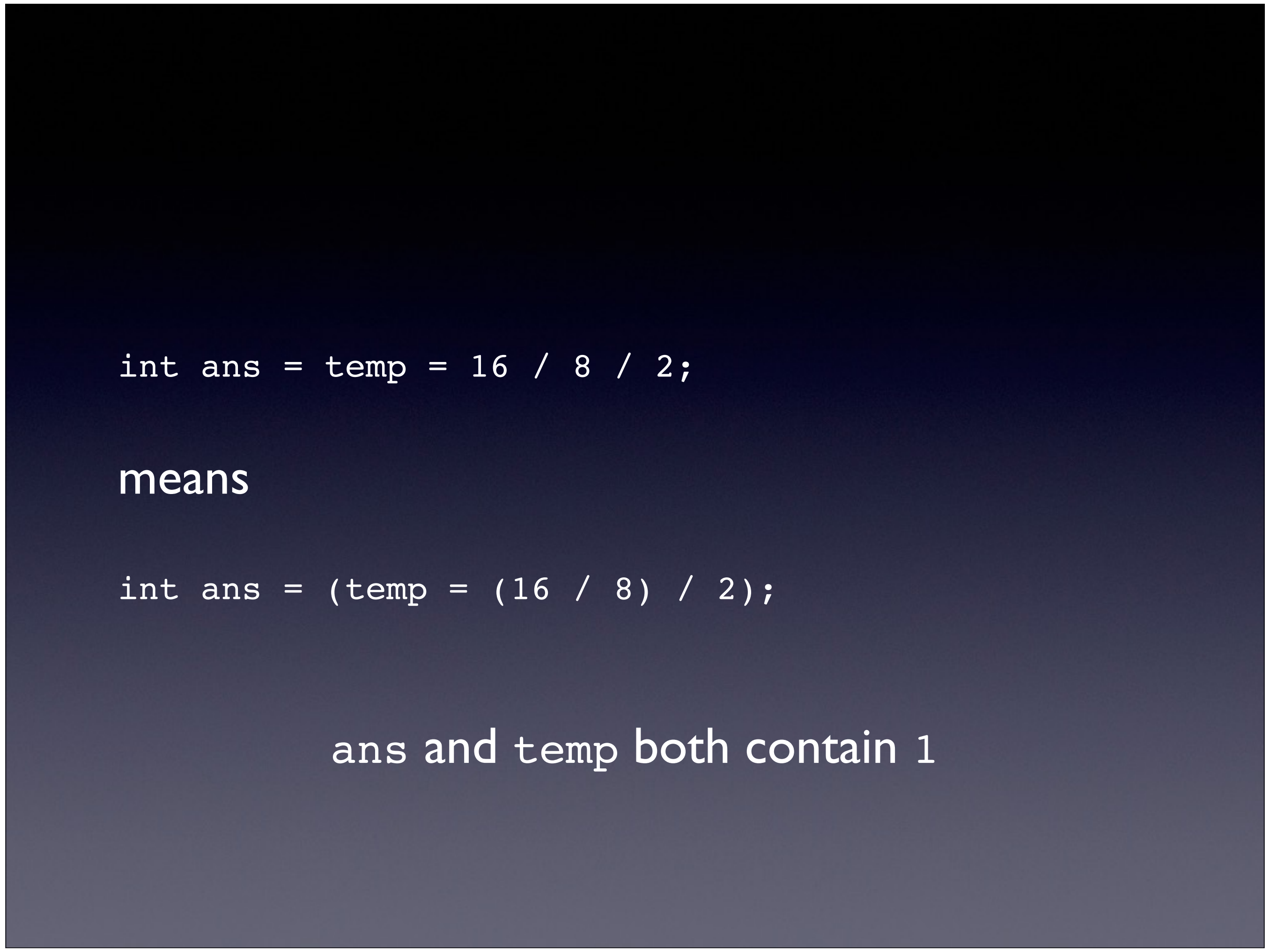
- C uses standard mathematical operators
- Work on one or two values (types will be promoted as necessary)
- These values can be from variables, functions, parameters etc.
- Compiler knows about *precedence* and *associativity*



Unary and Binary

- There are both unary and binary operators
- Unary operators bind more tightly than binary (but remember – can be both!)
- Operators of equal precedence bind left-to-right or right-to-left, depending on the operator

monadic, and dyadic



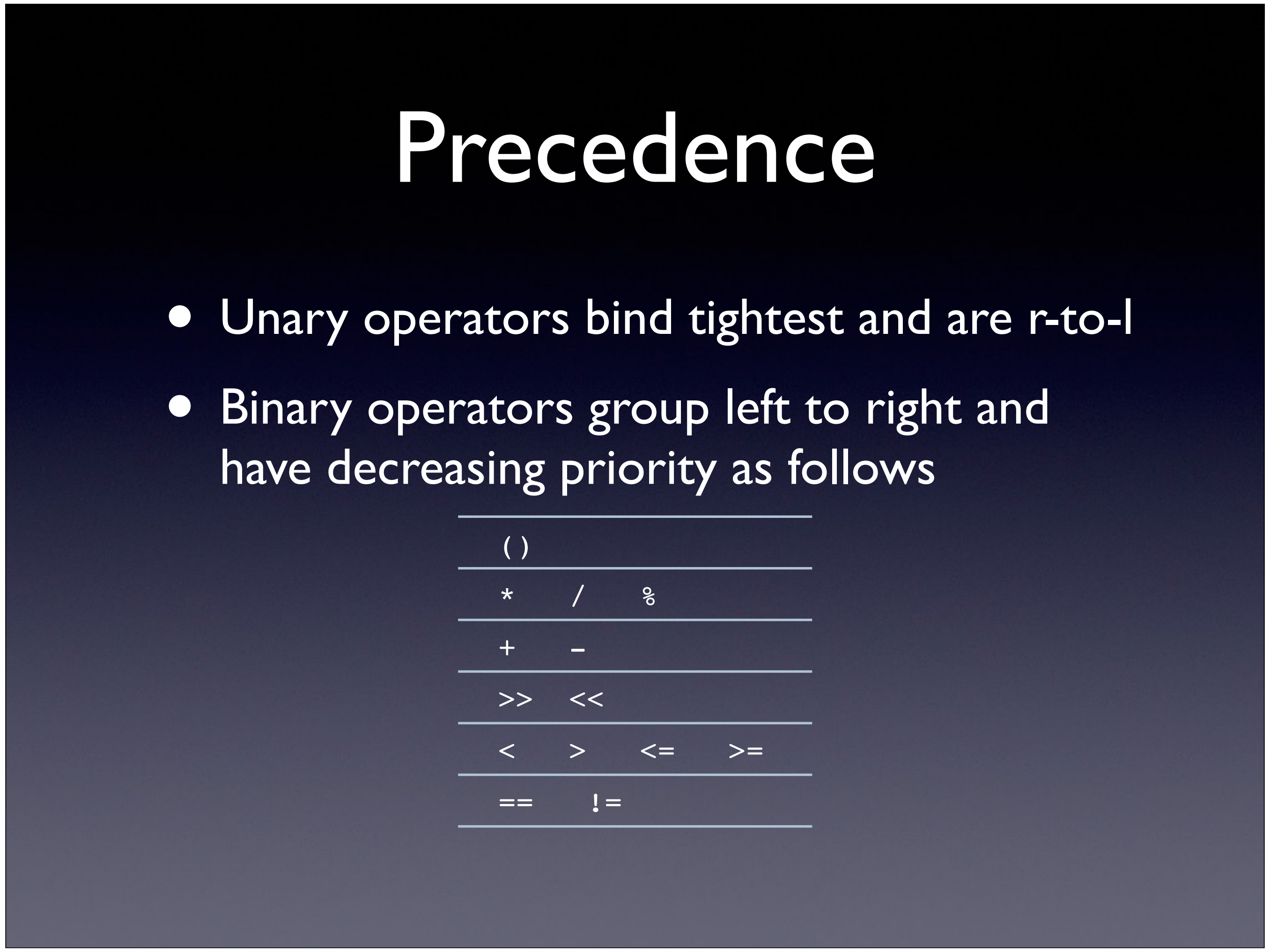
```
int ans = temp = 16 / 8 / 2;
```

means

```
int ans = (temp = (16 / 8) / 2);
```

`ans` and `temp` both contain `1`

/ binds left to right
= binds right to left



Precedence

- Unary operators bind tightest and are r-to-l
- Binary operators group left to right and have decreasing priority as follows

`( )`
`*  /  %`
`+  -`
`>>  <<`
`<  >  <=  >=`
`==  !=`

Not a complete list

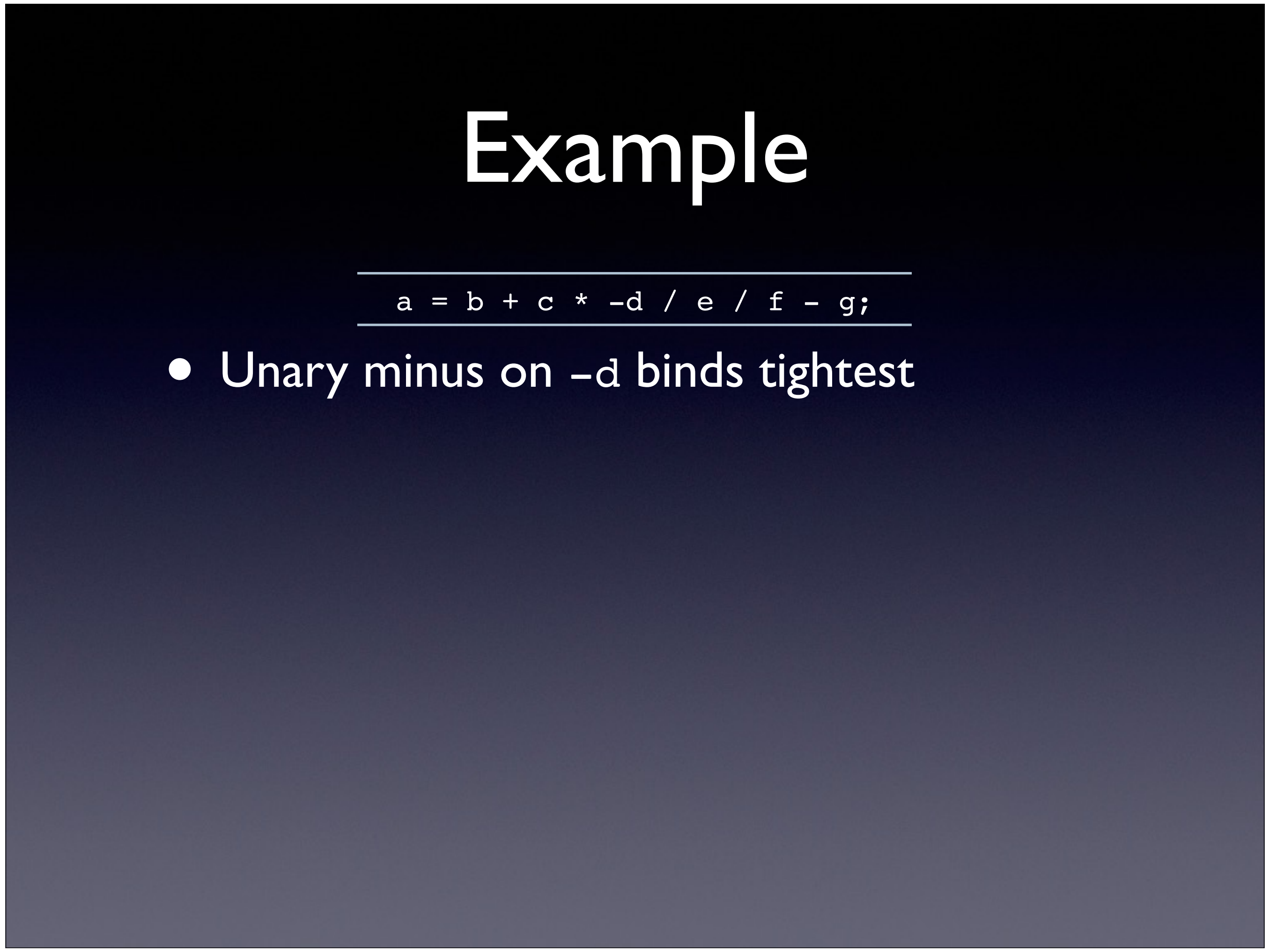
Assignment Precedence

- Assignment operators bind with the lowest priority
- Group right to left

=	+=	-=	>>=	<<=	...

Example

```
a = b + c * -d / e / f - g;
```



Example

```
a = b + c * -d / e / f - g;
```

- Unary minus on `-d` binds tightest

Example

```
a = b + c * -d / e / f - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right

Example

```
a = b + (((c * (-d)) / e) / f) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right

Example

```
a = b + (((c * (-d)) / e) / f) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right
- Next, `+` and `-` are equal but lower priority and bind left-to-right so addition is done first

Example

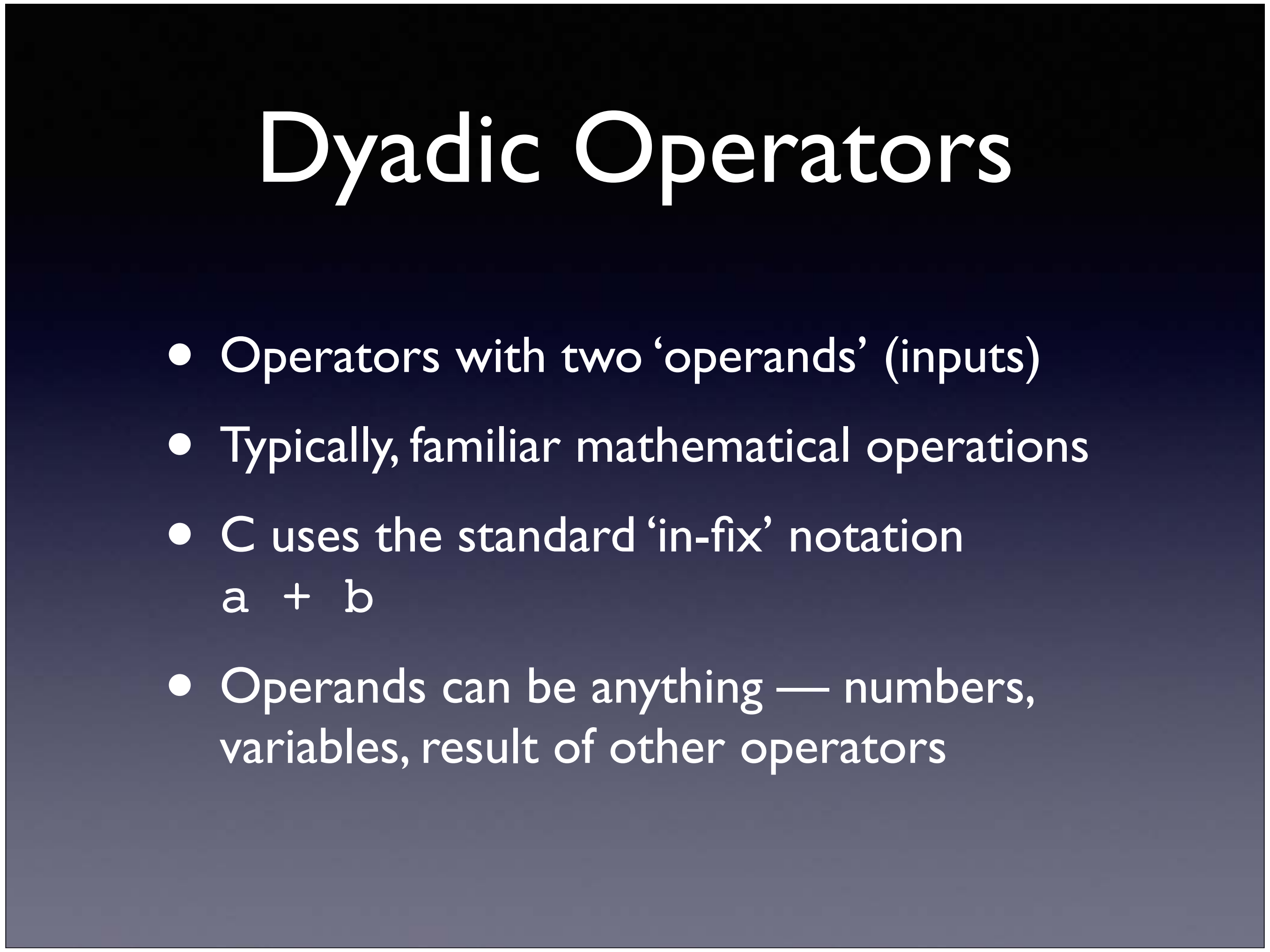
```
a = (b + (((c * (-d)) / e) / f)) - g;
```

- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right
- Next, + and – are equal but lower priority and bind left-to-right so addition is done first

Example

```
a = (b + (((c * (-d)) / e) / f)) - g;
```

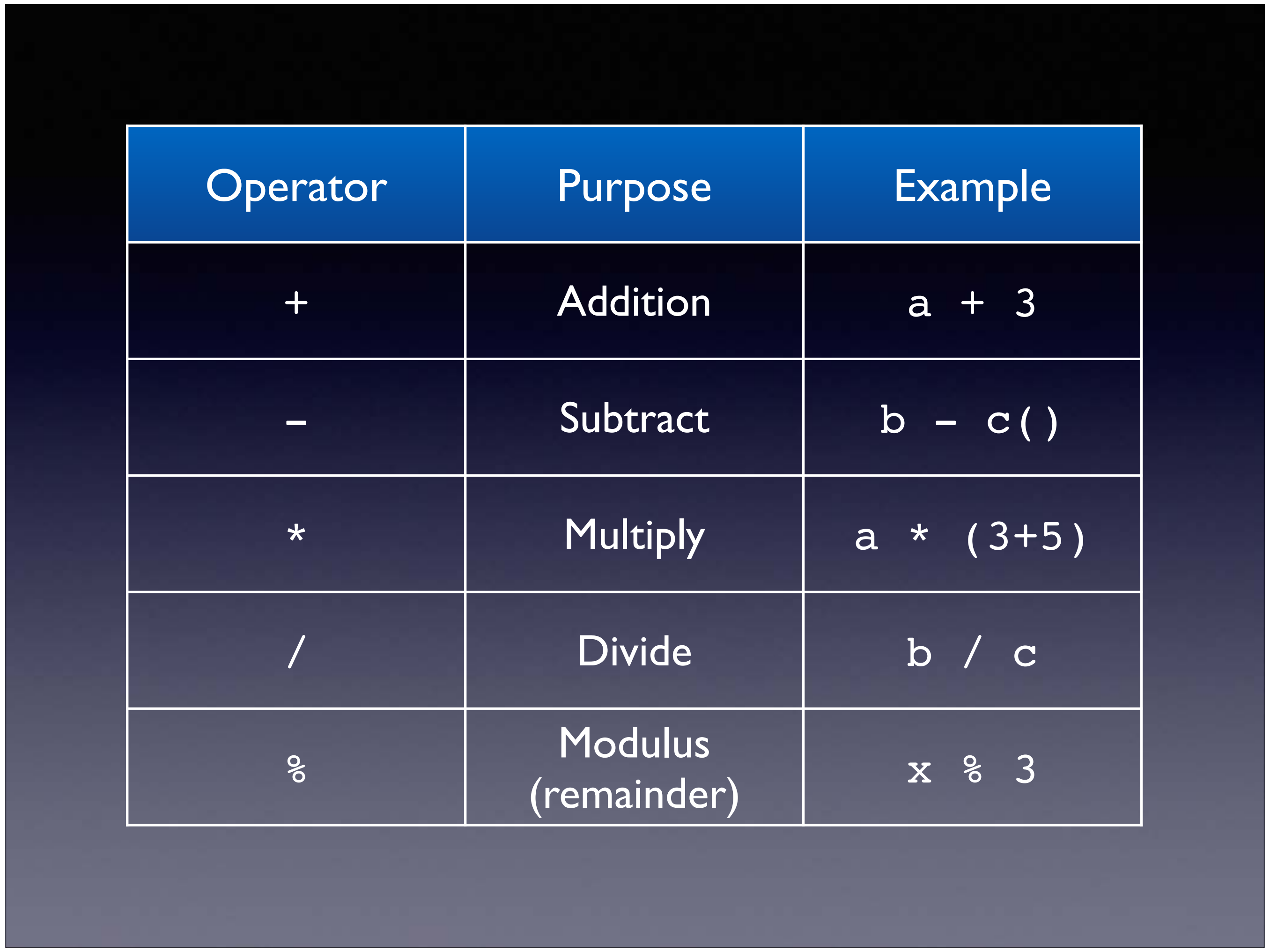
- Unary minus on `-d` binds tightest
- `*` and `/` are equal but bind left-to-right
- Next, + and - are equal but lower priority and bind left-to-right so addition is done first
- Finally, the result is assigned to `a`



Dyadic Operators

- Operators with two ‘operands’ (inputs)
- Typically, familiar mathematical operations
- C uses the standard ‘in-fix’ notation
  `a + b`
- Operands can be anything — numbers, variables, result of other operators

Dyadic, also called binary operators which shouldn’t be confused with the binary number system

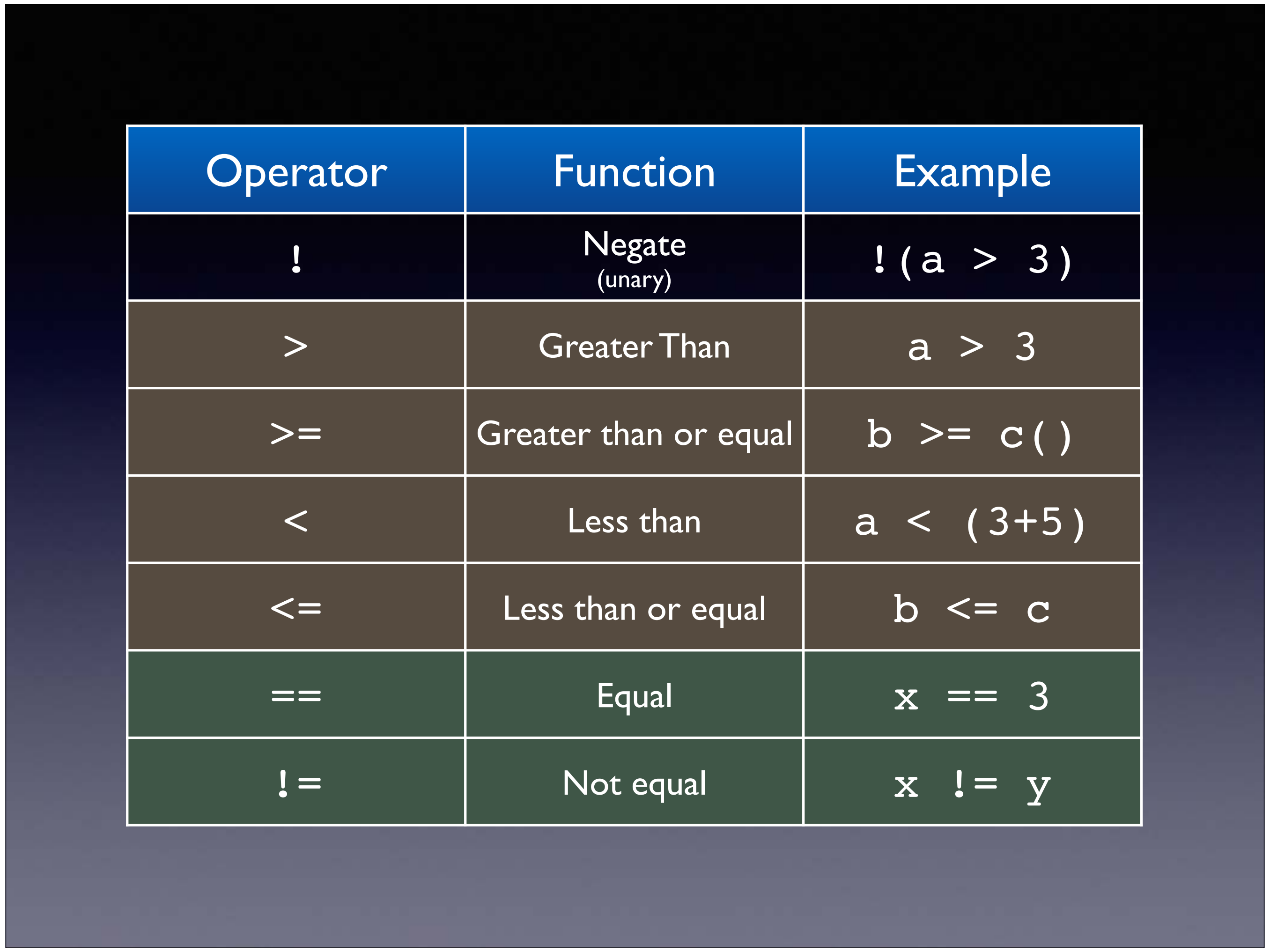


Operator	Purpose	Example
+	Addition	a + 3
-	Subtract	b - c()
*	Multiply	a * (3+5)
/	Divide	b / c
%	Modulus (remainder)	x % 3

Go build a function to convert Centigrade to Fahrenheit

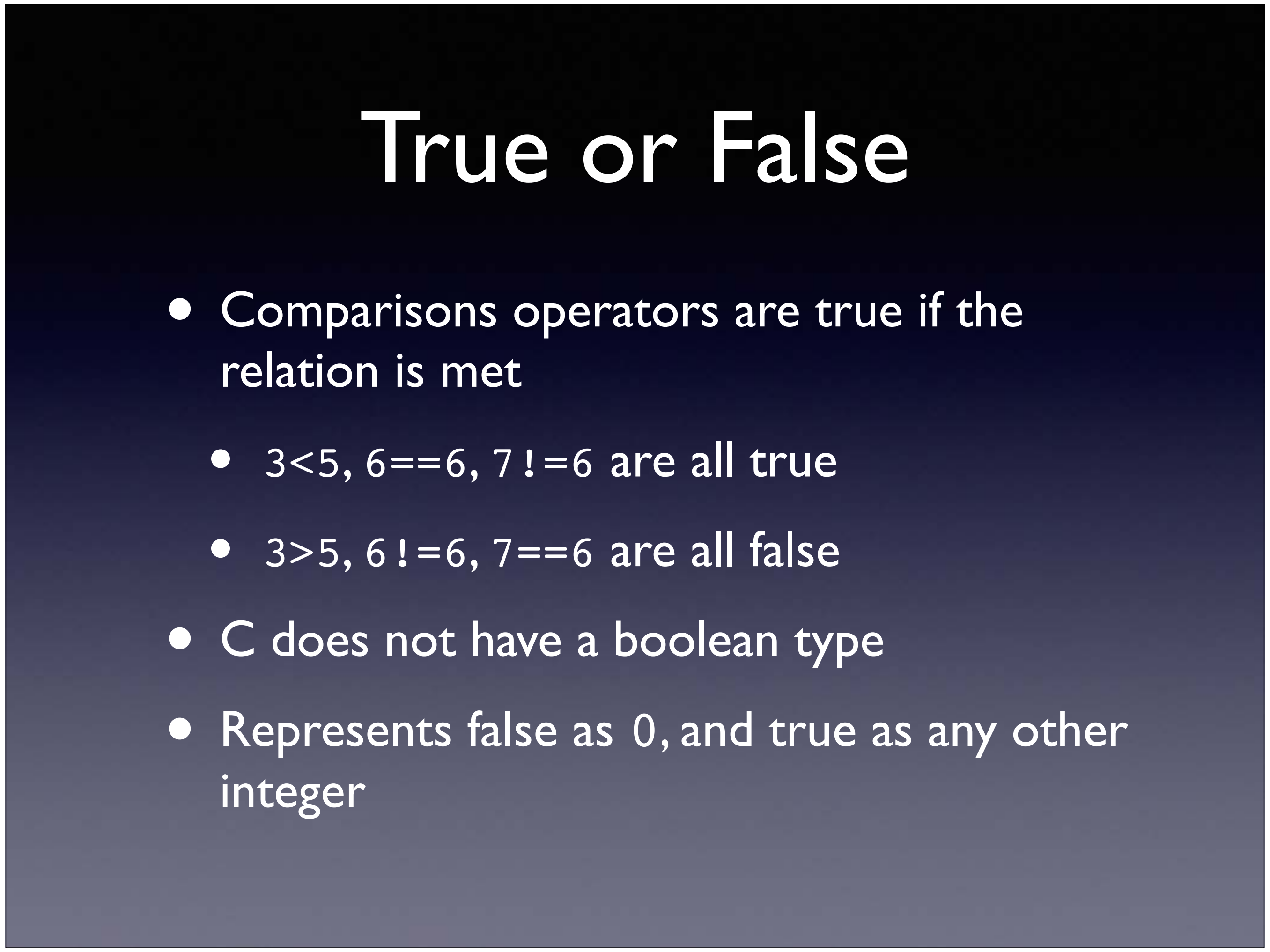
Comparison Operators

- Arithmetic operators are not the only dyadic operators C provides
- Also provides relational and equality operators
- These allow you to compare two values



Operator	Function	Example
`!`	Negate (unary)	`!(a > 3)`
`>`	Greater Than	`a > 3`
`>=`	Greater than or equal	`b >= c()`
`<`	Less than	`a < (3+5)`
`<=`	Less than or equal	`b <= c`
`==`	Equal	`x == 3`
`!=`	Not equal	`x != y`

Background colour denotes precedence groups, Highest precedence is at the top, lowest at the bottom



True or False

- Comparisons operators are true if the relation is met
  - `3<5`, `6==6`, `7!=6` are all true
  - `3>5`, `6!=6`, `7==6` are all false
- C does not have a boolean type
- Represents false as `0`, and true as any other integer

Go and show this with printf

Program Flow

- Programs usual flow from statement to statement
- The power of programming is when the programmer is in control
- Calling Functions allow us to reuse code
- But we can also do things *conditionally*

Conditional Execution

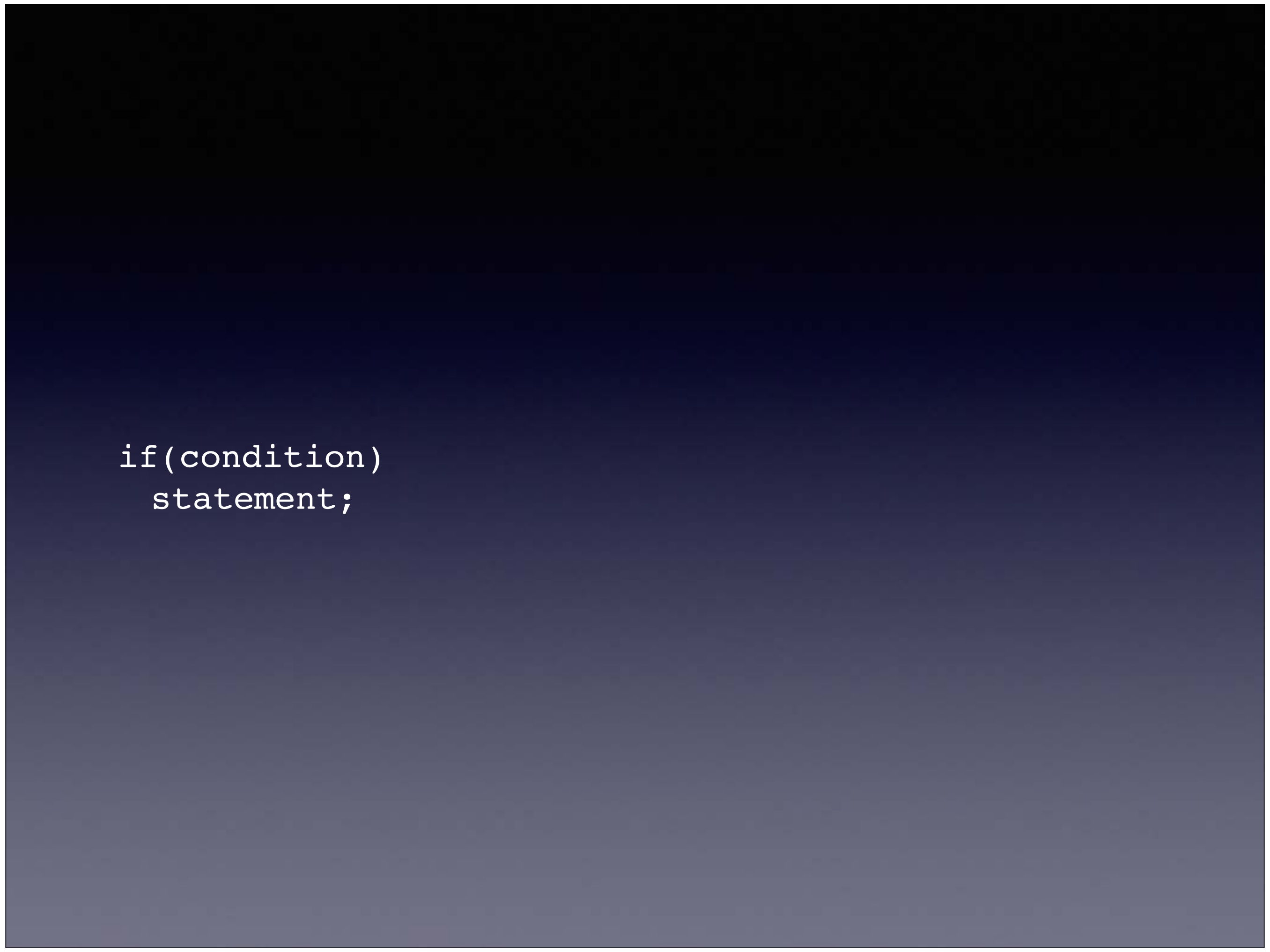
- Only execute some code *if* a particular condition is true
- What do we mean by condition?
- Anything that can be converted into a true or false value
- Any of the operators we’ve just seen!

Conditionals

- The condition can be based on
  - Values (not that useful on their own)
  - Value of a variable
  - Value returned by a function

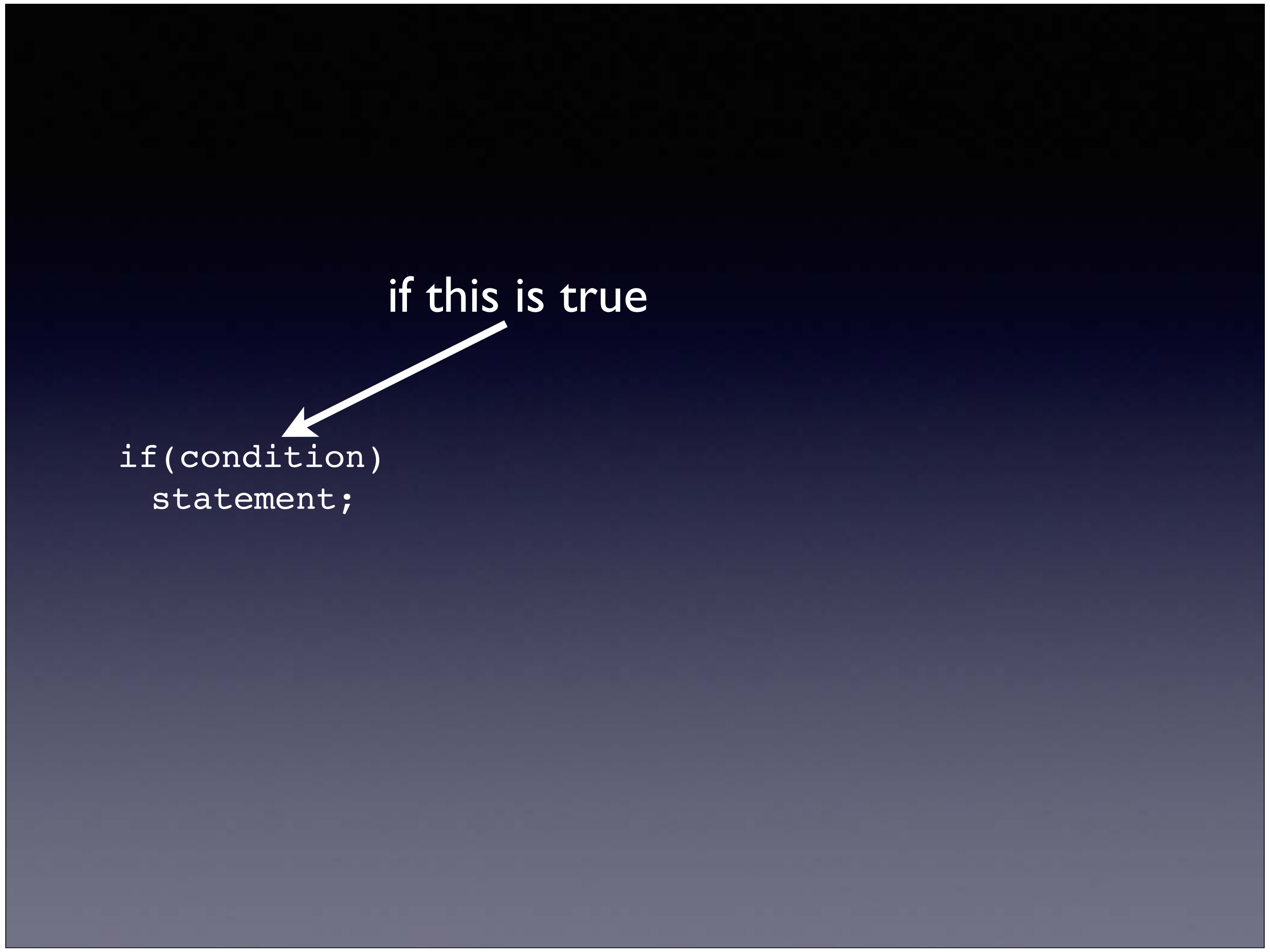
The `if` statement

- C’s `if` statement is used to express a decision
- If the condition is true, then execute the next statement
- If the condition is false, then don’t execute the next statement



```
if(condition)
  statement;
```

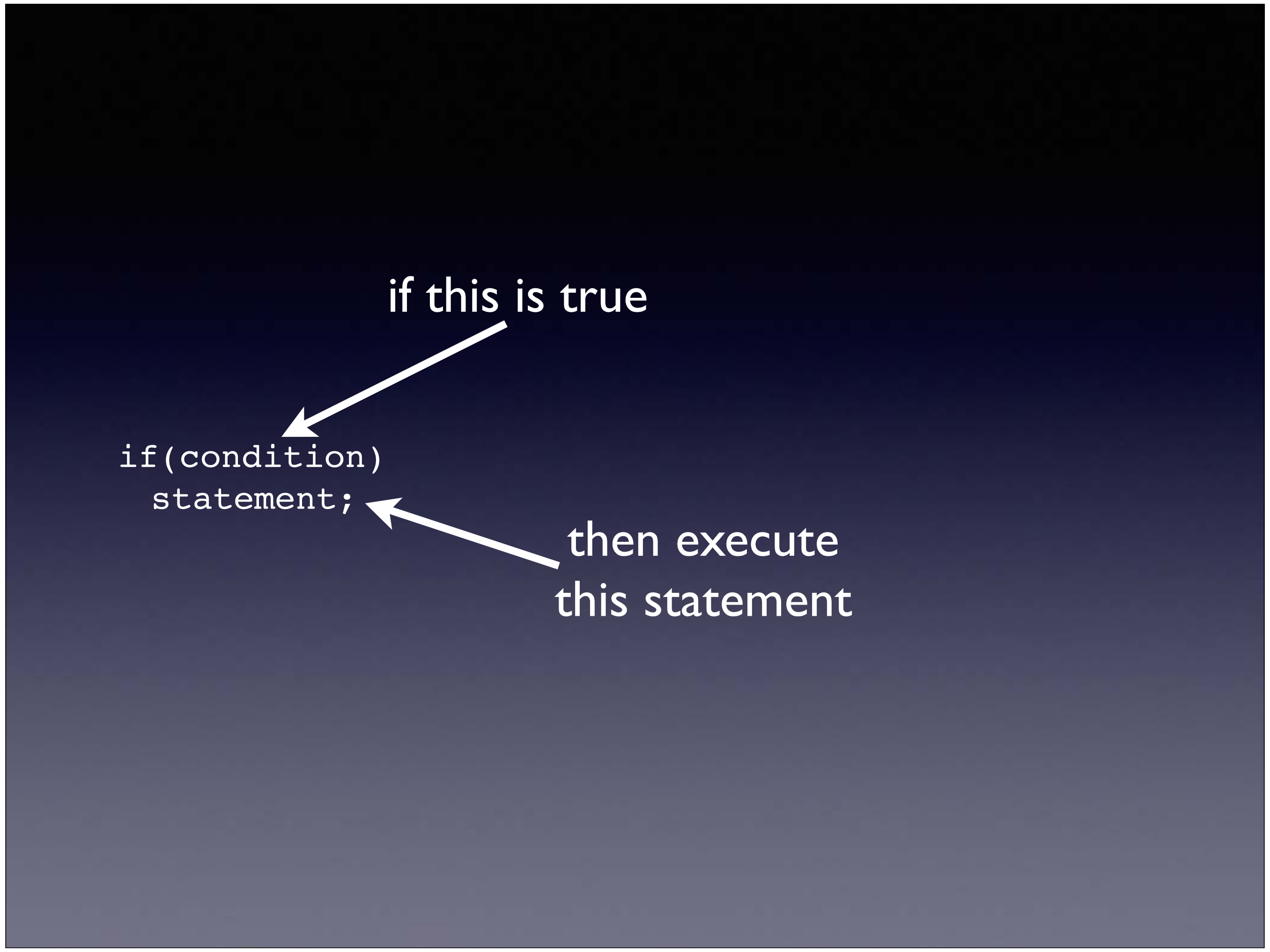
the statement is executed only if condition is true



if this is true

```
if(condition)
  statement;
```

the statement is executed only if condition is true

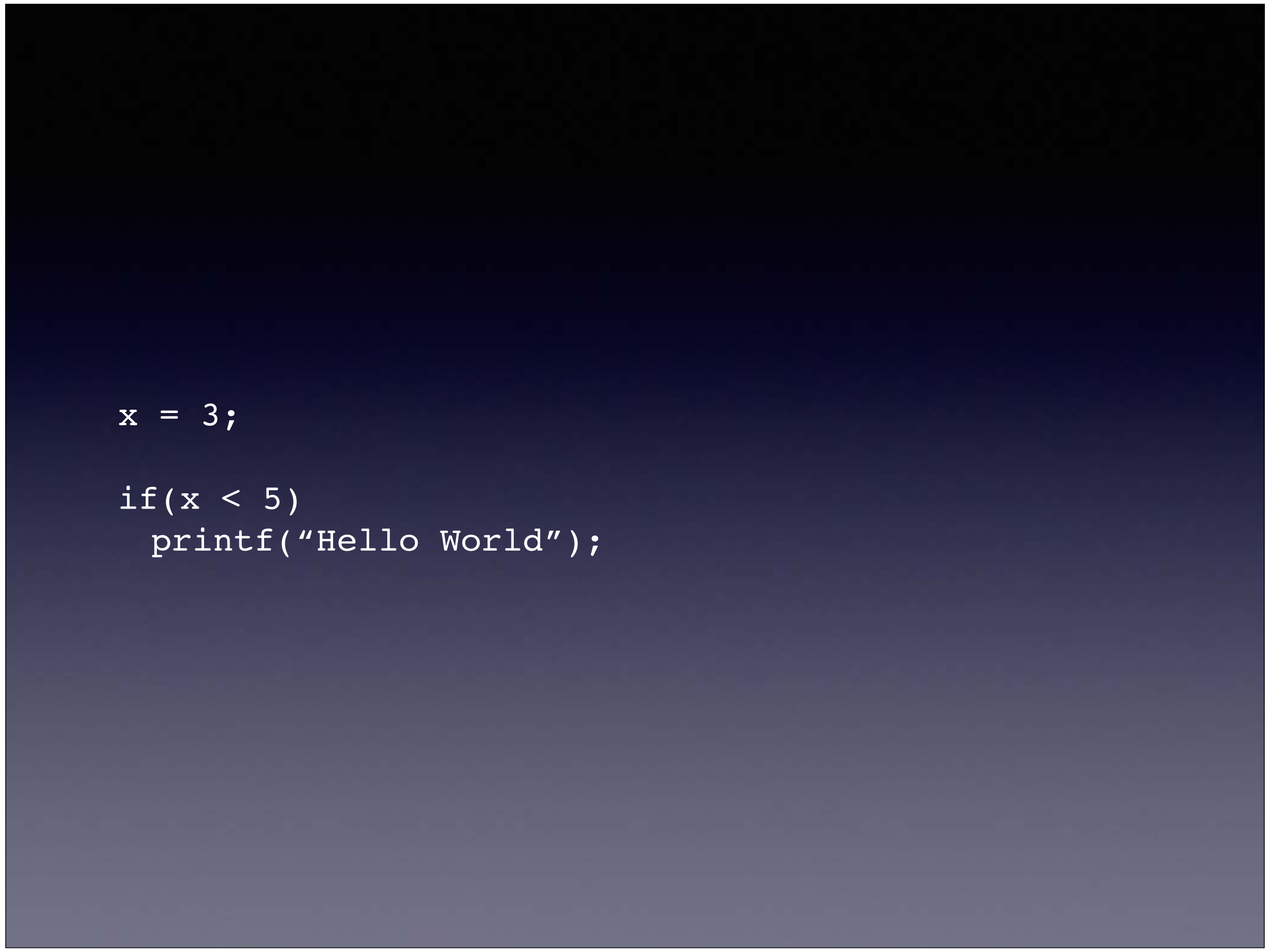


if this is true

```
if(condition)
  statement;
```

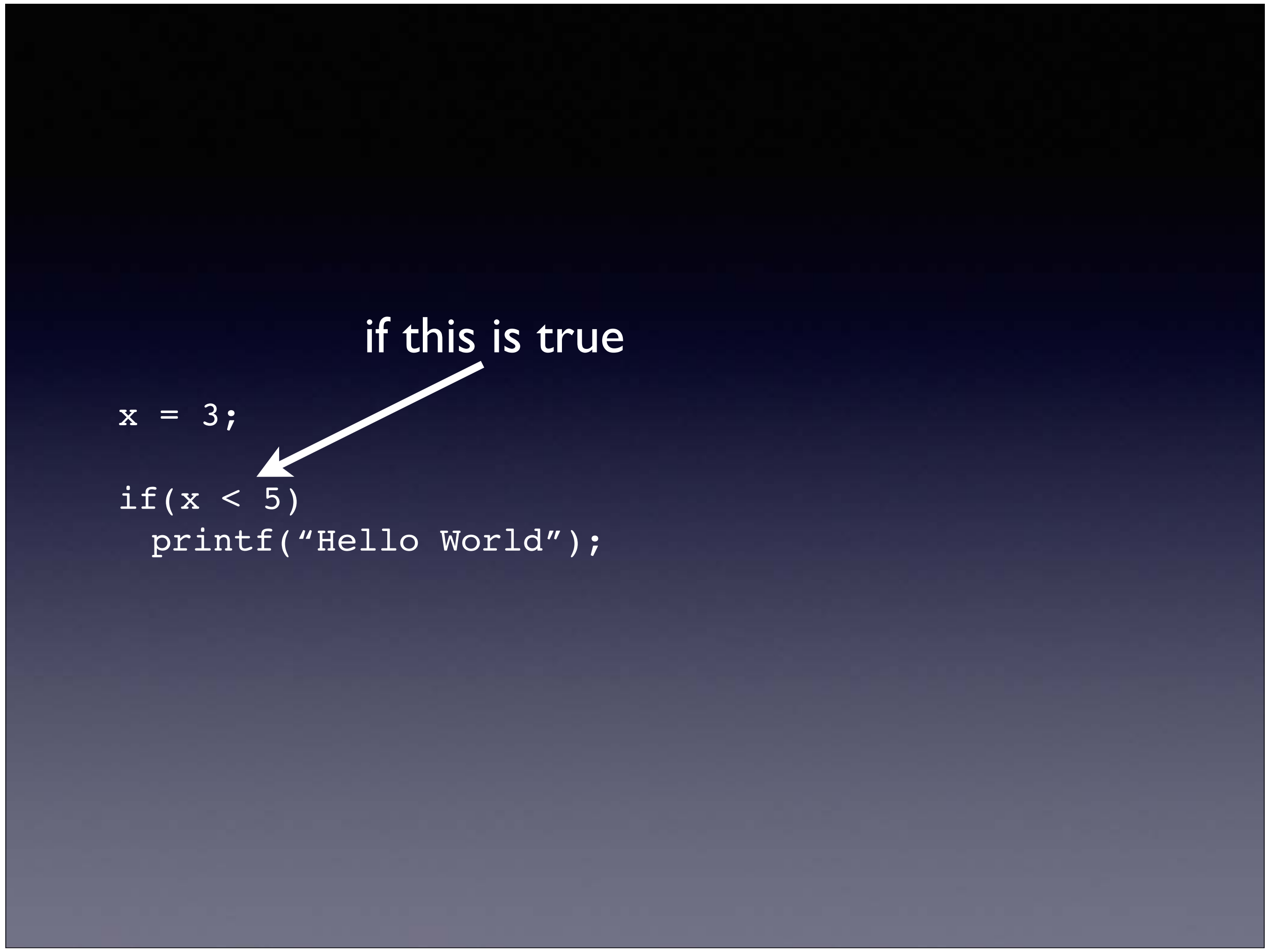
then execute
this statement

the statement is executed only if condition is true



```
x = 3;

if(x < 5)
  printf(“Hello World”);
```

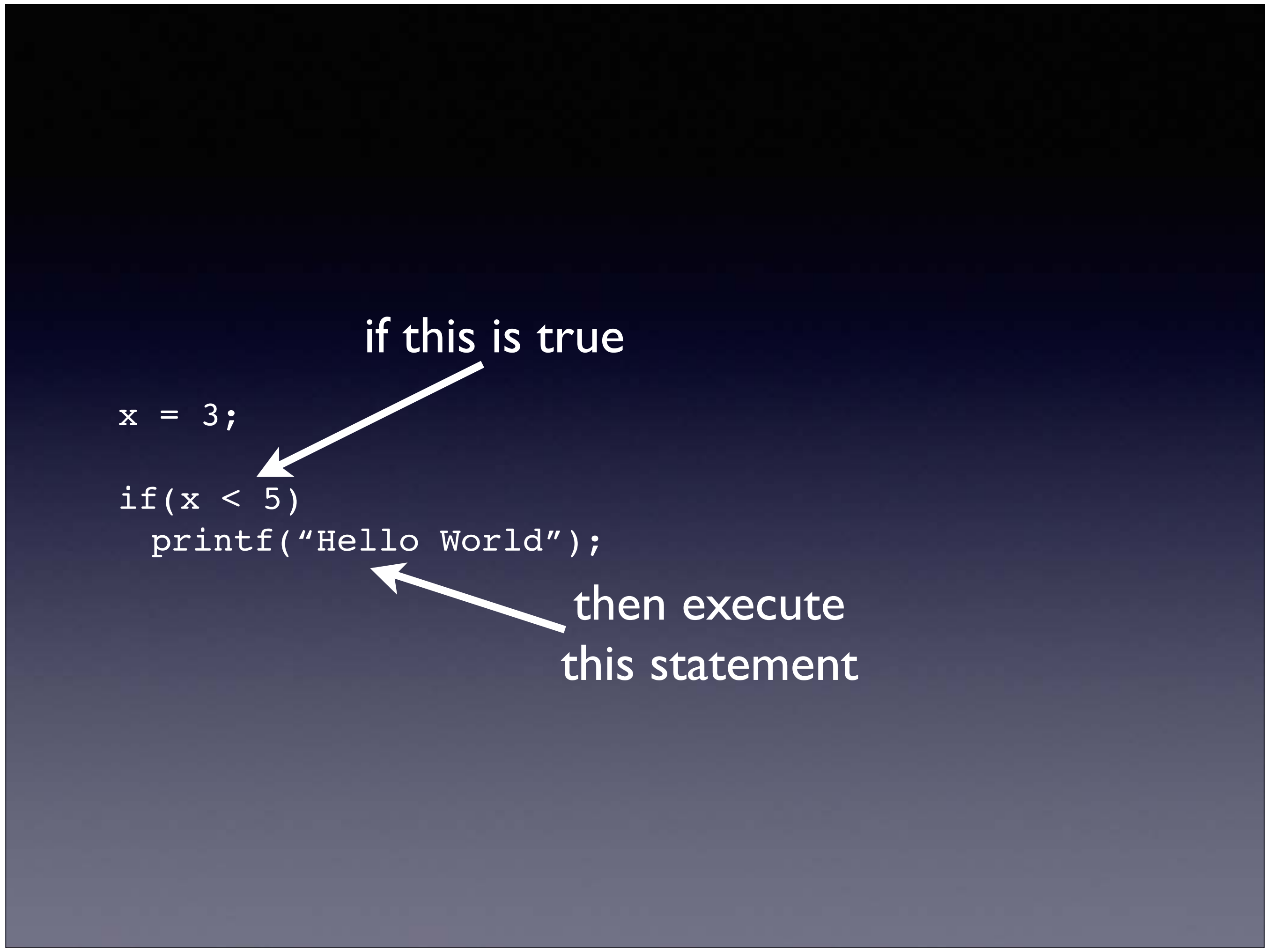
the statement is executed only if condition is true



if this is true

```
x = 3;

if(x < 5)
  printf(“Hello World”);
```

the statement is executed only if condition is true

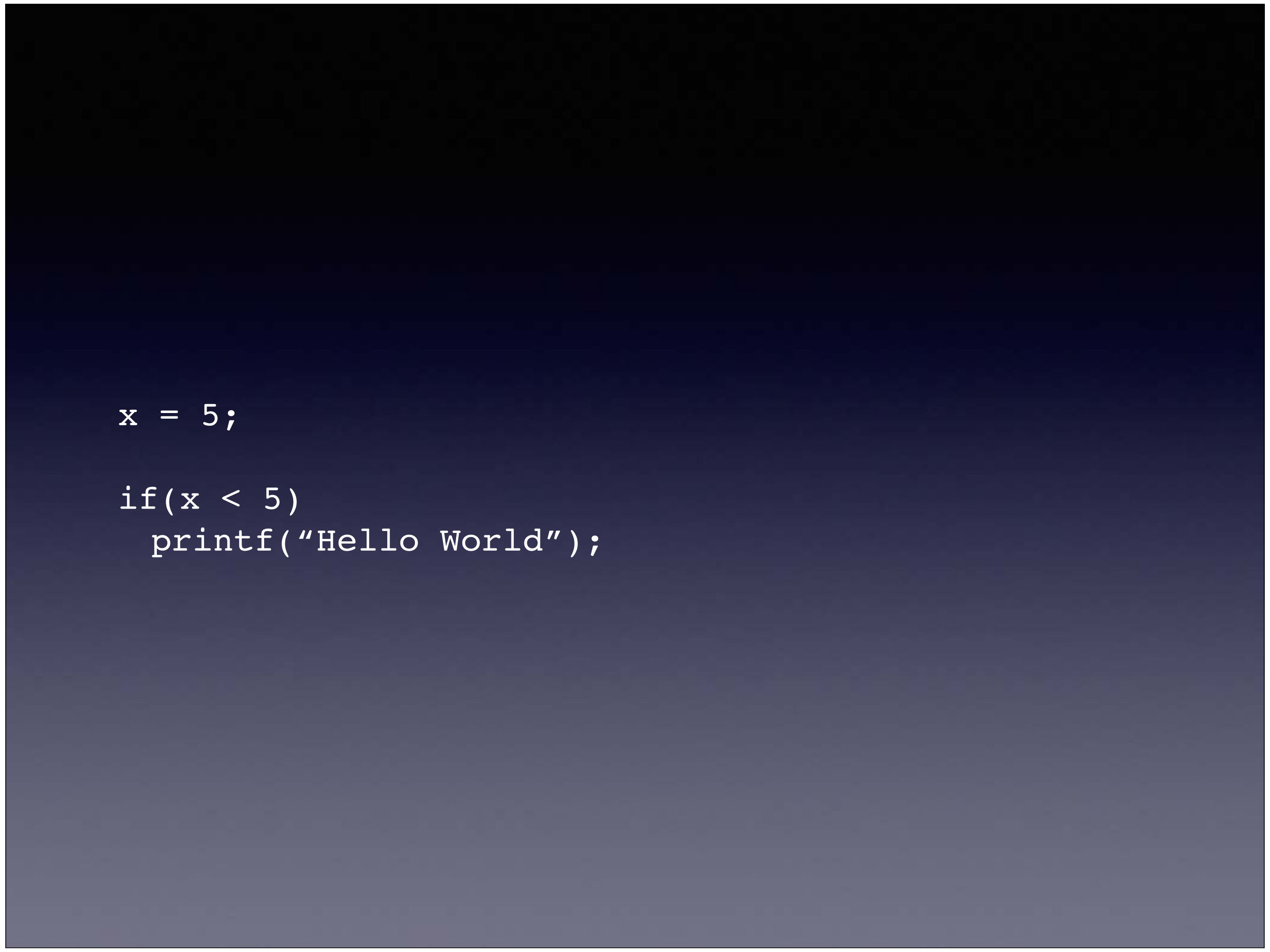


if this is true

```
x = 3;

if(x < 5)
  printf("Hello World");
```

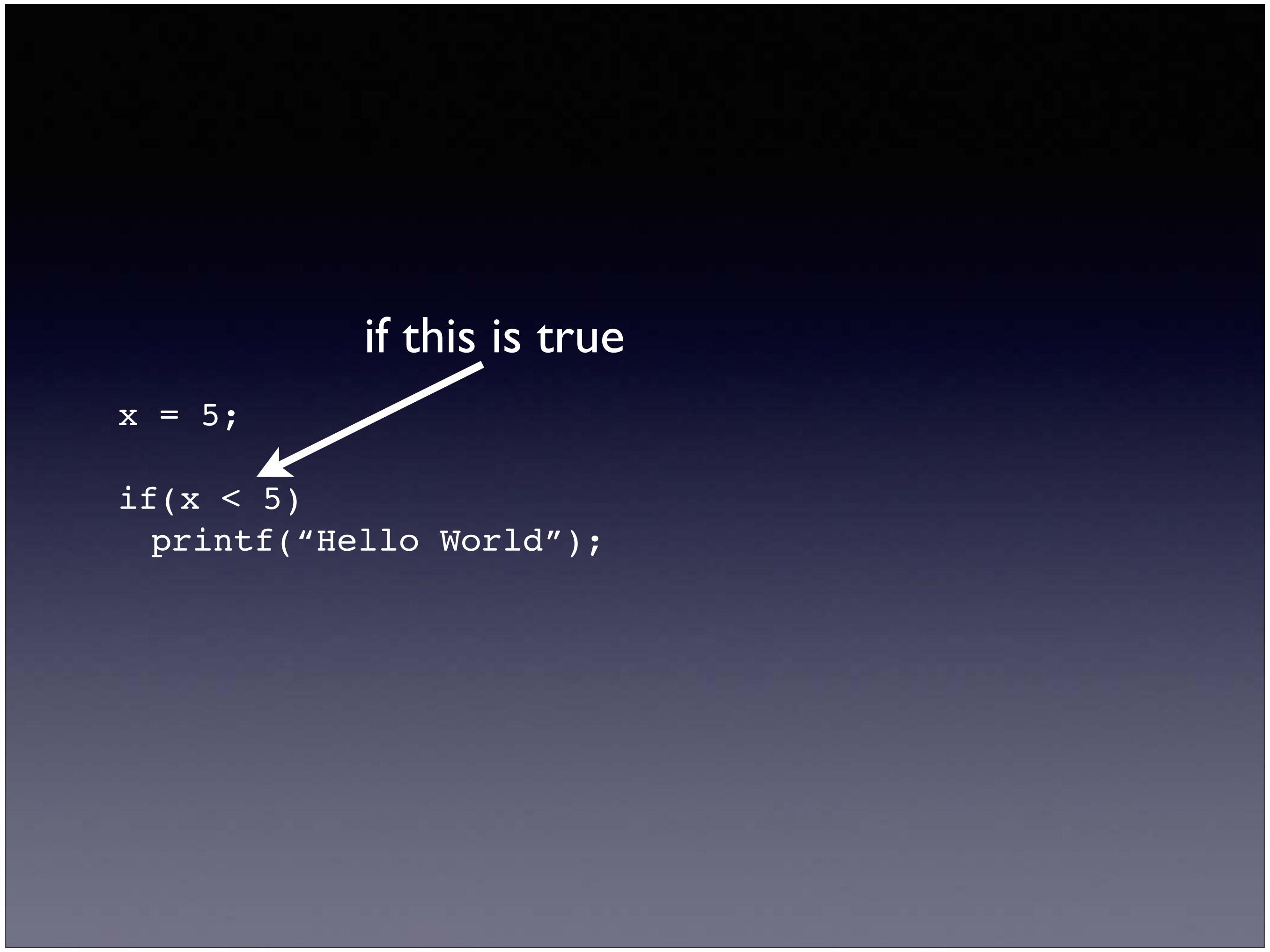
then execute
this statement

the statement is executed only if condition is true



```
x = 5;

if(x < 5)
  printf("Hello World");
```

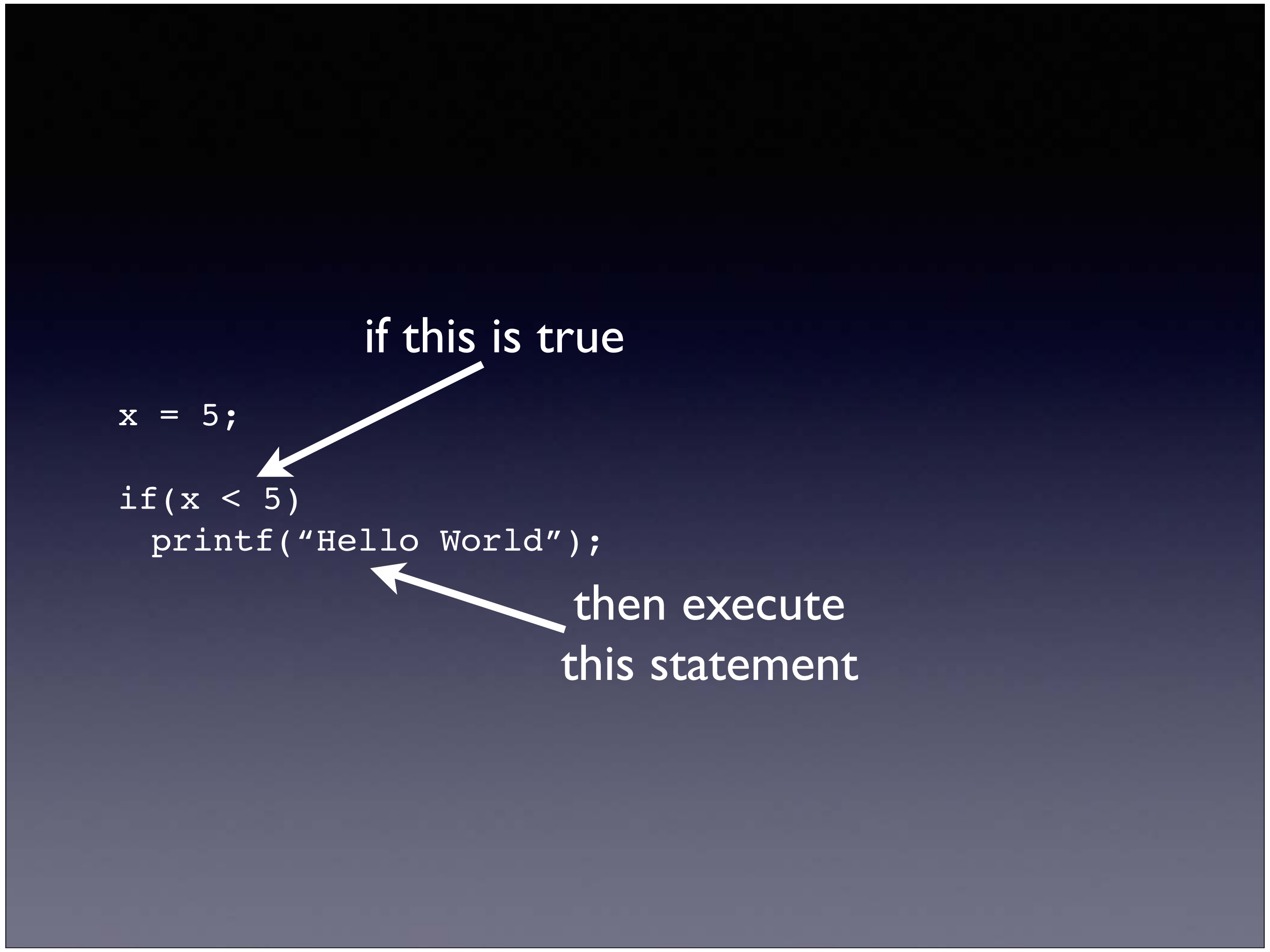
the statement is executed only if condition is true



if this is true

```
x = 5;

if(x < 5)
  printf(“Hello World”);
```

the statement is executed only if condition is true

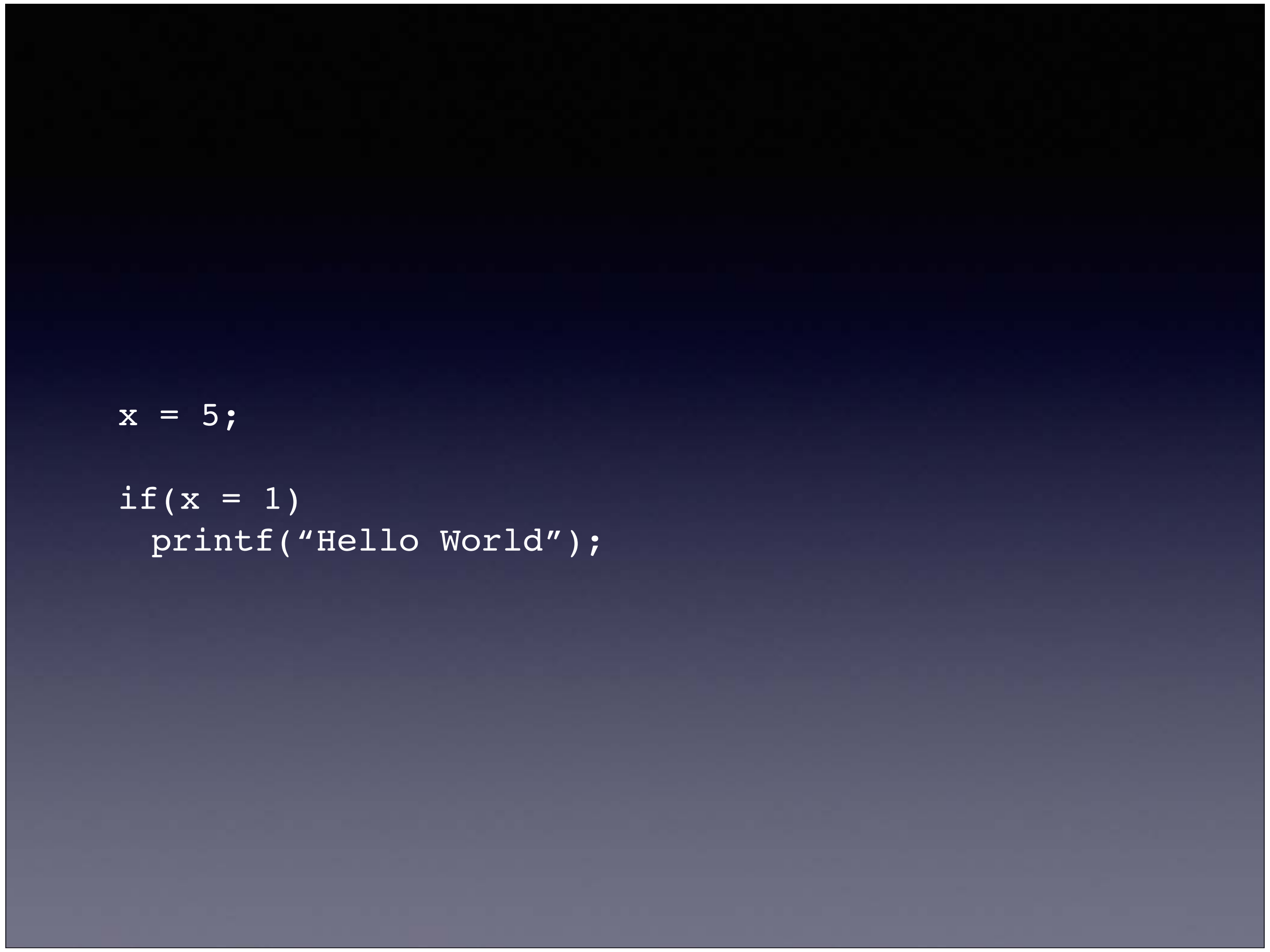


if this is true

```
x = 5;

if(x < 5)
  printf(“Hello World”);
```

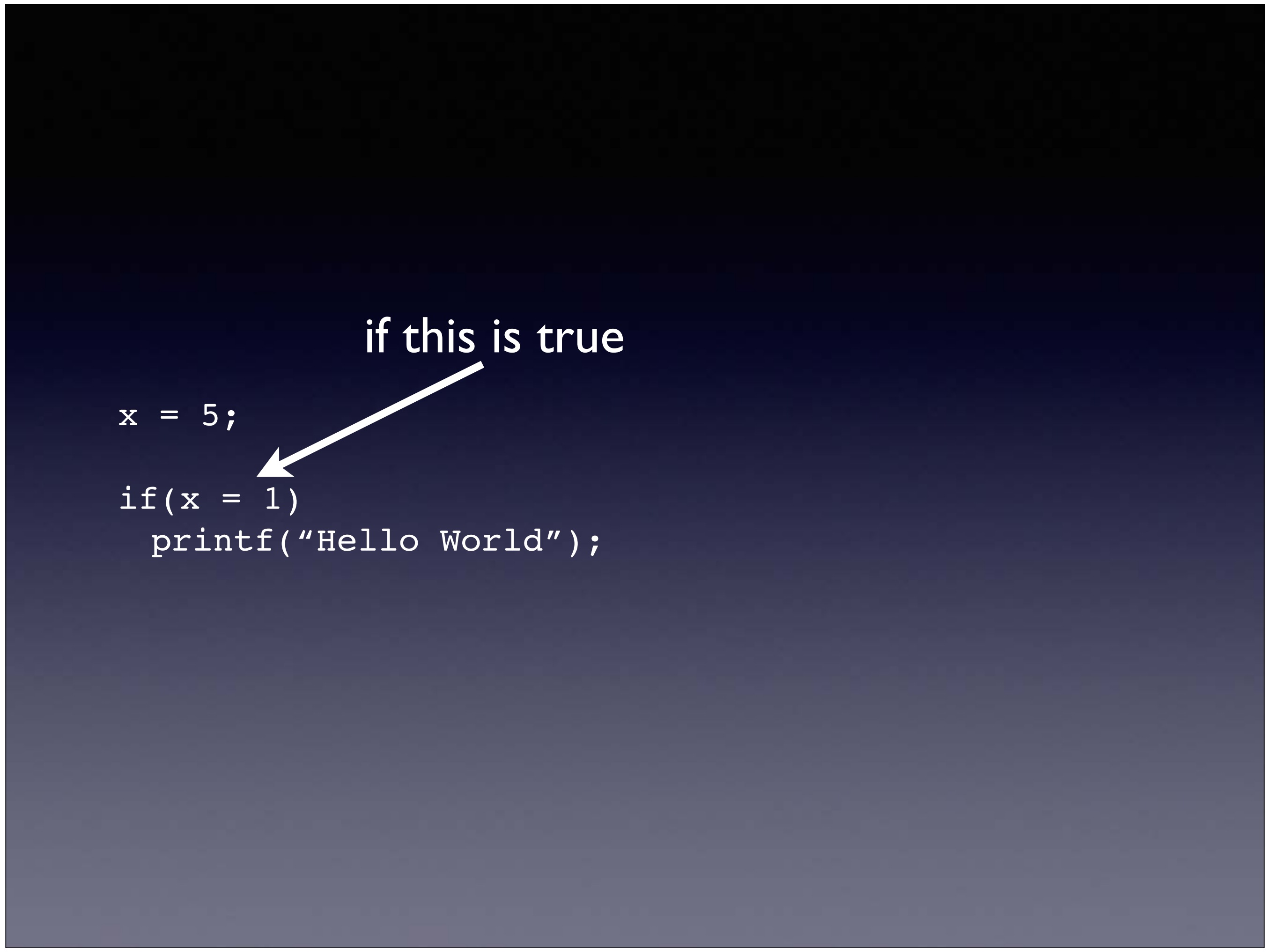
then execute
this statement

the statement is executed only if condition is true



```
x = 5;

if(x = 1)
  printf(“Hello World”);
```

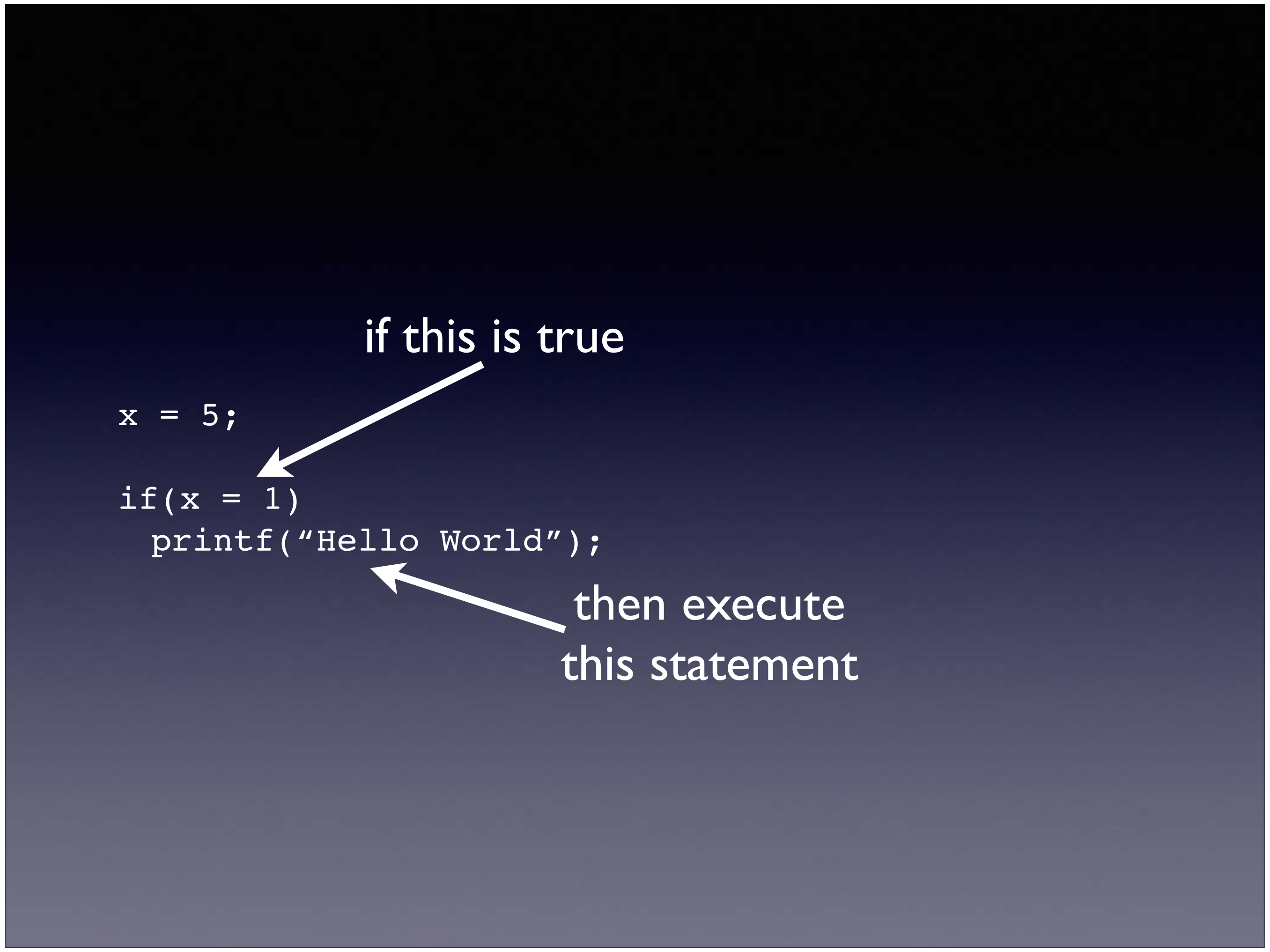
we’re not testing equality here -- we are setting x to equal 1, the result of that expression is 1 which is interpreted as true
 you want...



if this is true

```
x = 5;

if(x = 1)
  printf(“Hello World”);
```

we’re not testing equality here -- we are setting x to equal 1, the result of that expression is 1 which is interpreted as true
 you want...

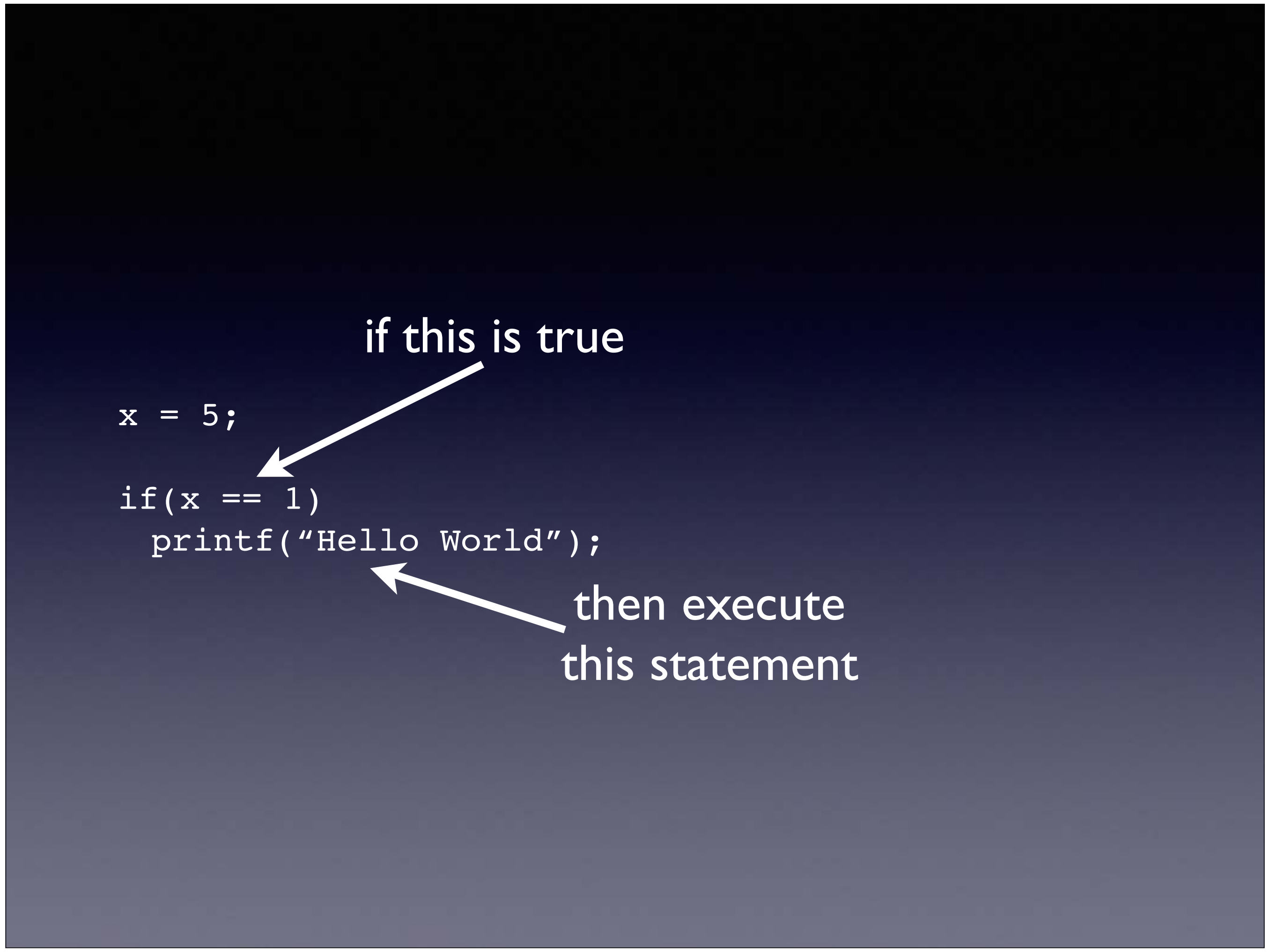


if this is true

```
x = 5;

if(x = 1)
  printf(“Hello World”);
```

then execute
this statement

we’re not testing equality here -- we are setting x to equal 1, the result of that expression is 1 which is interpreted as true
you want...

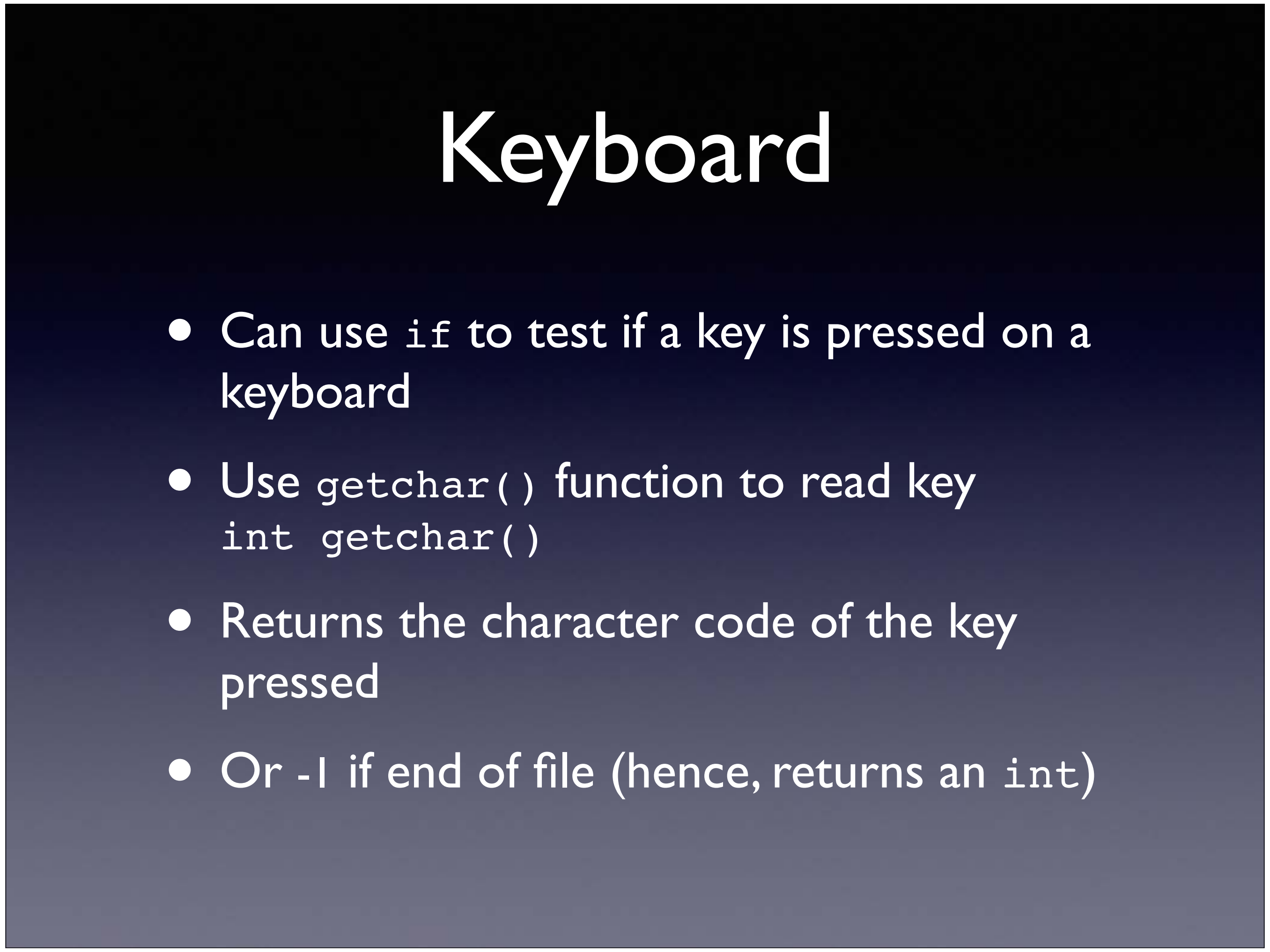


if this is true

```
x = 5;

if(x == 1)
  printf(“Hello World”);
```

then execute
this statement

here x does not equal 1 so it doesn’t print

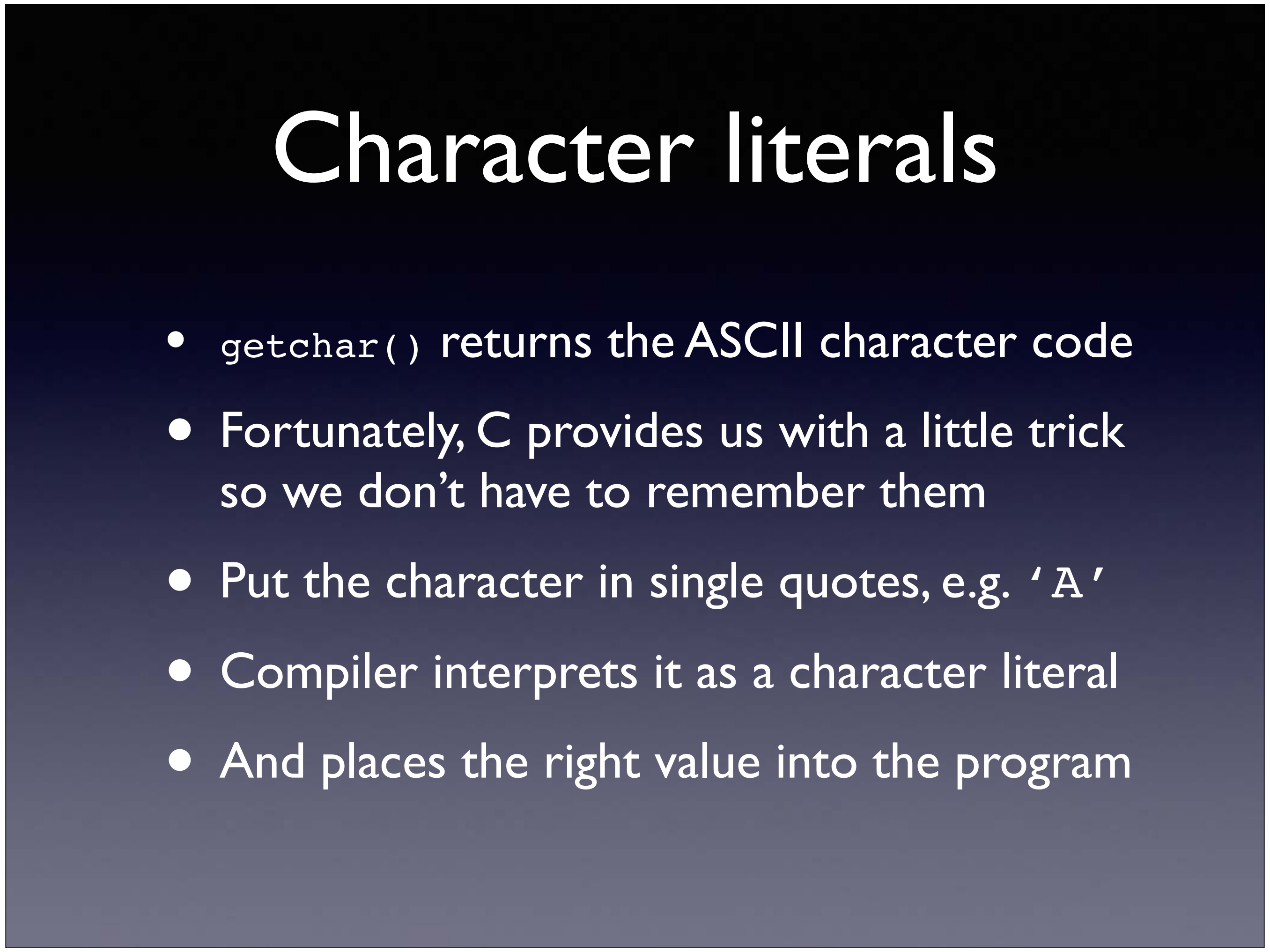


Keyboard

- Can use `if` to test if a key is pressed on a keyboard
- Use `getchar()` function to read key
  `int getchar()`
- Returns the character code of the key pressed
- Or -1 if end of file (hence, returns an `int`)

Press ENTER

- Slight problem…
- UNIX only returns keyboard input when a carriage return is pressed
- So we have to also press RETURN before we see our keypress



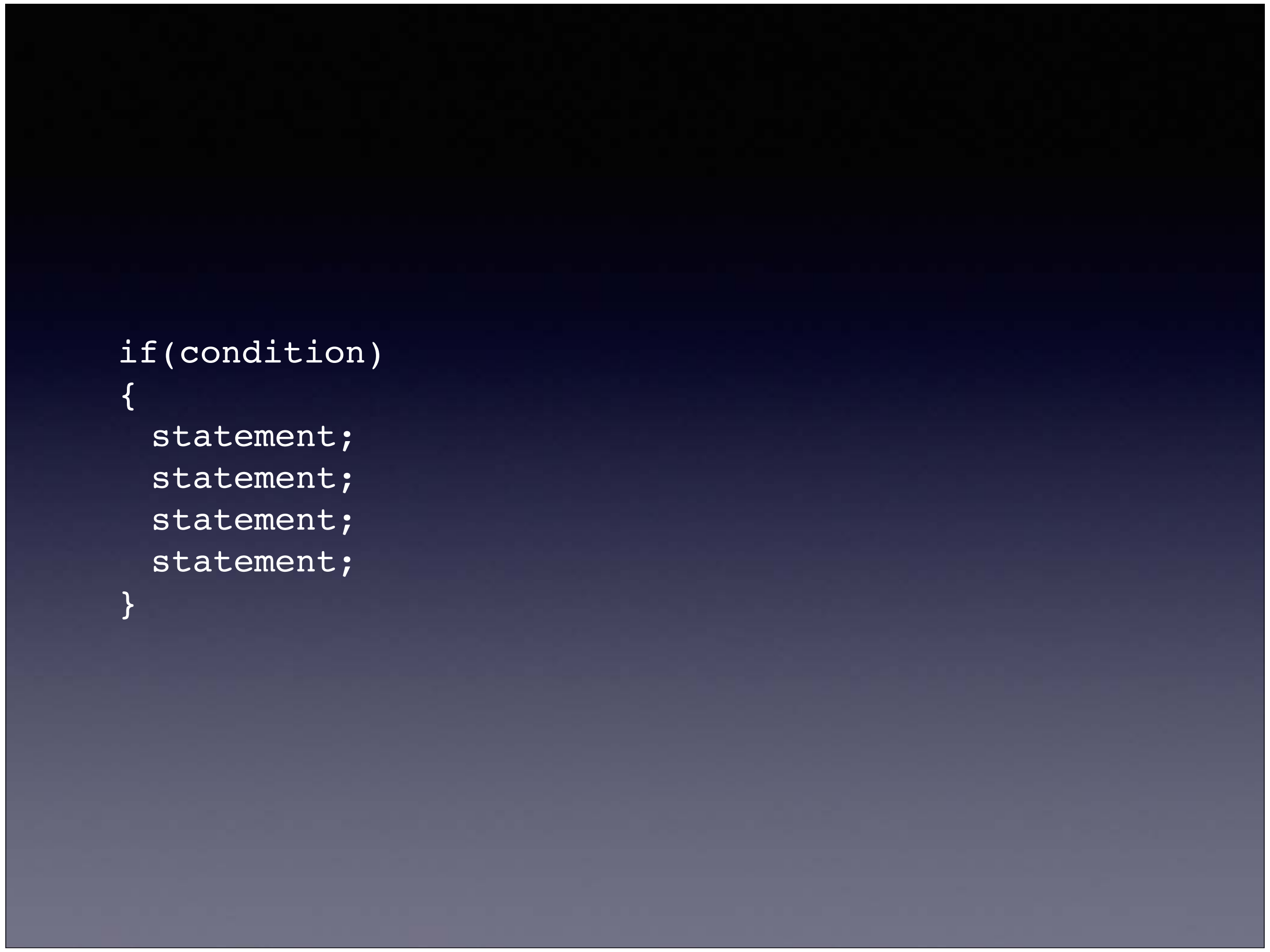
Character literals

- `getchar()` returns the ASCII character code
- Fortunately, C provides us with a little trick so we don’t have to remember them
- Put the character in single quotes, e.g. `‘A’`
- Compiler interprets it as a character literal
- And places the right value into the program

In this case, the number 65 (for ‘A’)
DEMO!

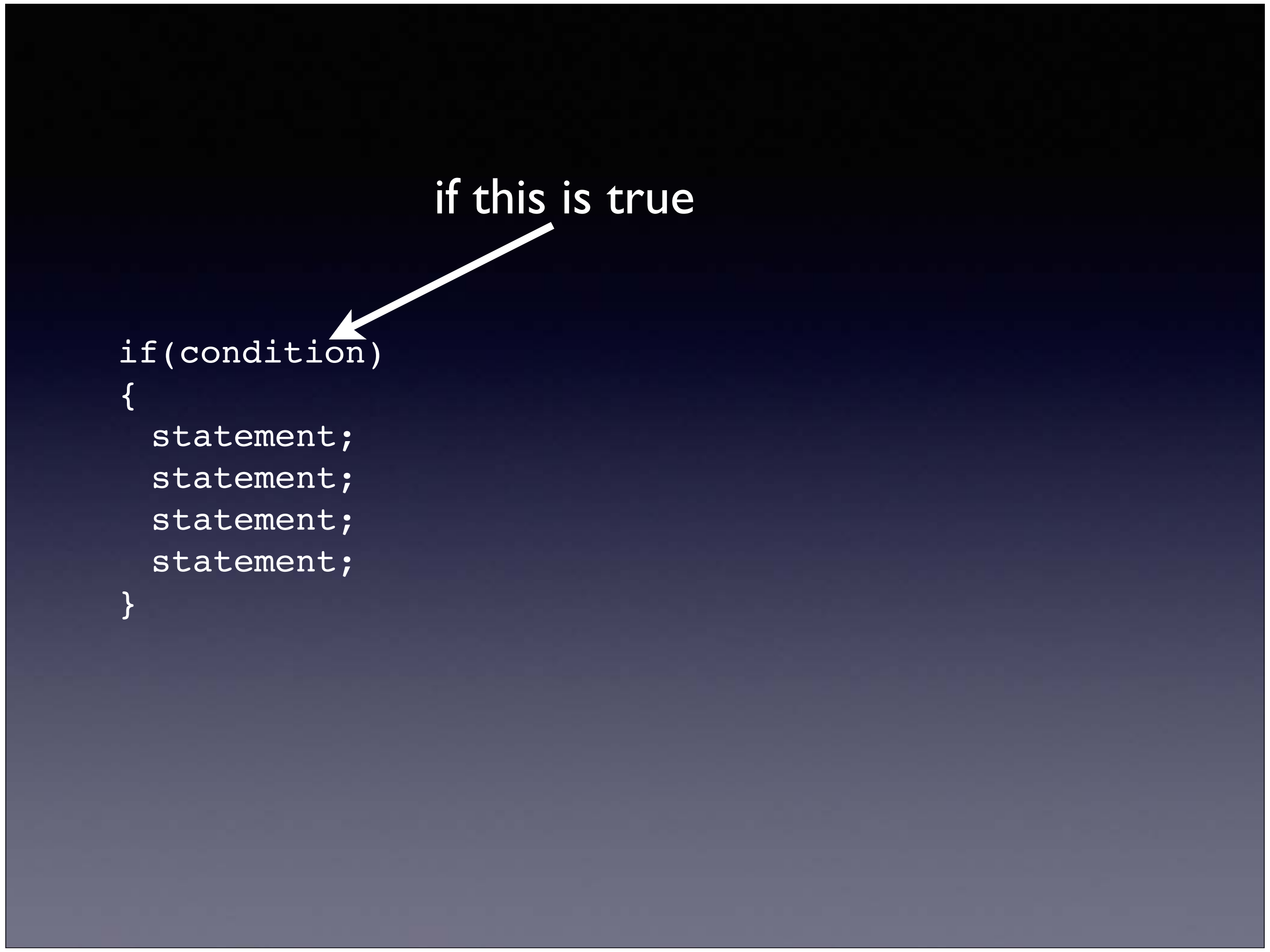
Blocks

- Executing only one statement is limiting
- We can execute a block of statements by putting that block in `{ … }`
- Like we do for functions
- All the statements in the block then executed if the condition is true
- Often sensible to include `{  }` anyway



```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
```

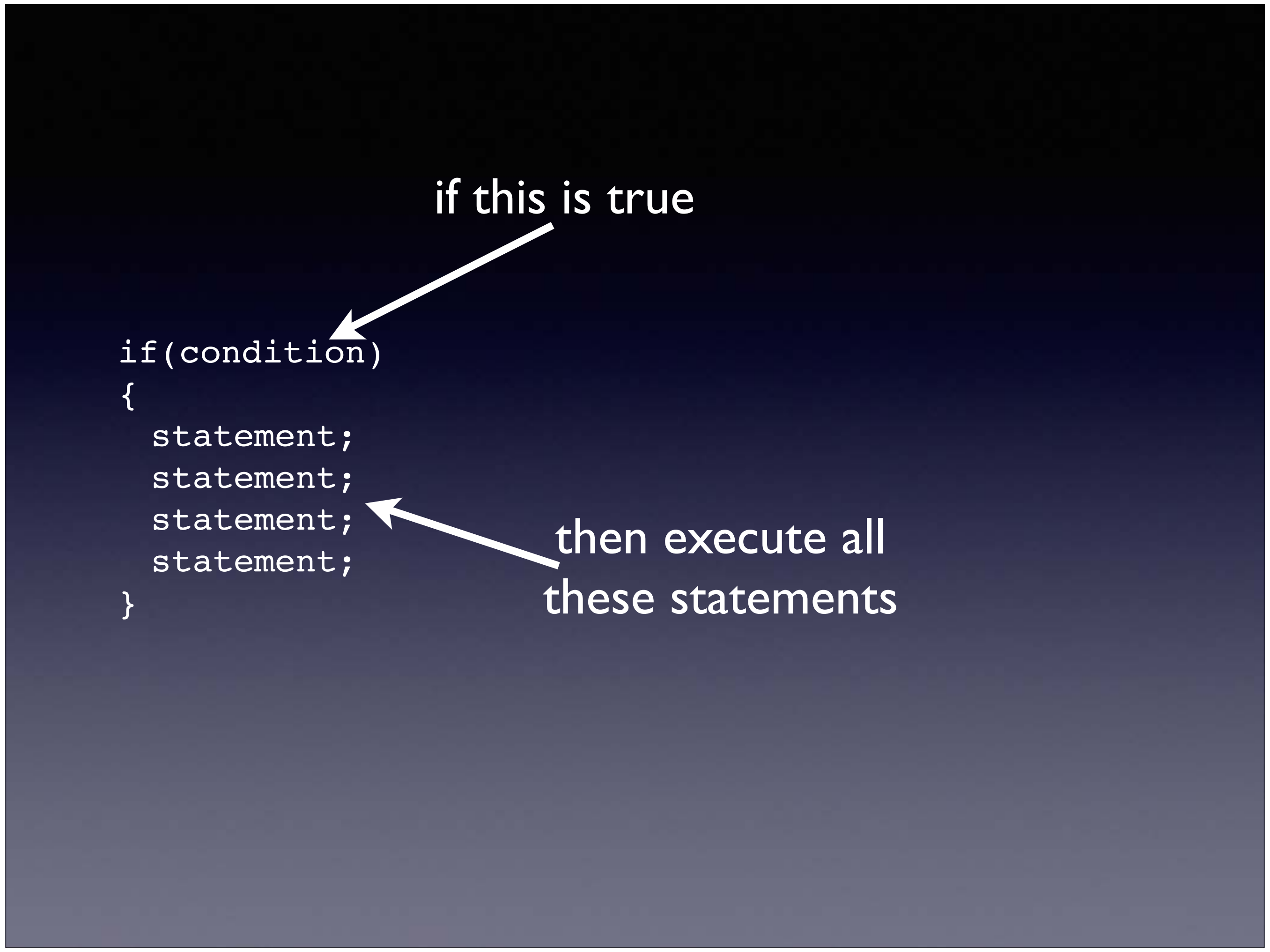
the statements are executed only if condition is true
Modify program to demo



if this is true

```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
```

the statements are executed only if condition is true
Modify program to demo



if this is true

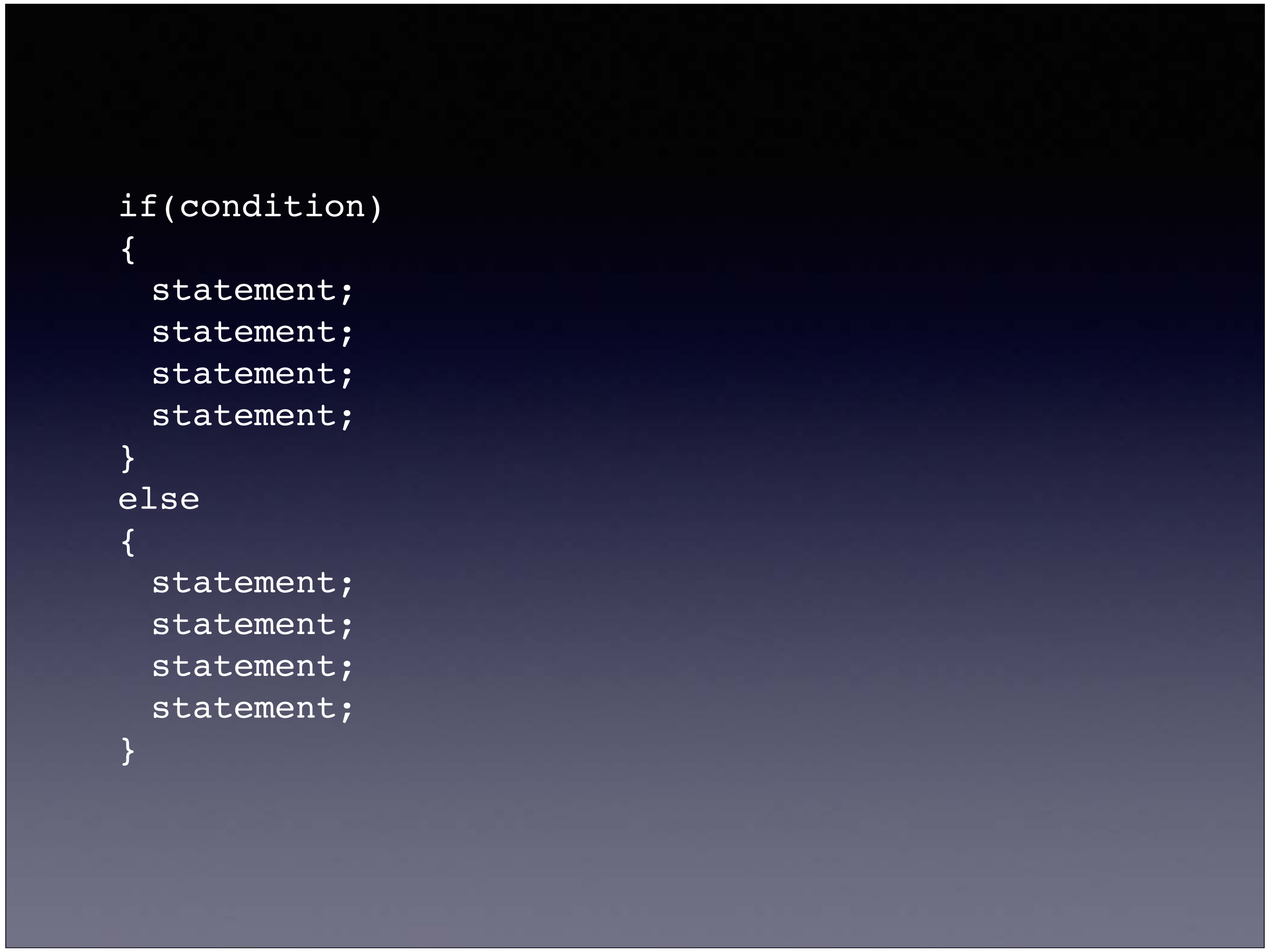
```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
```

then execute all these statements

the statements are executed only if condition is true
Modify program to demo

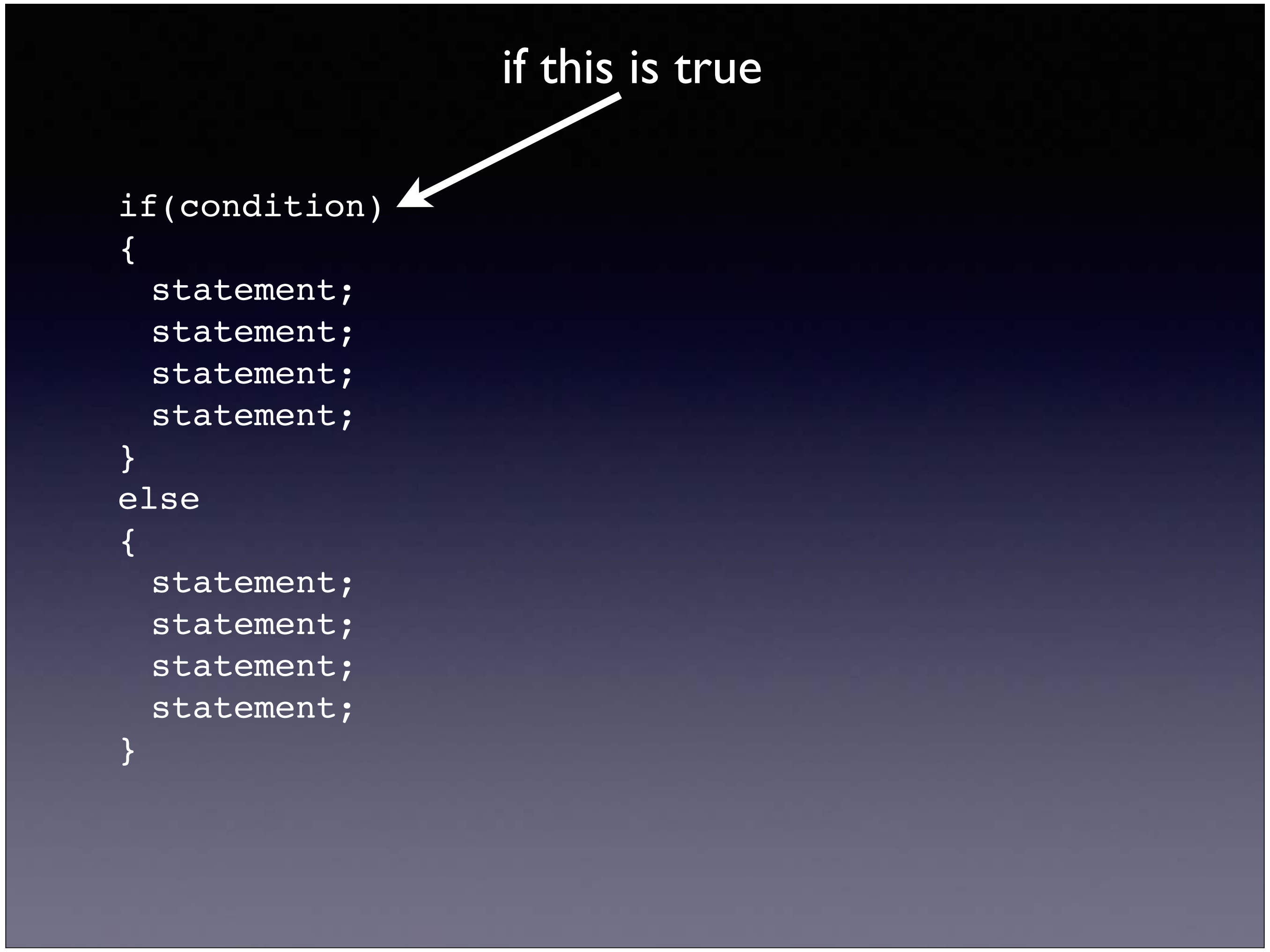
if true do this, else do that

- What happens if we want to do one thing if the condition is true
- But something else if it is false
- Could use a second if, but not always possible
- C provides an `else` clause



```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
else
{
  statement;
  statement;
  statement;
  statement;
}
```

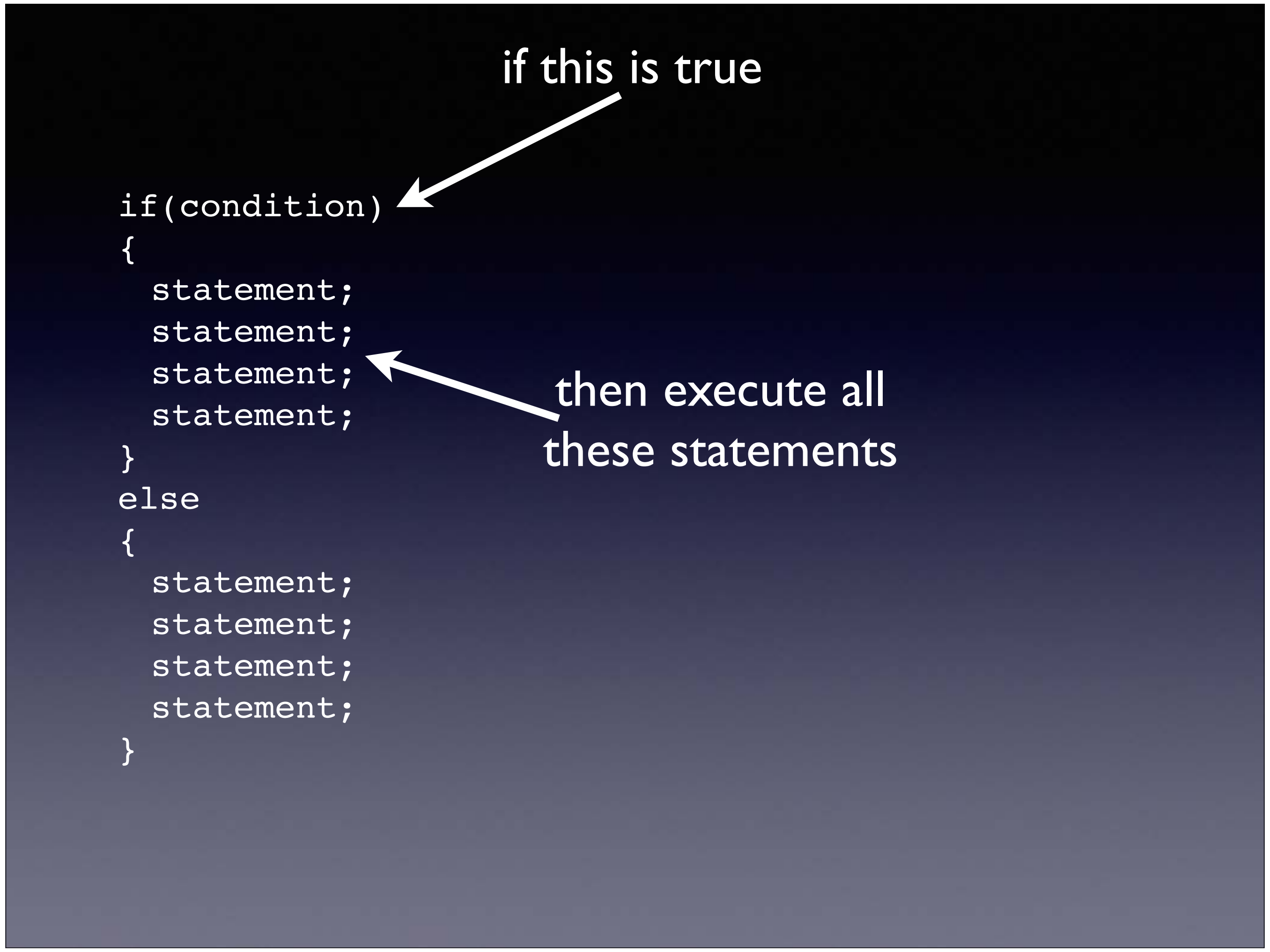
Demo



if this is true

```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
else
{
  statement;
  statement;
  statement;
  statement;
}
```

Demo

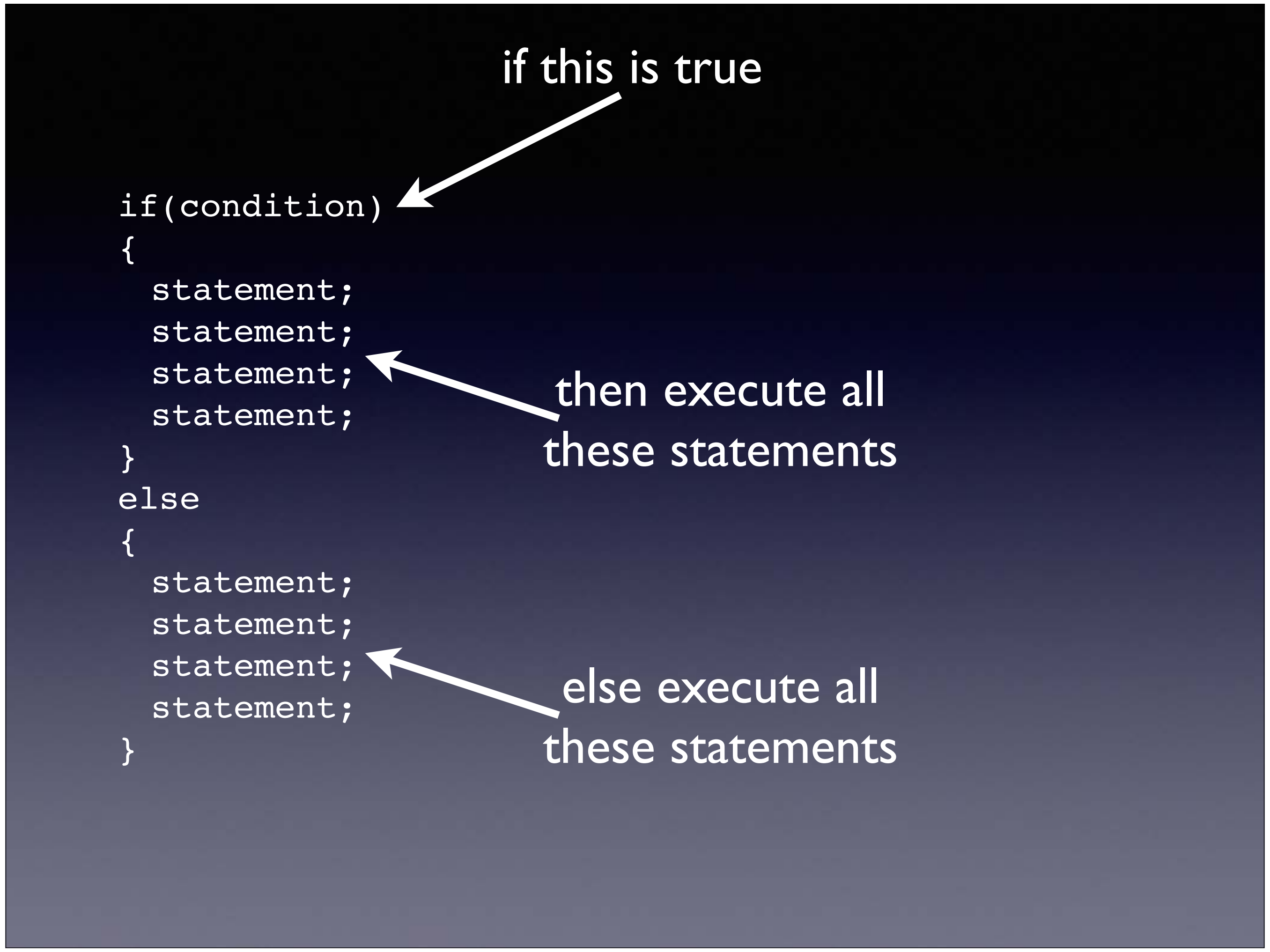


if this is true

```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
else
{
  statement;
  statement;
  statement;
  statement;
}
```

then execute all these statements

Demo



if this is true

```
if(condition)
{
  statement;
  statement;
  statement;
  statement;
}
else
{
  statement;
  statement;
  statement;
  statement;
}
```

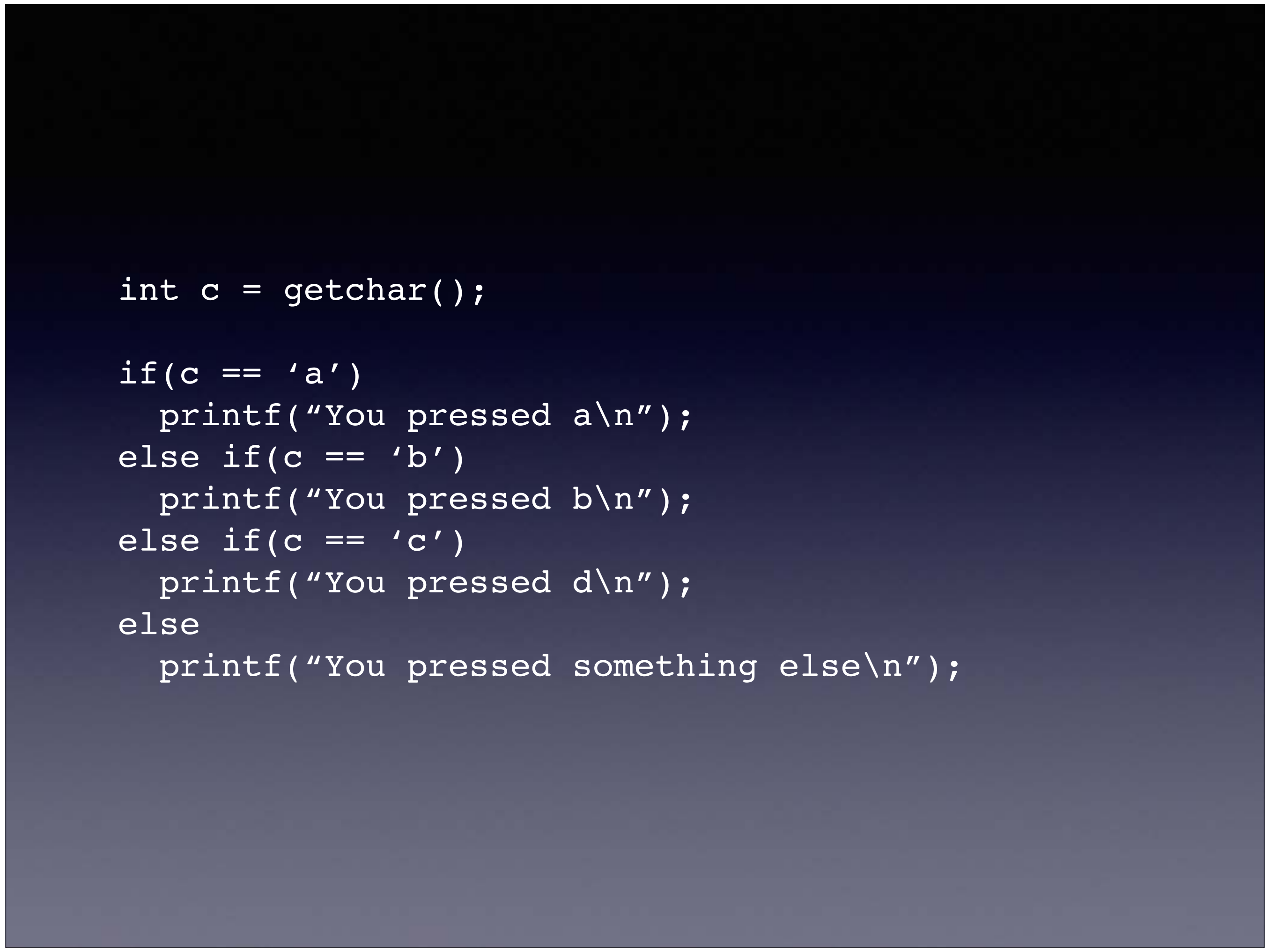
then execute all these statements

else execute all these statements

Demo

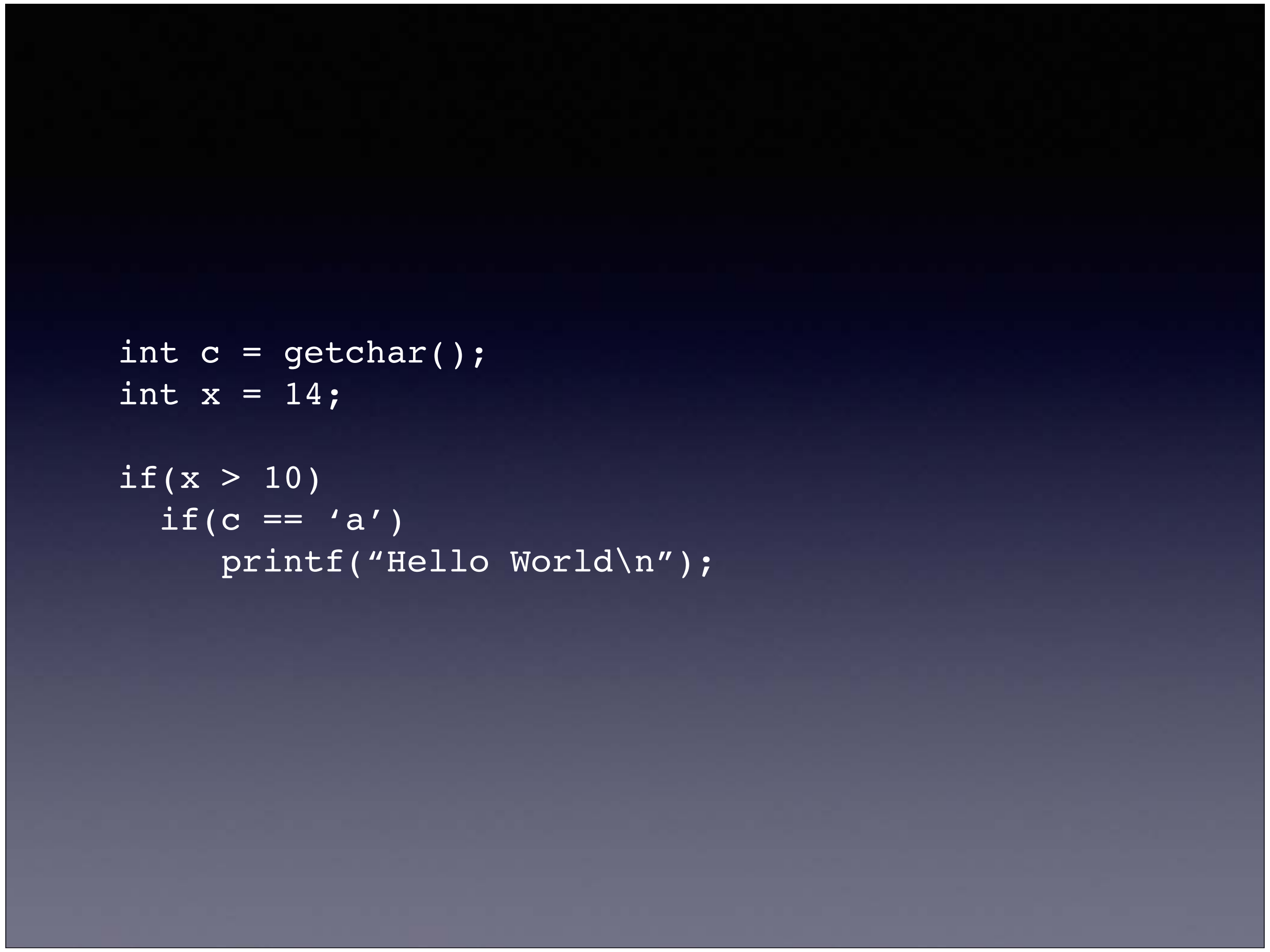
if else if else if

- Can chain `if…else` statements
- If this then, else if something else then, else…
- E.g. for handling menu choices
- Can also combine in the `if` part too
- But be careful…



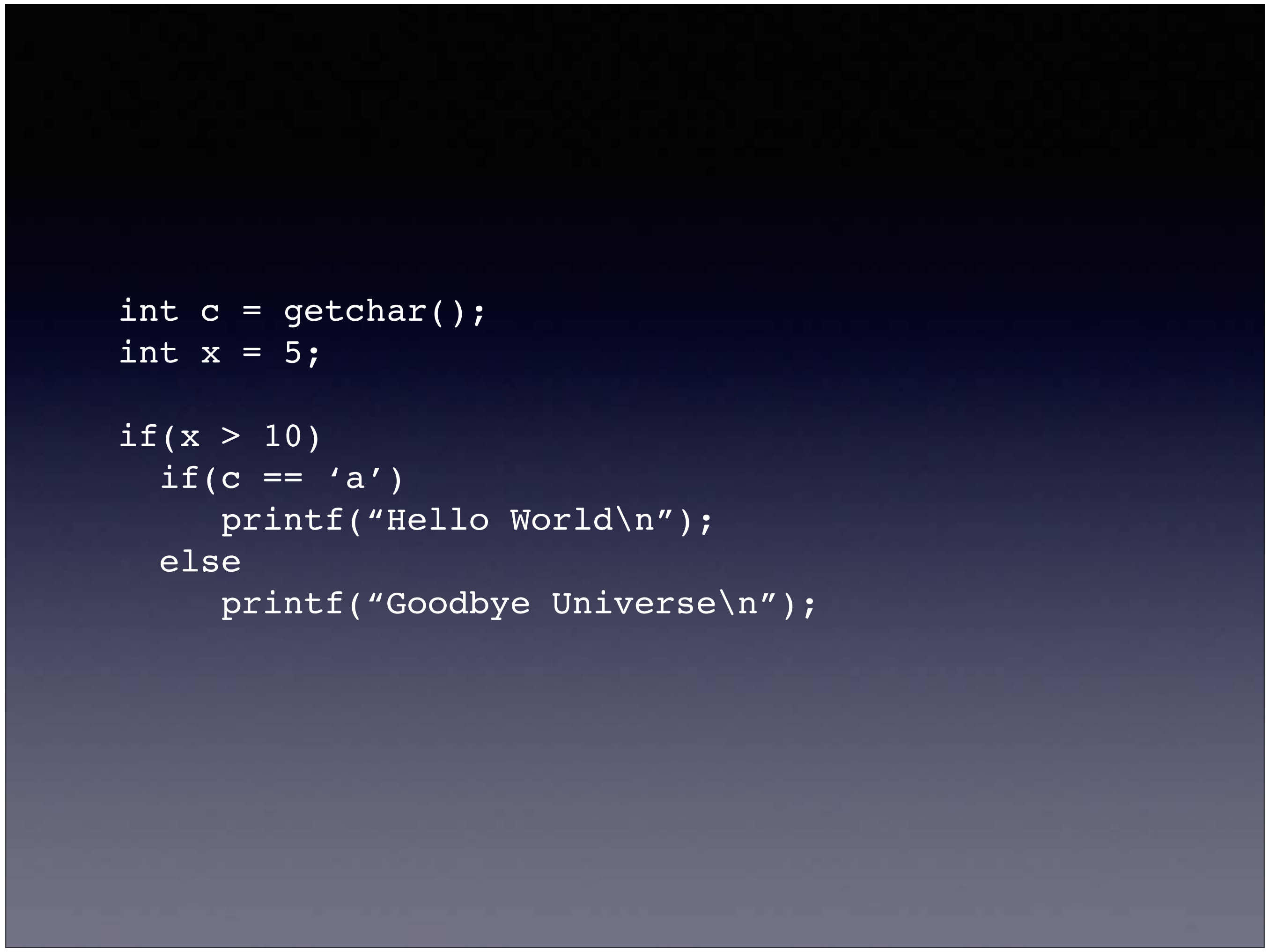
```
int c = getchar();

if(c == ‘a’)
  printf(“You pressed a\n”);
else if(c == ‘b’)
  printf(“You pressed b\n”);
else if(c == ‘c’)
  printf(“You pressed d\n”);
else
  printf(“You pressed something else\n”);
```

Demo
Have to store the value from getchar() or it would mean something else…



```
int c = getchar();
int x = 14;

if(x > 10)
  if(c == ‘a’)
     printf(“Hello World\n”);
```

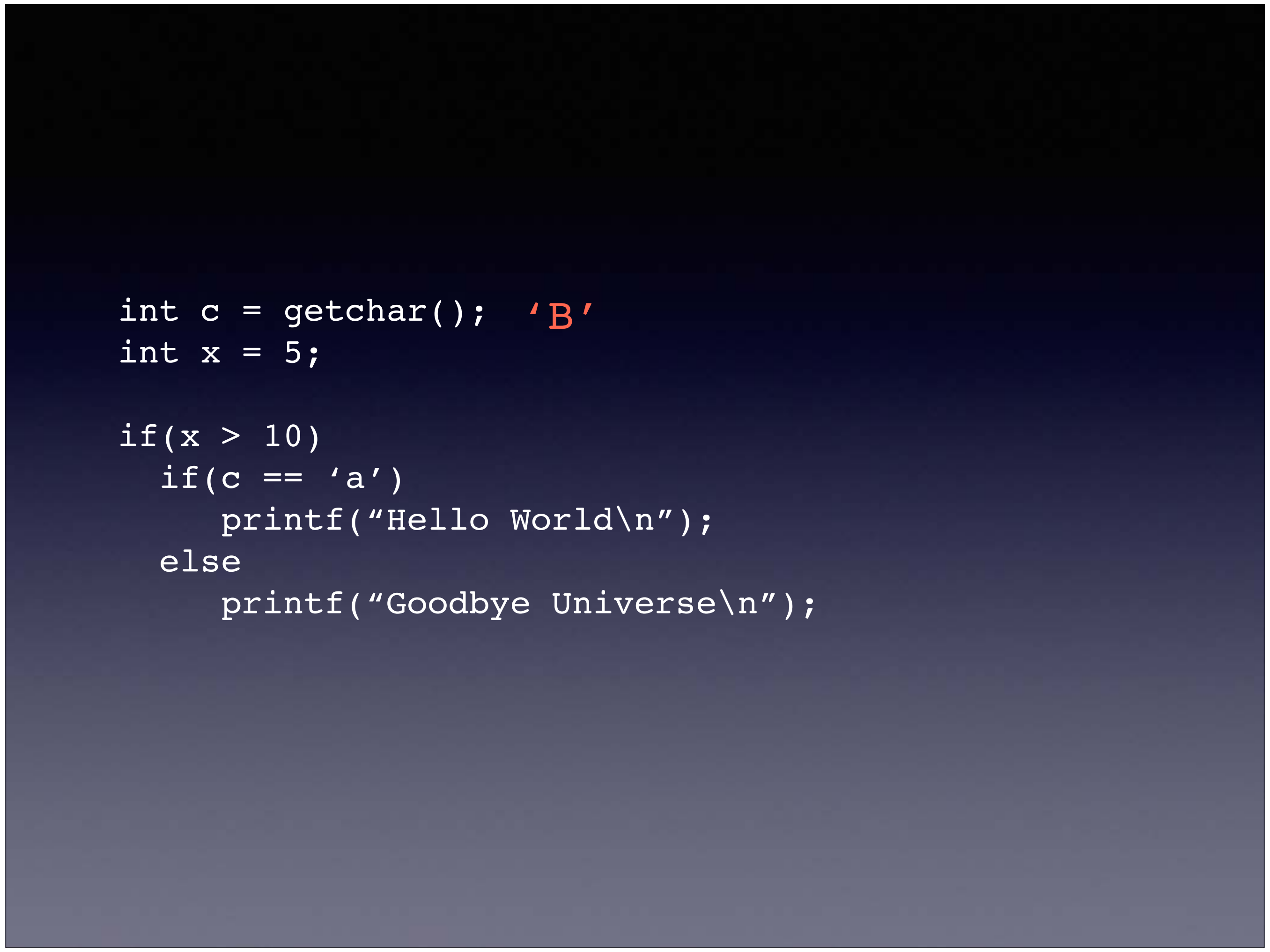
Makes sense if x is greater than 10 and c is a then print Hello world



```
int c = getchar();
int x = 5;

if(x > 10)
  if(c == ‘a’)
     printf(“Hello World\n”);
  else
     printf(“Goodbye Universe\n”);
```

What happens if I press b?

Does the else associate with if(x>10) or if(c == ‘a’)???



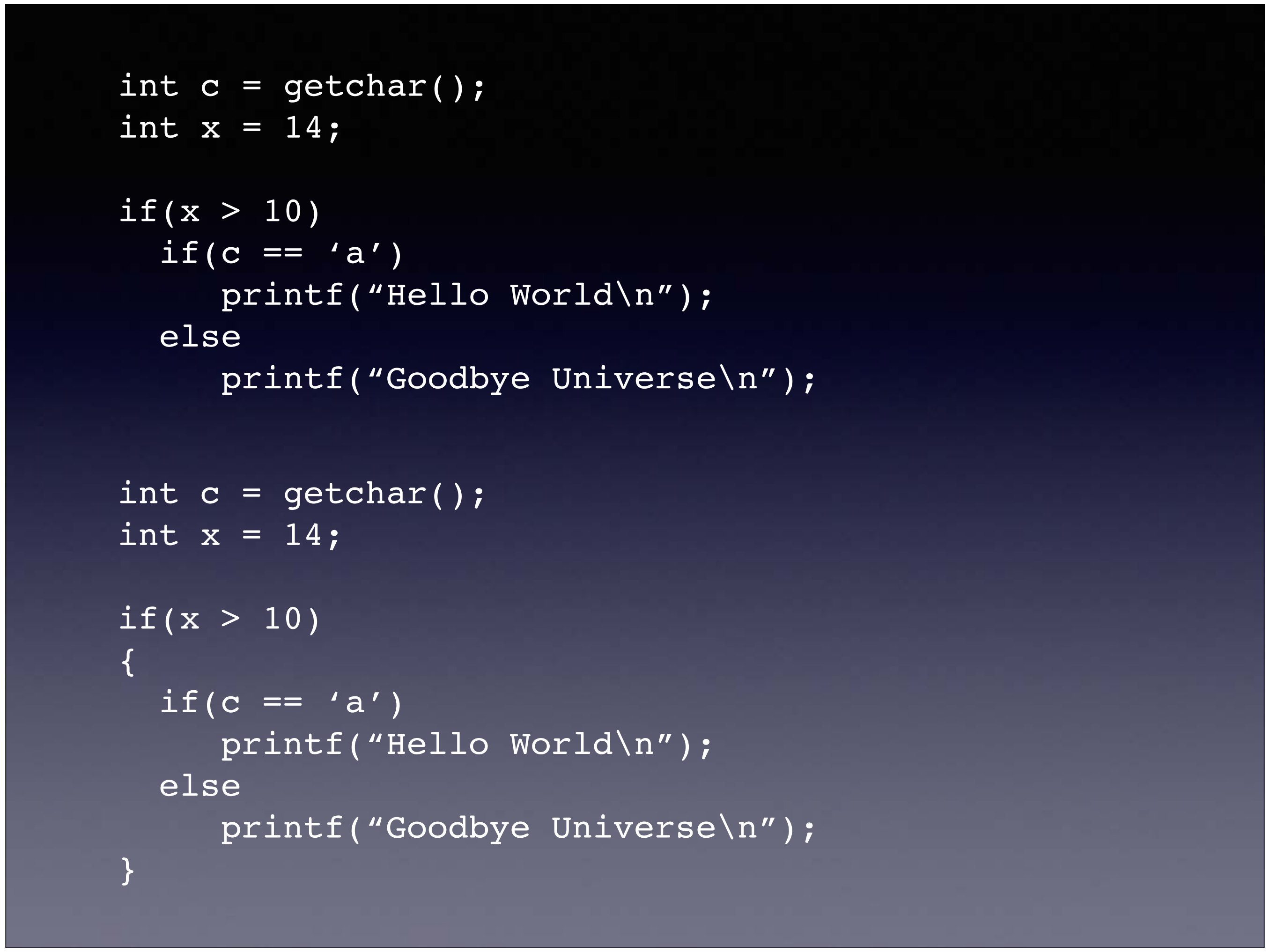
```
int c = getchar();  ‘B’
int x = 5;

if(x > 10)
  if(c == ‘a’)
     printf(“Hello World\n”);
  else
     printf(“Goodbye Universe\n”);
```

What happens if I press b?

Does the else associate with if(x>10) or if(c == ‘a’)???

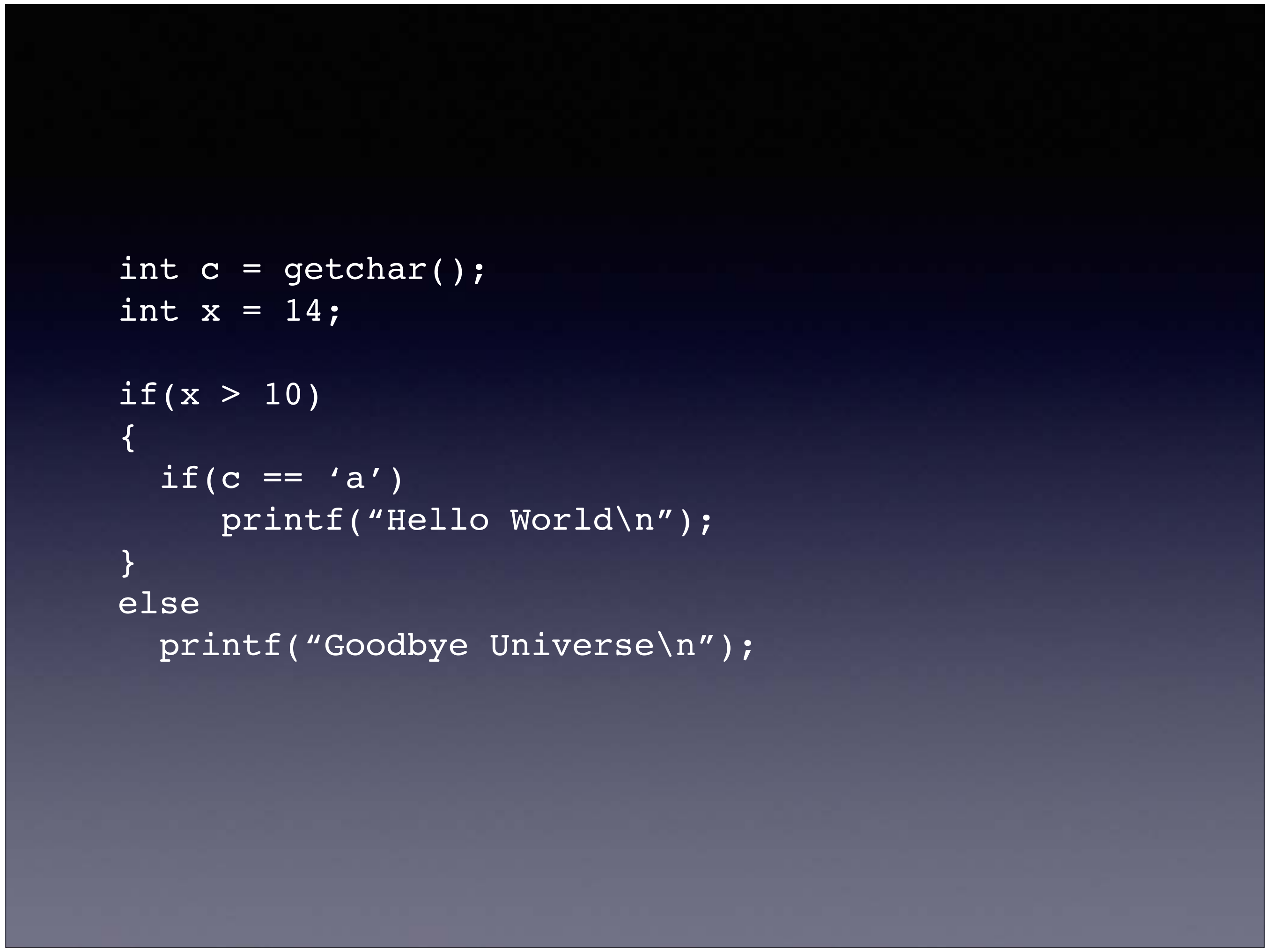
Association

- Previous code is ambiguous since `else` is optional
- C removes the ambiguity by saying `else` associates with the closest previous `else`-less `if`
- Use braces to form a block if you want the opposite



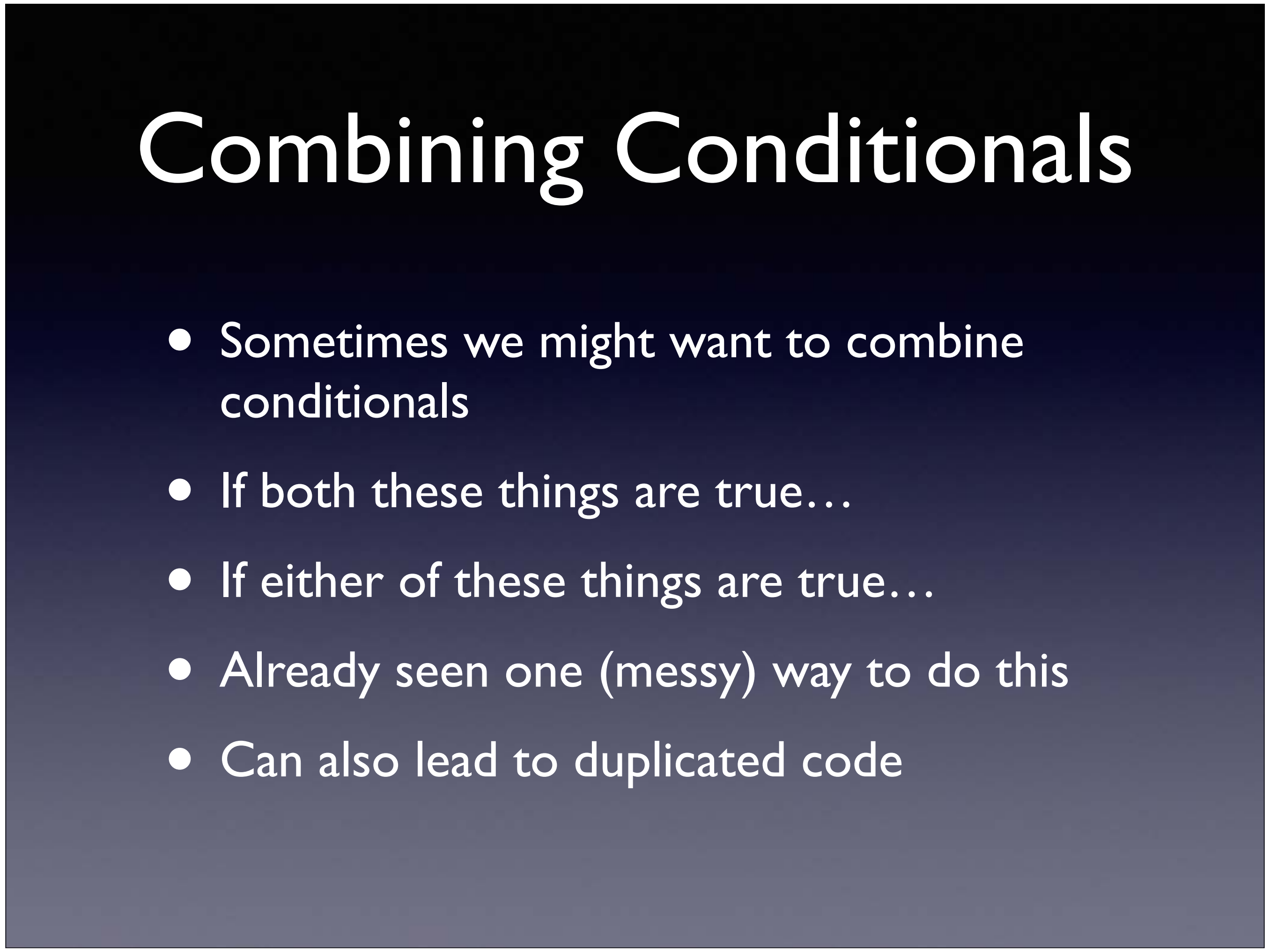
```
int c = getchar();
int x = 14;

if(x > 10)
  if(c == ‘a’)
     printf(“Hello World\n”);
  else
     printf(“Goodbye Universe\n”);


int c = getchar();
int x = 14;

if(x > 10)
{
  if(c == ‘a’)
     printf(“Hello World\n”);
  else
     printf(“Goodbye Universe\n”);
}
```

These two are equivalent...



```
int c = getchar();
int x = 14;

if(x > 10)
{
  if(c == ‘a’)
     printf(“Hello World\n”);
}
else
  printf(“Goodbye Universe\n”);
```

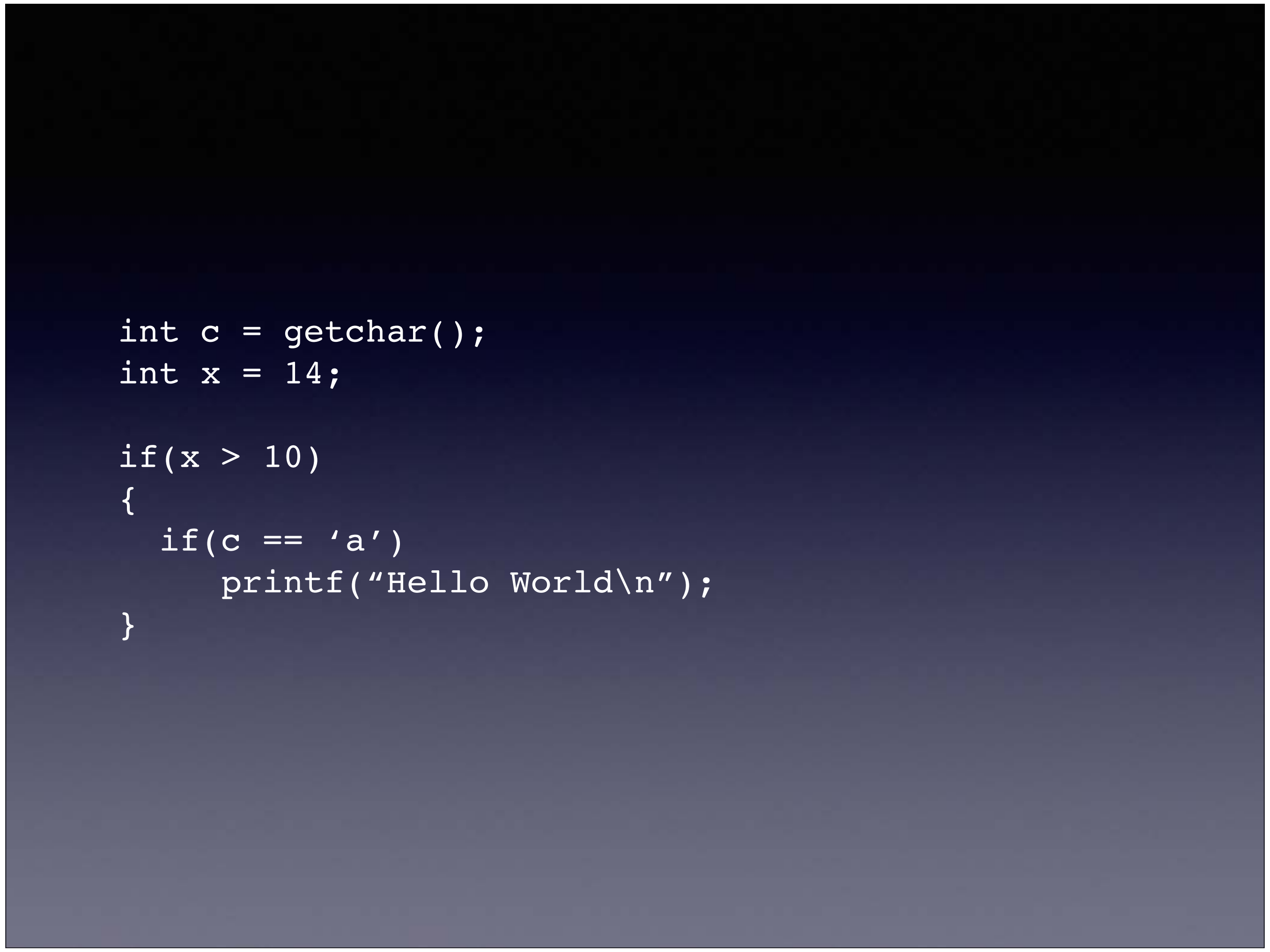
This is the opposite form...



Combining Conditionals

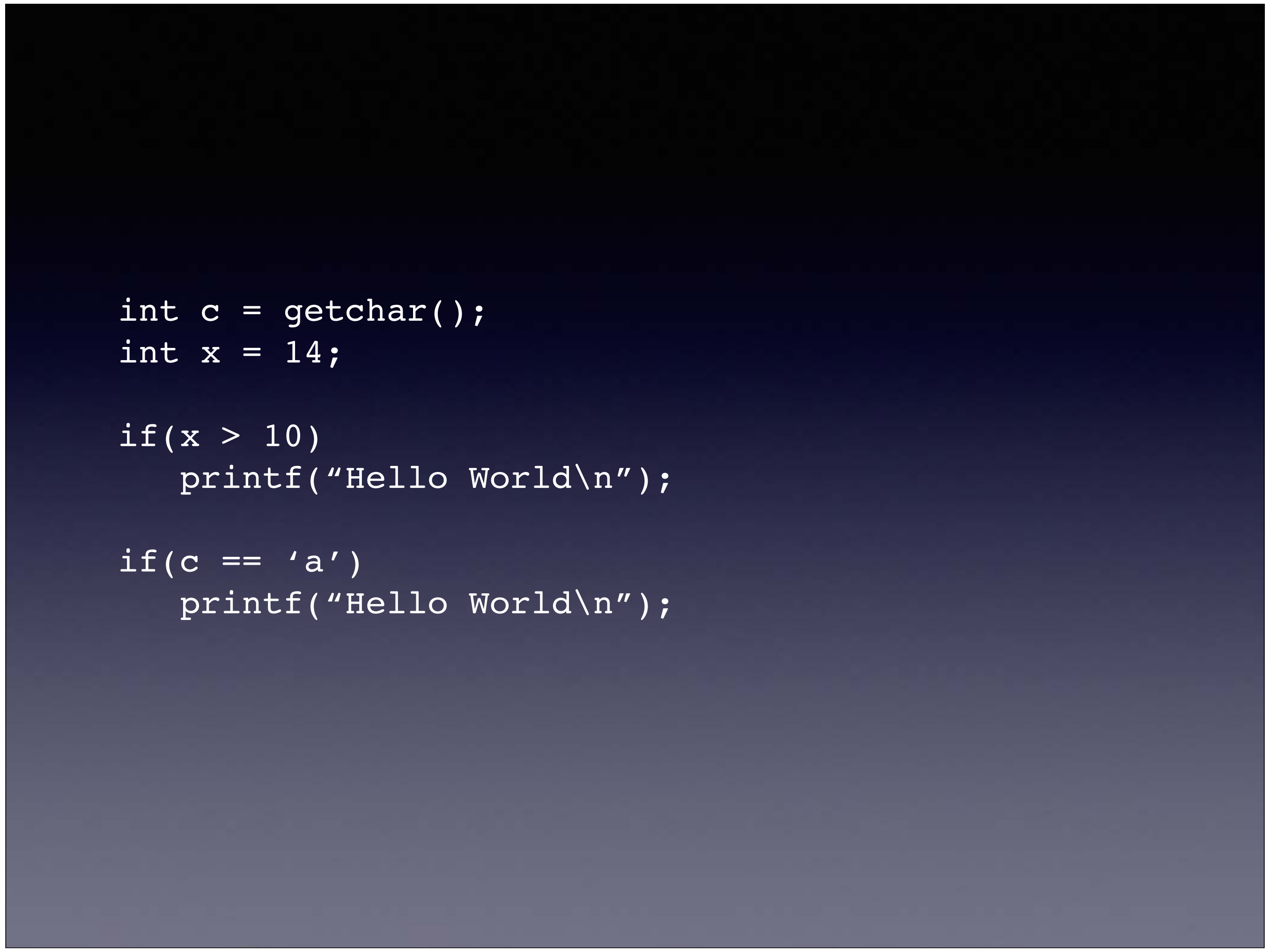
- Sometimes we might want to combine conditionals
- If both these things are true…
- If either of these things are true…
- Already seen one (messy) way to do this
- Can also lead to duplicated code

duplicated code is bad



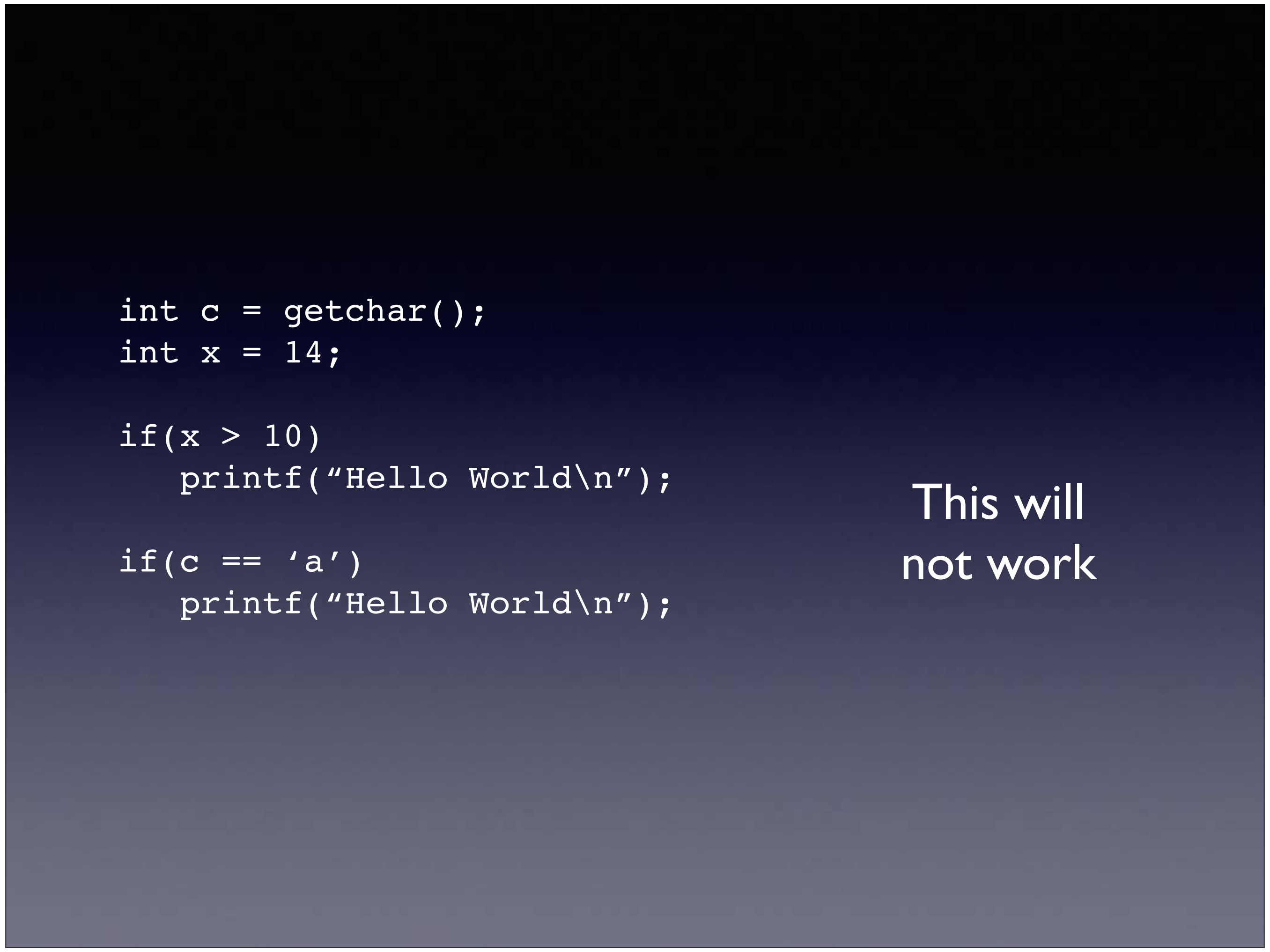
```
int c = getchar();
int x = 14;

if(x > 10)
{
  if(c == ‘a’)
     printf(“Hello World\n”);
}
```

printf Hello World only printed if both x >10 and c == ‘a’



```
int c = getchar();
int x = 14;

if(x > 10)
   printf(“Hello World\n”);

if(c == ‘a’)
   printf(“Hello World\n”);
```

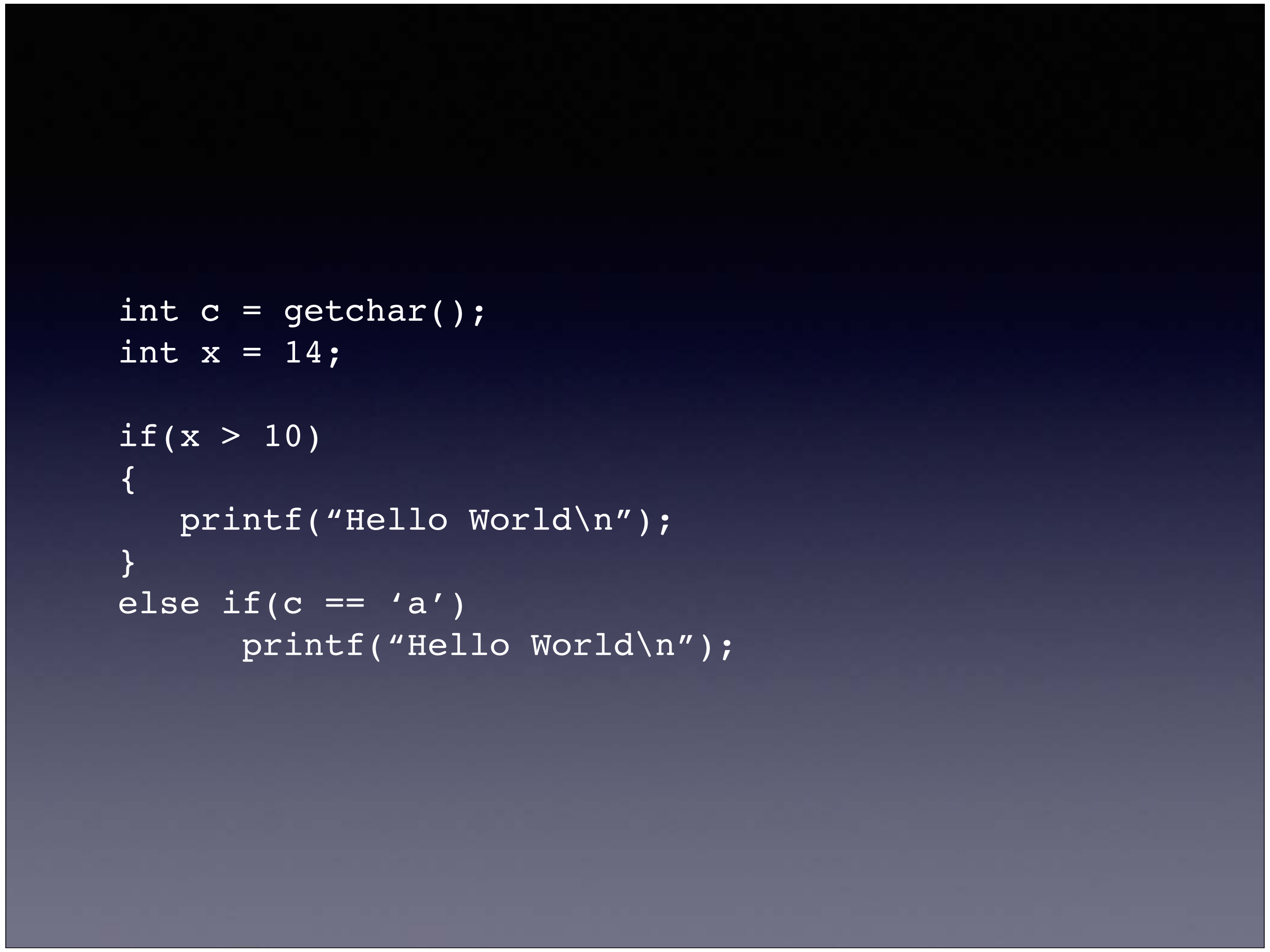
Or is harder -- if both true, then HEllo World would be printed twice...
Need to make a note if we execute this



```
int c = getchar();
int x = 14;

if(x > 10)
   printf(“Hello World\n”);

if(c == ‘a’)
   printf(“Hello World\n”);
```

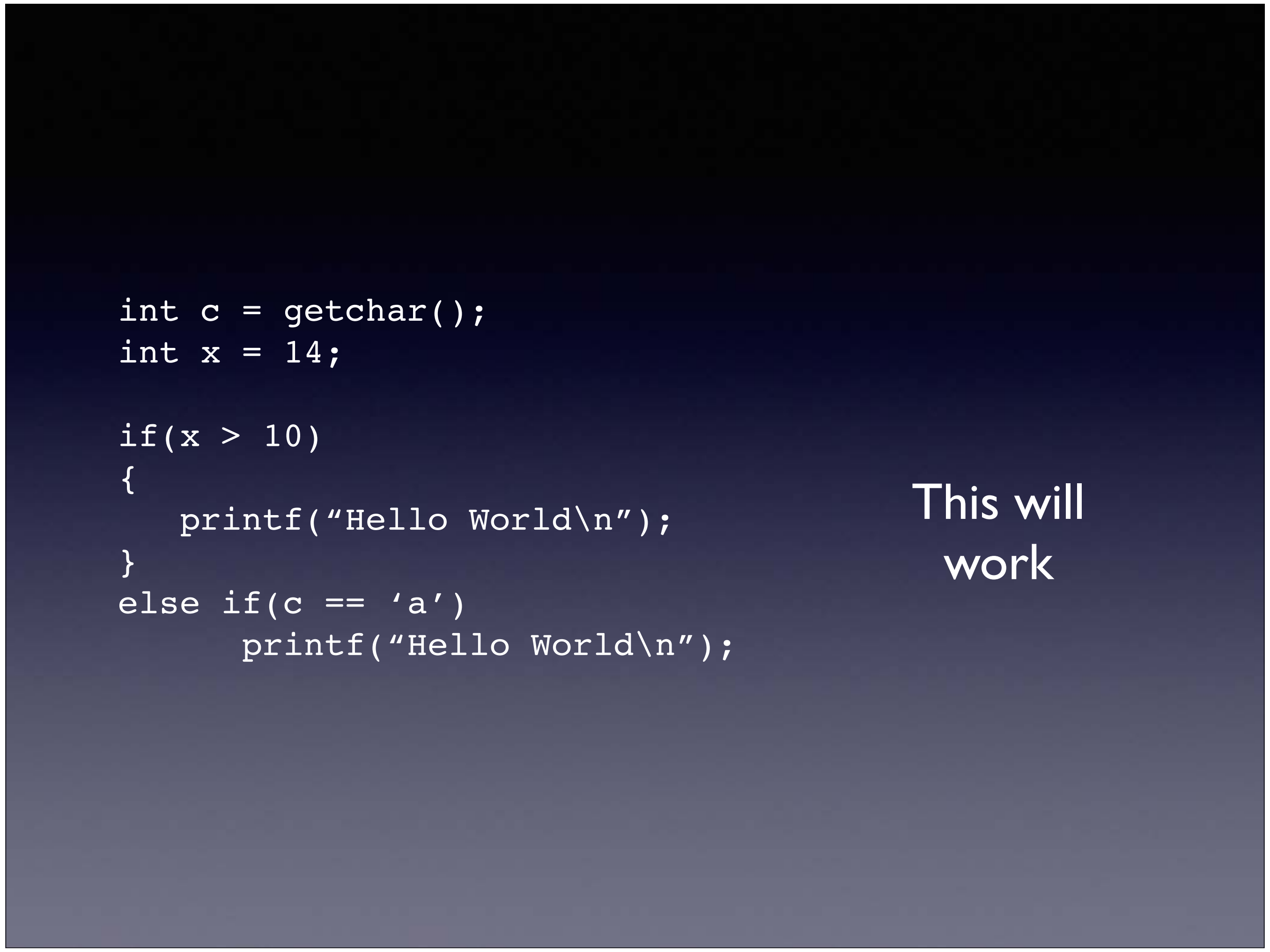
This will
not work

Or is harder -- if both true, then HEllo World would be printed twice...
Need to make a note if we execute this



```
int c = getchar();
int x = 14;

if(x > 10)
{
   printf(“Hello World\n”);
}
else if(c == ‘a’)
      printf(“Hello World\n”);
```

For ‘Or’, we can use an else...



```
int c = getchar();
int x = 14;

if(x > 10)
{
   printf(“Hello World\n”);
}
else if(c == ‘a’)
      printf(“Hello World\n”);
```

This will work

For ‘Or’, we can use an else...