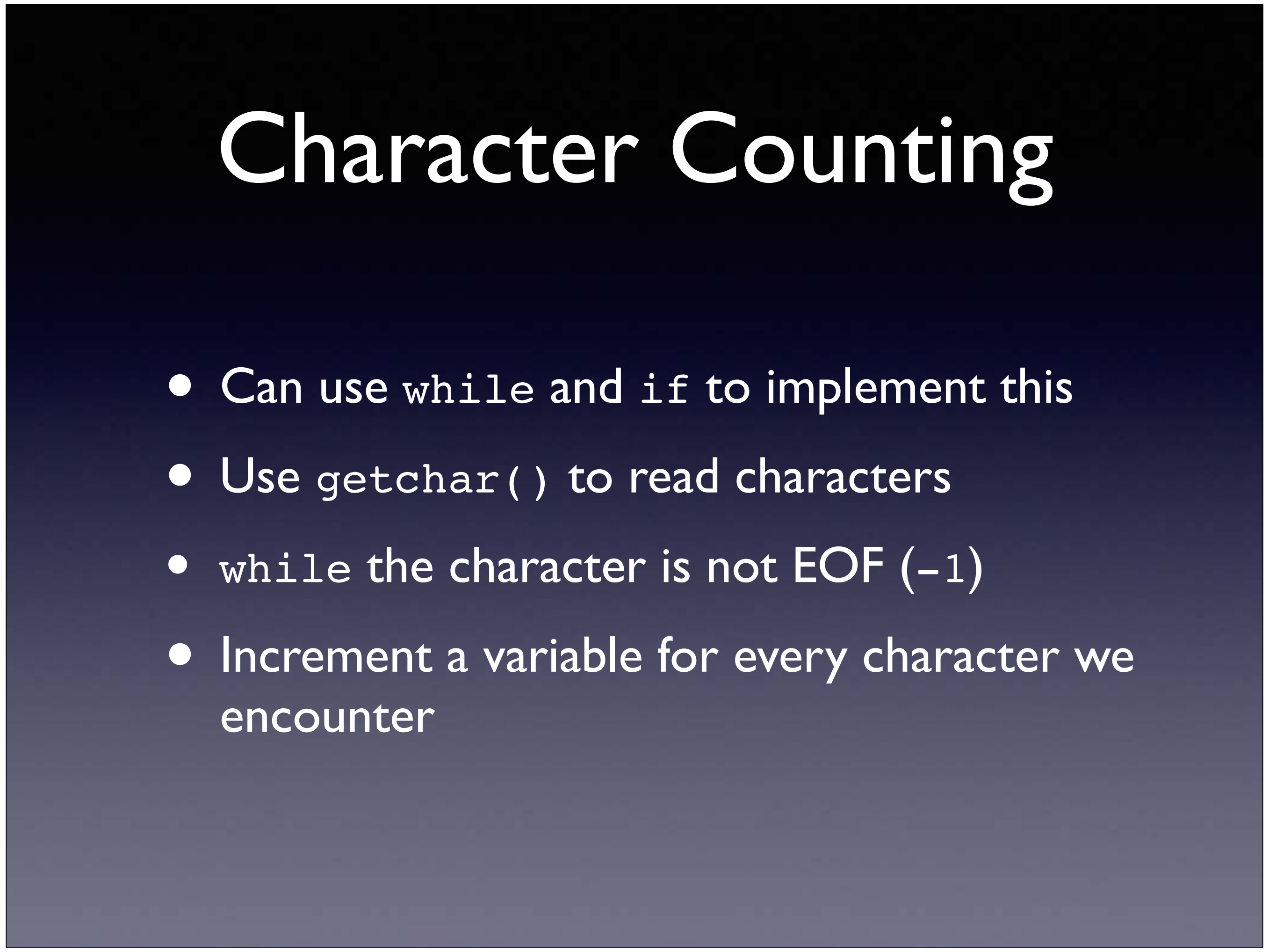
Interlude
Counting Words

Steven R. Bagley

Counting Characters

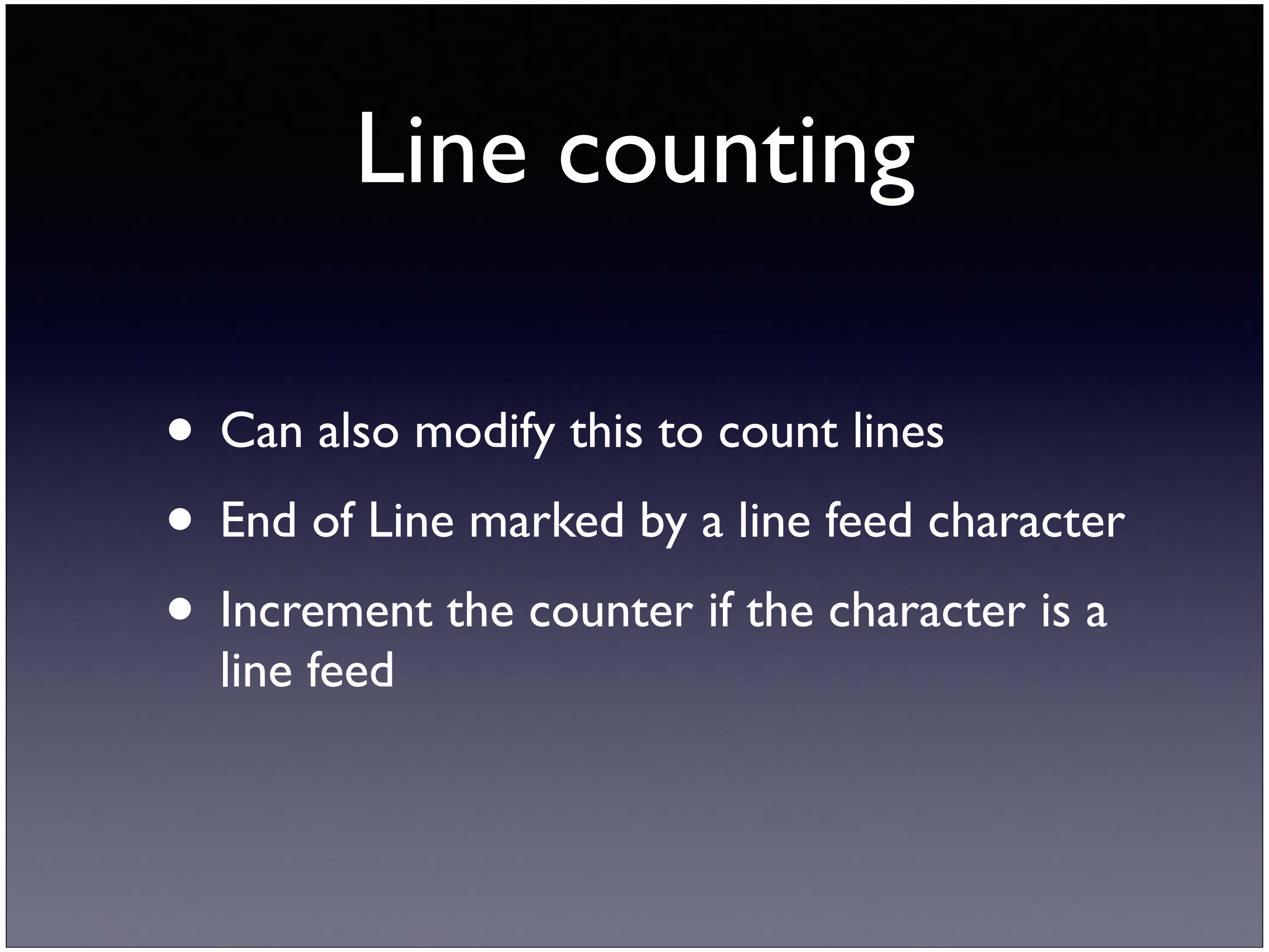
```
Alice was beginning to get very
tired of sitting by her sister
on the bank, and of having
nothing to do: once or twice she
had peeped into the book her
sister was reading, but it had
no pictures or conversations in
it, “and what is the use of a
book,” thought Alice “without
pictures or conversation?”
```



Character Counting

- Can use `while` and `if` to implement this
- Use `getchar()` to read characters
- `while` the character is not EOF (`-1`)
- Increment a variable for every character we encounter

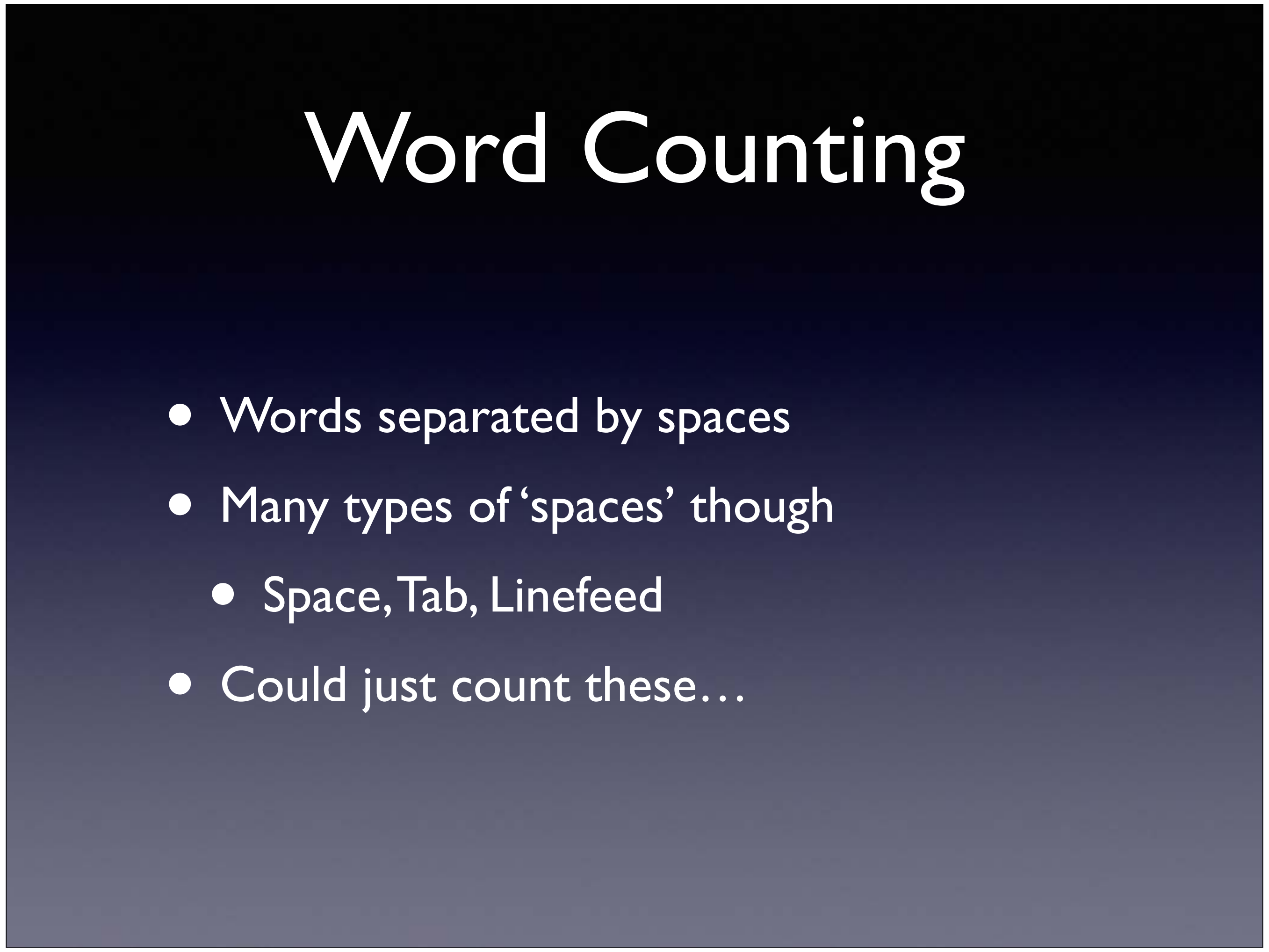
EOF == end of file
Implement



Line counting

- Can also modify this to count lines
- End of Line marked by a line feed character
- Increment the counter if the character is a line feed

Go add this to our program



Word Counting

- Words separated by spaces
- Many types of ‘spaces’ though
  - Space, Tab, Linefeed
- Could just count these…

Go Count...

Word Counting

- Problem…
- Words separated by space, tab or linefeed
- But there may be more than one space
- Therefore, can’t just increment counter when we find one
- That would count spaces, not words…

Word Counting

- Need to think about this logically
- What is a word?
- Word is a sequence of letters ended by a space
- Define a letter as a character that is not a space, tab or newline

Word Count

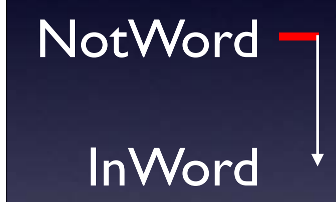
- We need to keep *state*
- Start off in *not in a word* state
- When we detect a ‘letter’, we move to an *in word* state
- Stay in this state while we see more letters
- Then when we detect a space, we move into the *not in a word* state

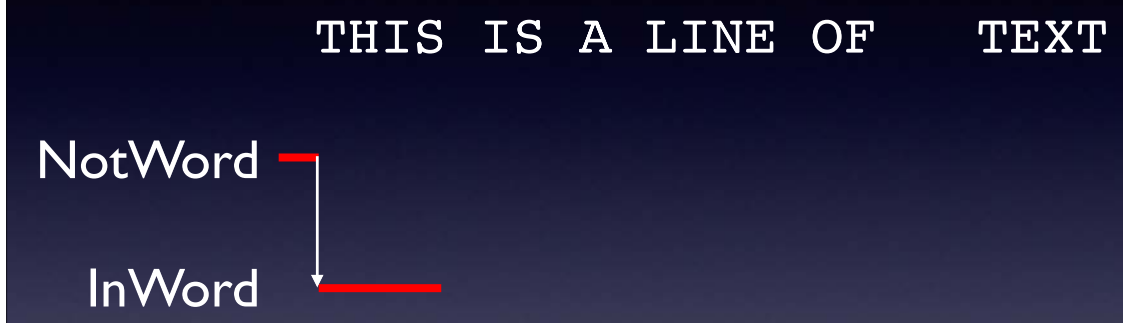
```
THIS IS A LINE OF    TEXT
```

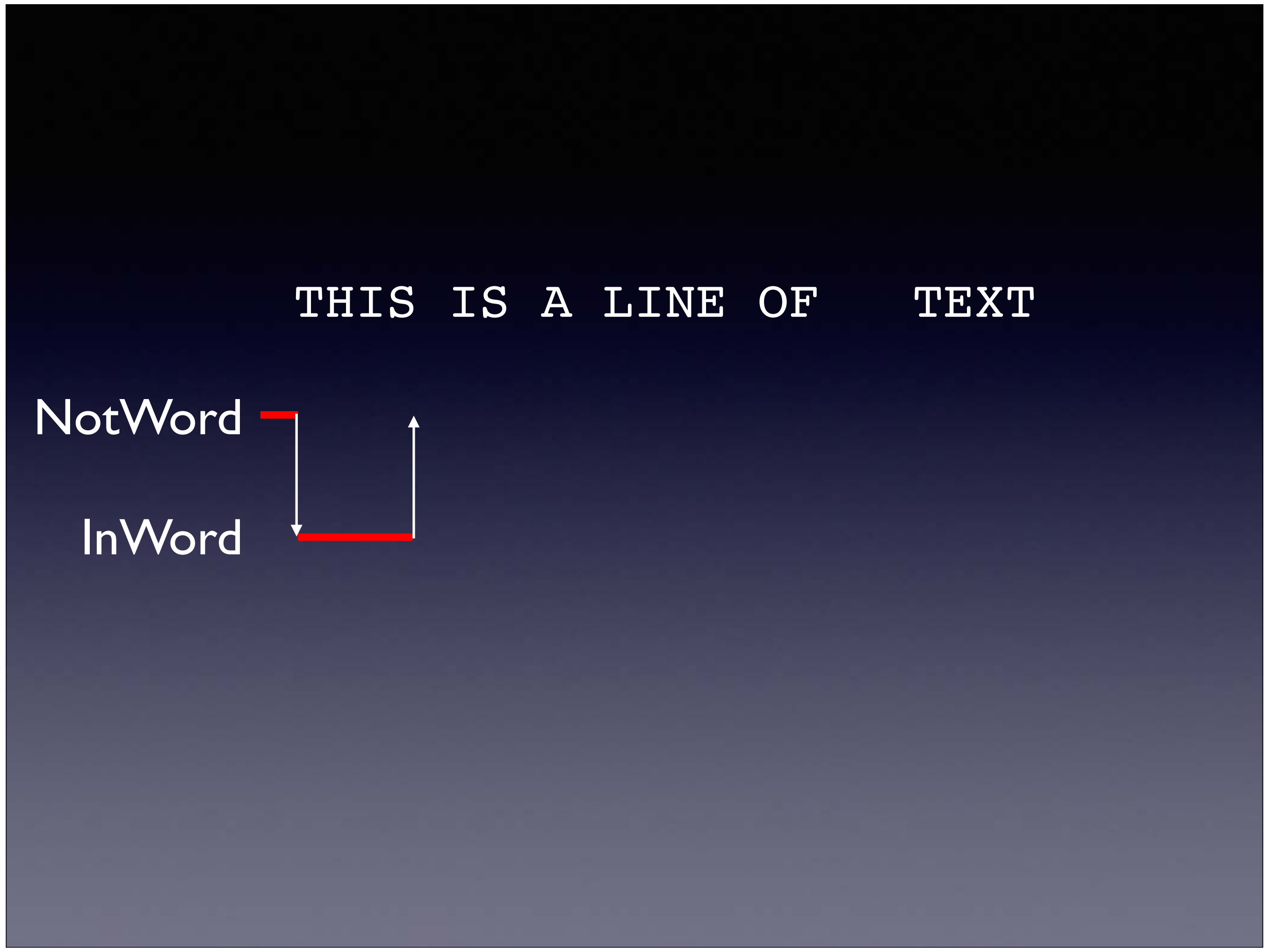
NotWord —

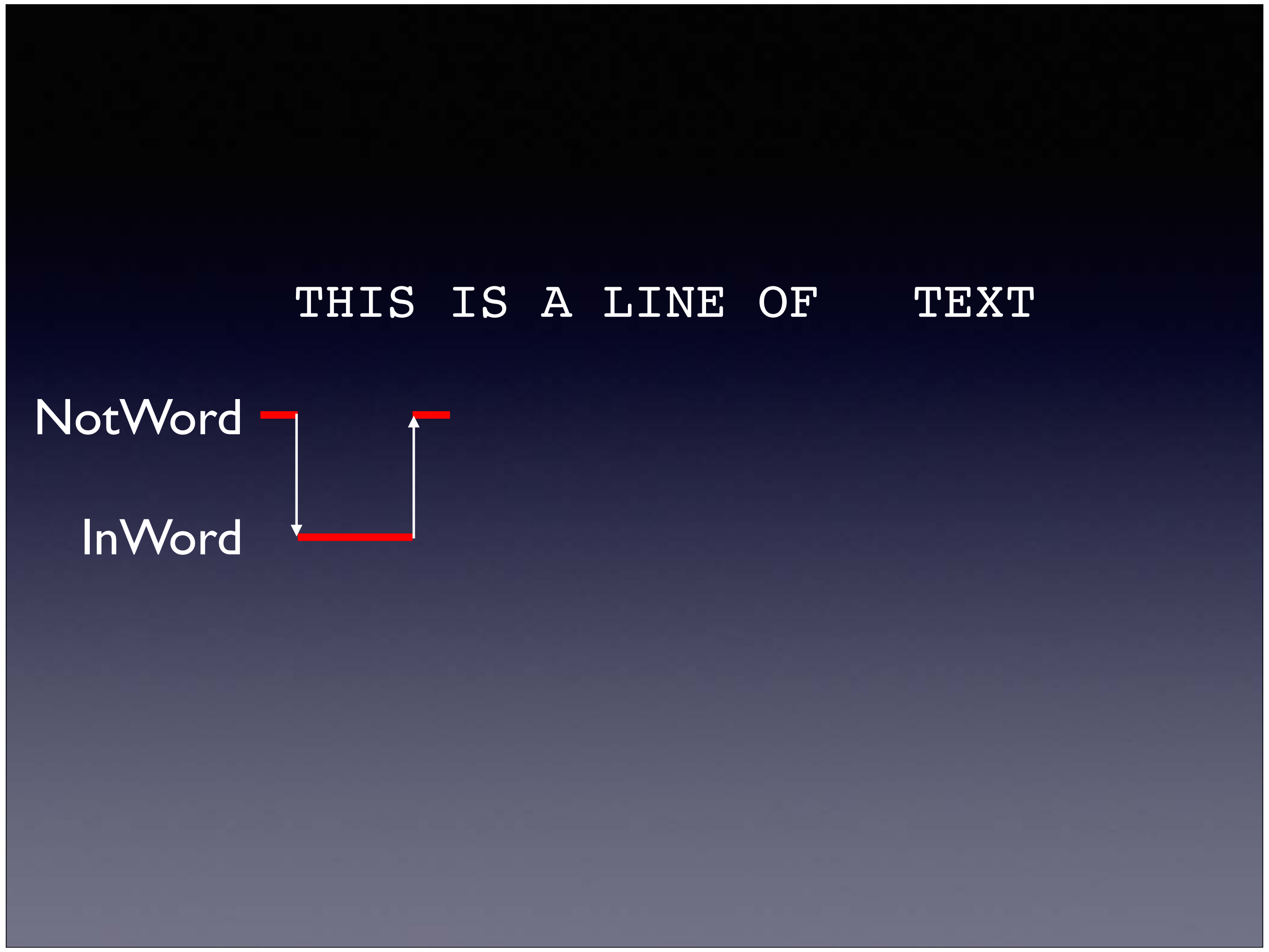
InWord

```
THIS IS A LINE OF    TEXT
```

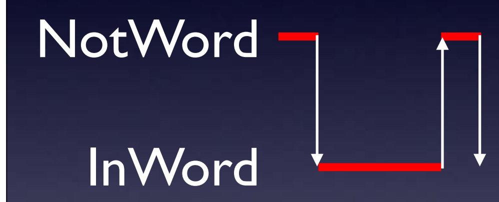



THIS IS A LINE OF   TEXT
NotWord
InWord

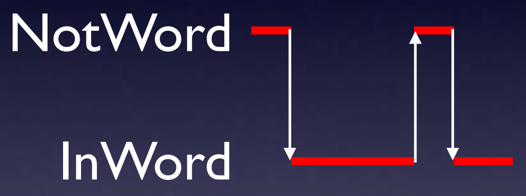

THIS IS A LINE OF   TEXT
NotWord
InWord

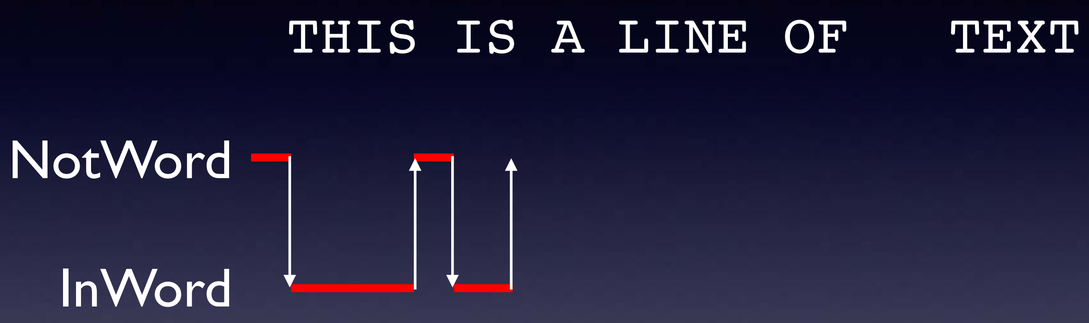


```
THIS IS A LINE OF    TEXT
THIS IS A LINE OF    TEXT
```

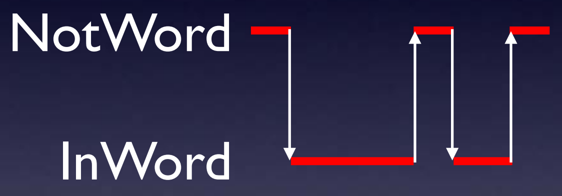


```
THIS IS A LINE OF    TEXT
```

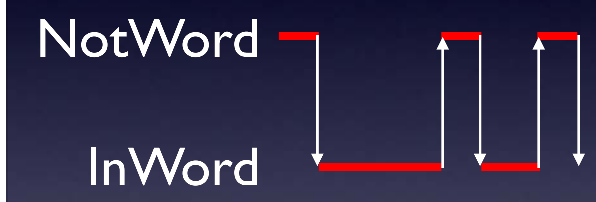



THIS IS A LINE OF   TEXT
NotWord
InWord

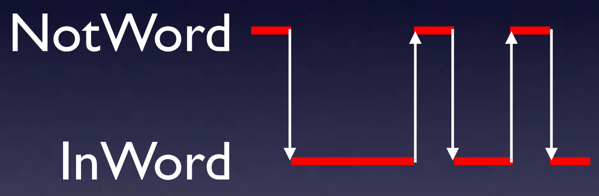
```
THIS IS A LINE OF    TEXT
```



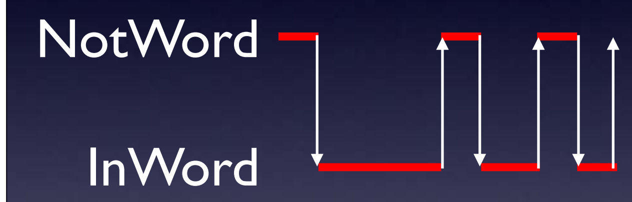
```
THIS IS A LINE OF    TEXT
```



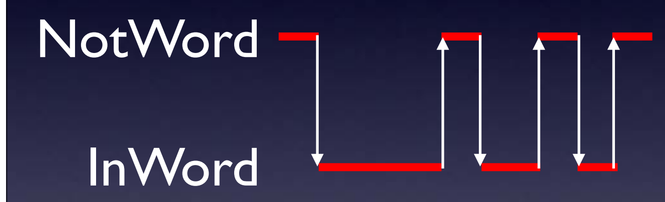
```
THIS IS A LINE OF    TEXT
```



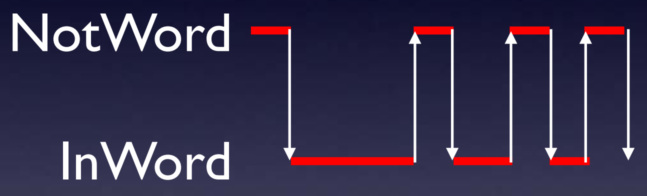
```
THIS IS A LINE OF    TEXT
```



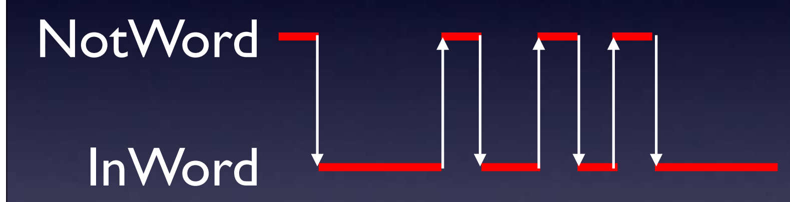
```
THIS IS A LINE OF    TEXT
```

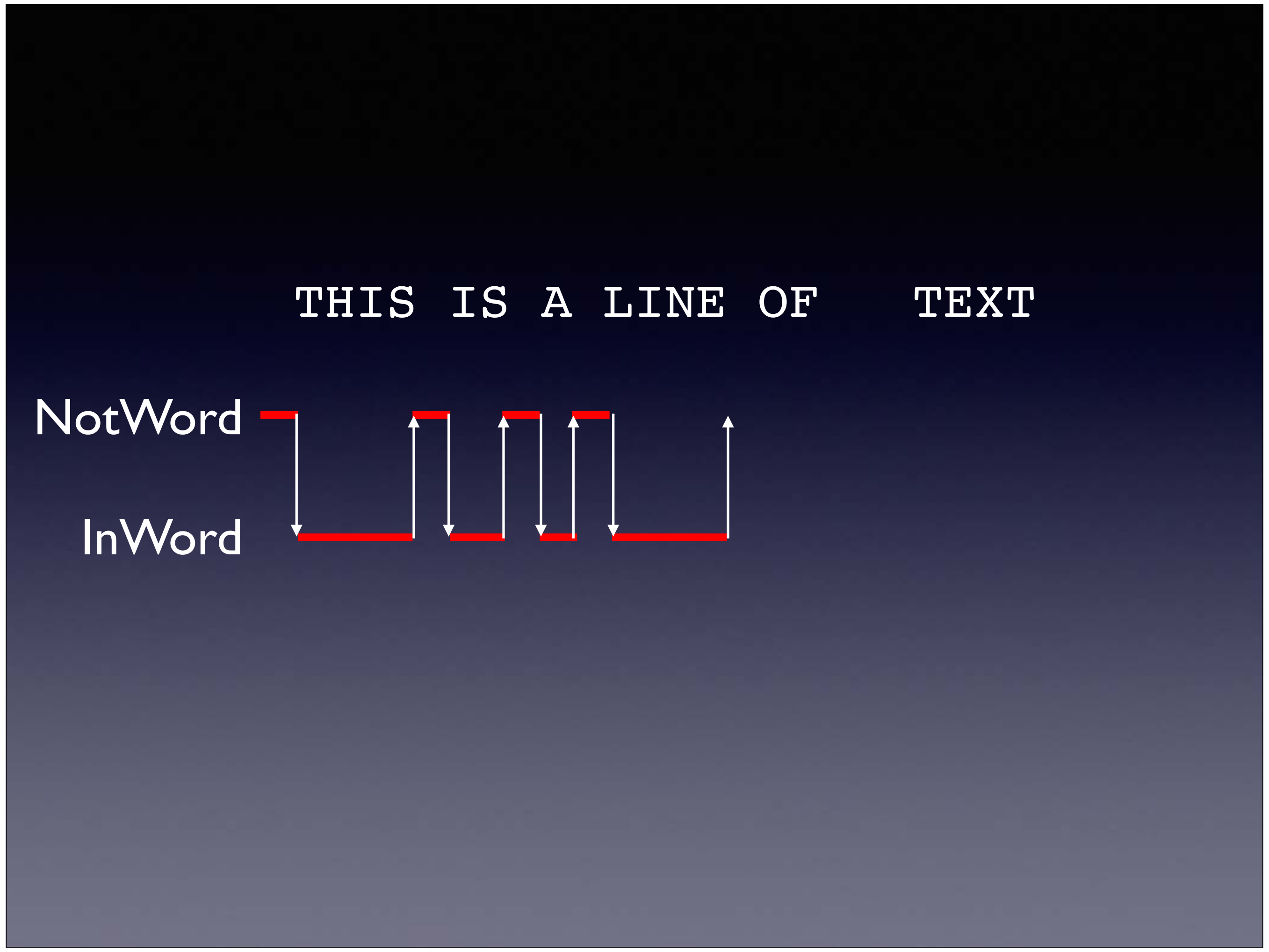


```
THIS IS A LINE OF    TEXT
```

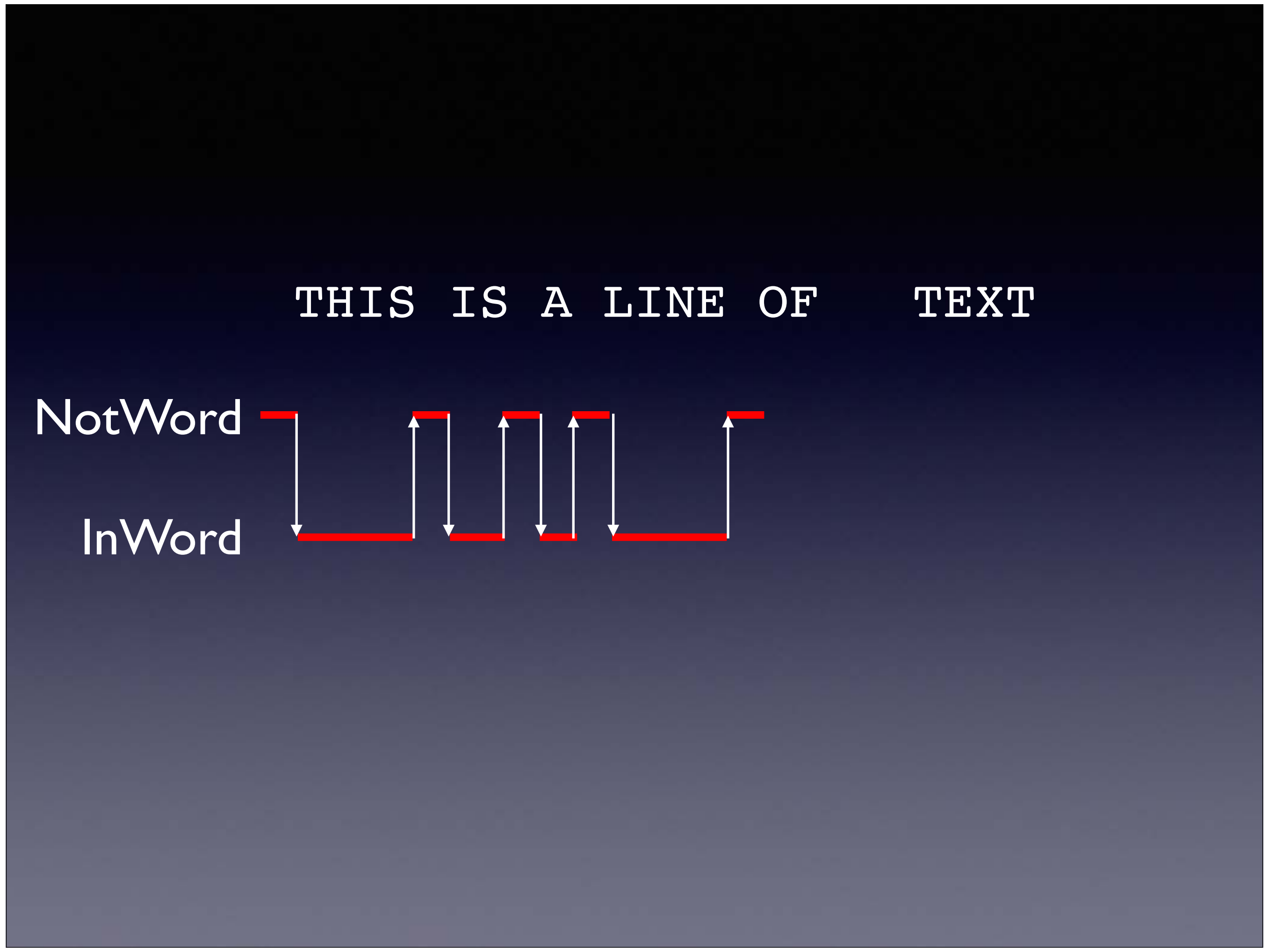


```
THIS IS A LINE OF    TEXT
```

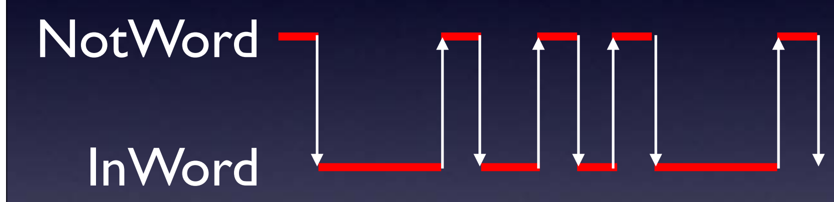


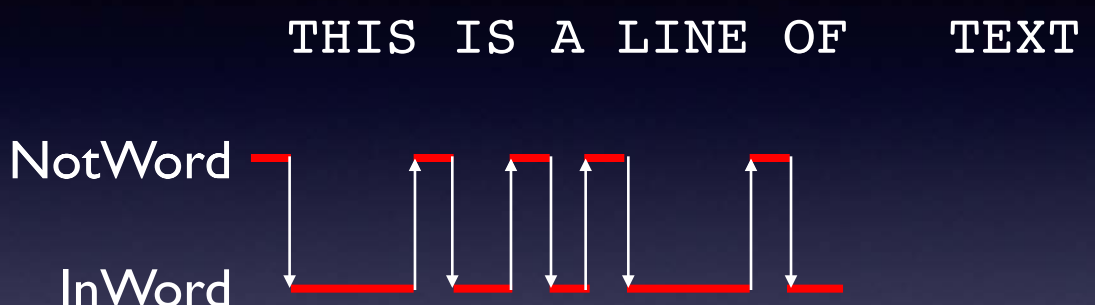


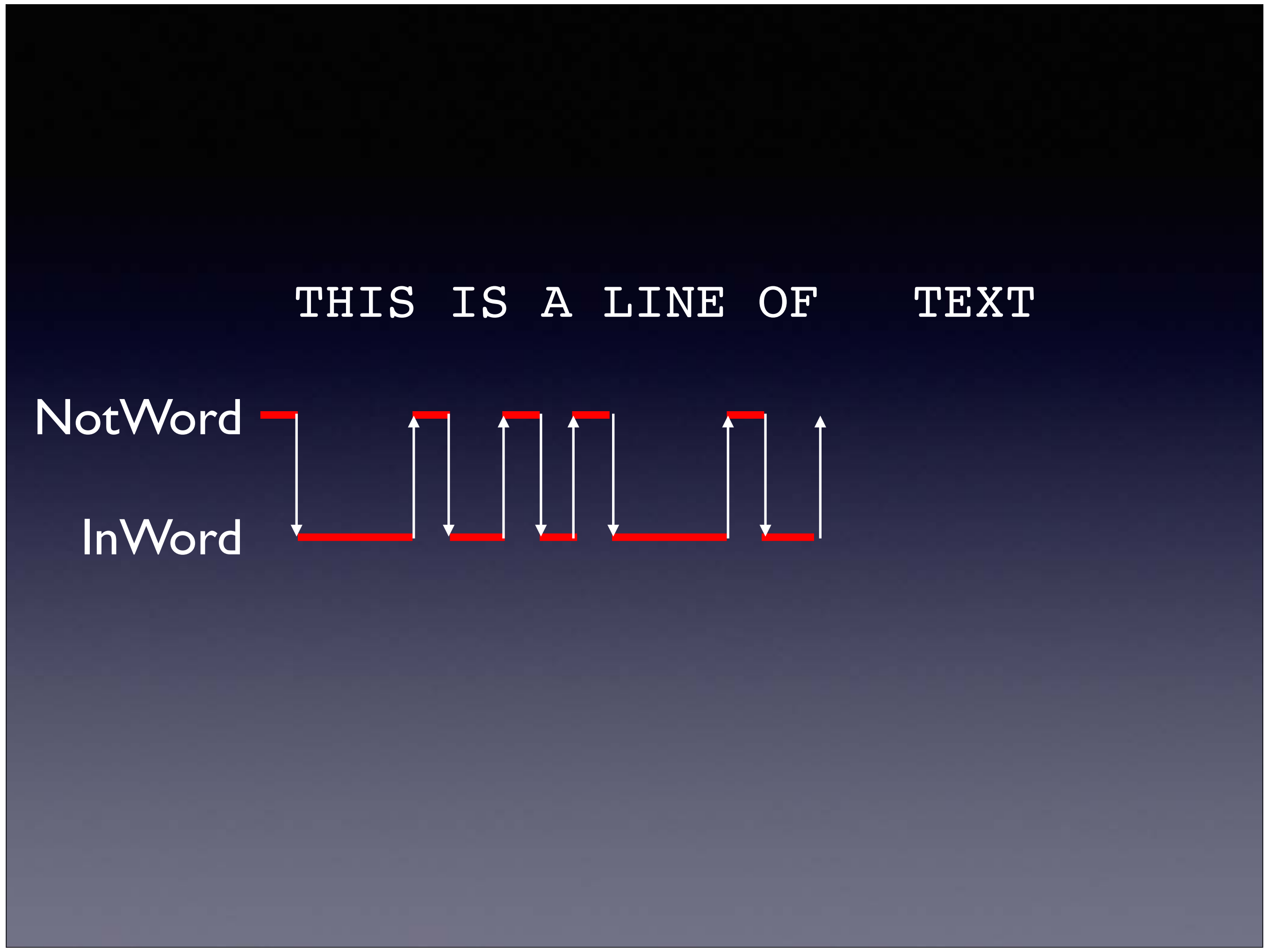
```
THIS IS A LINE OF    TEXT
```


THIS IS A LINE OF   TEXT

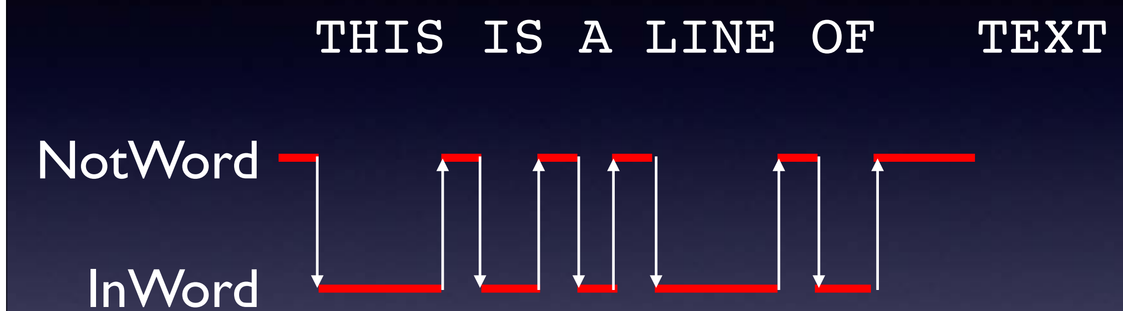
THIS IS A LINE OF   TEXT



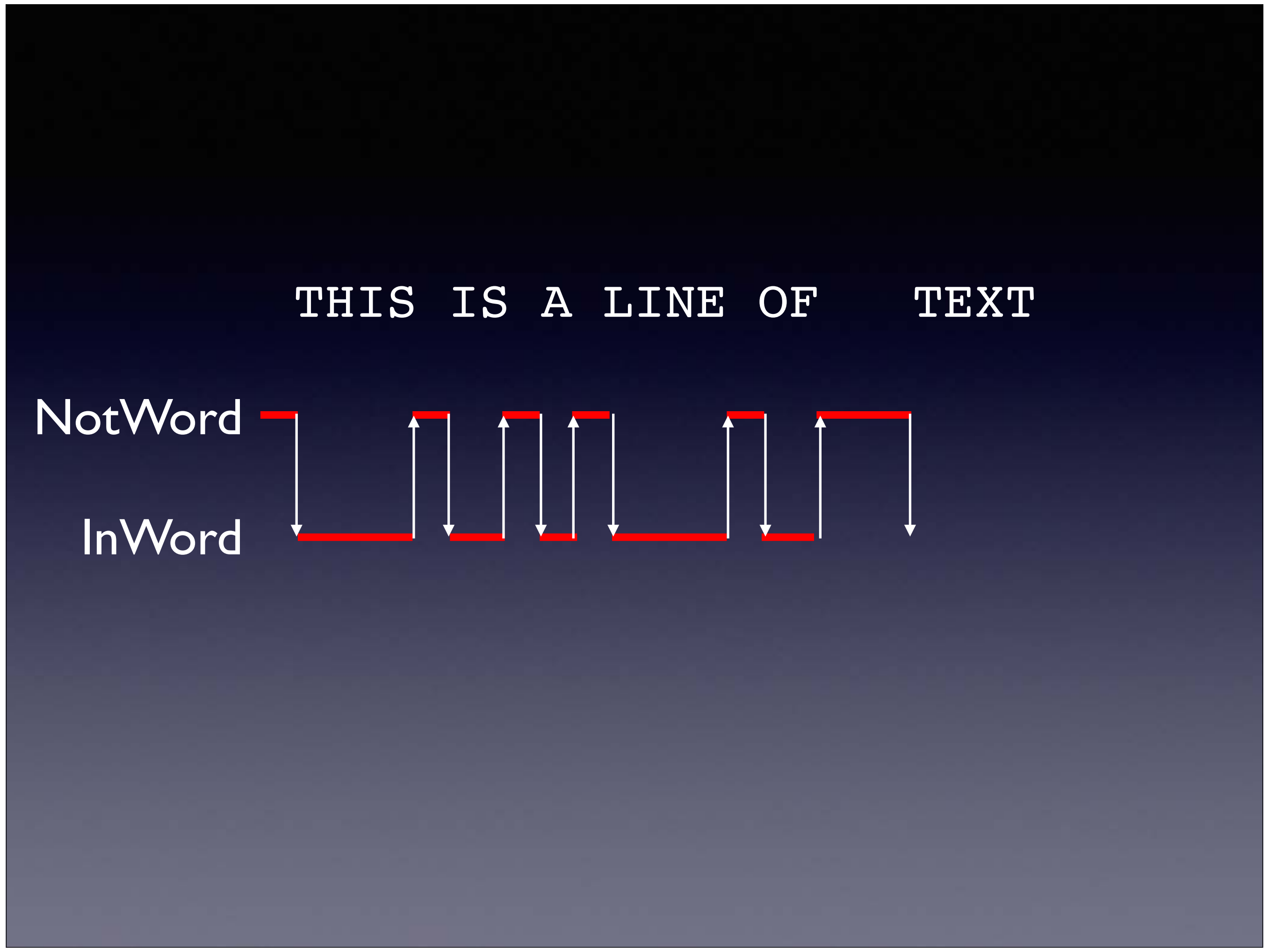

THIS IS A LINE OF   TEXT
NotWord
InWord

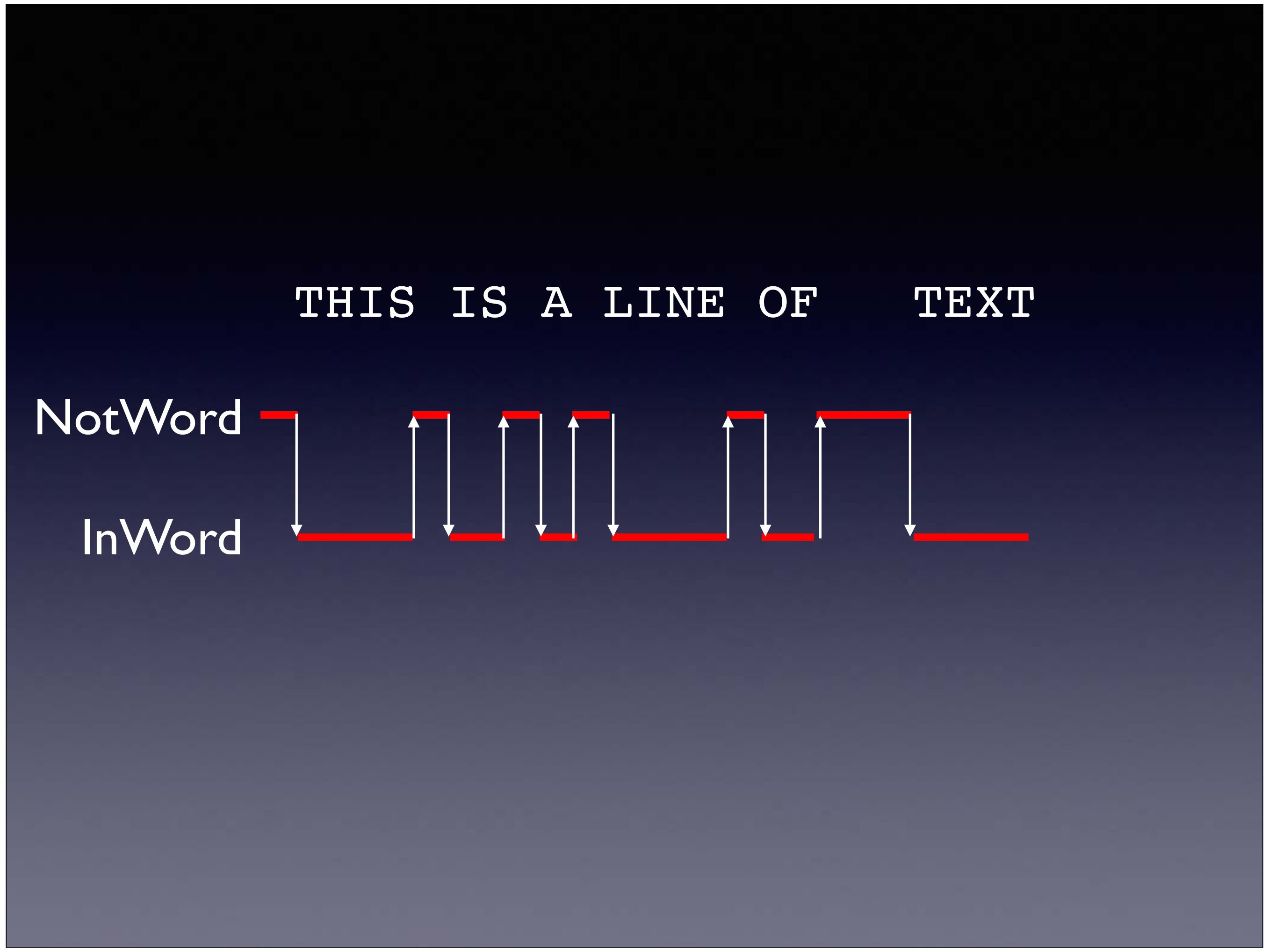


```
THIS IS A LINE OF    TEXT
```



THIS IS A LINE OF   TEXT
NotWord
InWord


THIS IS A LINE OF   TEXT
NotWord
InWord


THIS IS A LINE OF   TEXT
NotWord
InWord

State Transition

- Count words by counting the transitions from **not in word** to **in word** state
- Or vice versa, but final transition may be missing
- How do we model state?

State in Programs

- Easy…
- Store it in a variable
- If the variable contains one value (e.g. 0), then we are not in a word
- If it contains another value (e.g. 1), then we are in a word

Defining names for Values

- We are going to use the numbers `0` and `1` to represent the states
- But it won't necessary be clear what they mean
- So we'll give them a name – `OUT` and `IN`
- Could use a variable…
- But the values will never vary…

Constants

- Only wanting to name the values
- So a variable doesn’t really make sense
- Can be slower to access a variable than a value on some machines
- We just want to define the name to be that value

`#define`

- The C pre-processor lets us do this using the `#define` directive
  ```
  #define OUTWORD 0
  #define INWORD  1
  ```
- The pre-processor replaces all occurrences of the name with the literal value
- C compiler only ever sees the value

Word State

- Initialize state variable to `OUTWORD` state
- If we find a letter,
  - If we are in `OUTWORD` state, set variable to `INWORD` state and increment word count
  - If we are in `INWORD` state, stay in `INWORD` state

Word State

- If we find a space,
  - If `INWORD` state, set to `OUTWORD` state
  - If `OUTWORD` state, stay in `OUTWORD` state
- Note we can, make an optimization here
- If we find a space, set state to `OUTWORD`
- Removes a compare from the program

Space comparison

- Lots of compares mentioned in the previous slides
- Can remove some of them…
- Already mentioned spaces and `OUTWORD` state
- Defined letters as *not* spaces
- So can use an `if…else`

if space, do this, else its a letter so do that

Corner cases

- Program will work for most input
- But there are some cases where it might not do what we expect
- Need to develop ways of testing the edges
- If the program works at the edges and corners, then it'll probably work for most input…

Corners

- Is a line a line if it doesn't end with a newline?
- What if there is only one word in the file and no spaces?
- What if the file is entirely spaces?

Increment

- Mentioned the increment operators
- C provides two
  - Pre-increment (`++x`)
  - Post-increment (`x++`)
- Difference is to do with what is the result of the calculation

Increment

- Both increment the value *stored* in `x` by one
- Preincrement gives the new value as the result
- Postincrement gives the old value as the result
- Can see this by using `printf()`

More loops: For, and Nests

Steven R. Bagley

Recap

- Programs are a series of statements
- Defined in functions
- Can call functions to alter program flow
- `if` statement can determine whether code gets run
- `while` loop can execute code multiple times

Increment

- Mentioned the increment operators
- C provides two
  - Pre-increment (`++x`)
  - Post-increment (`x++`)
- Difference is to do with what is the result of the calculation

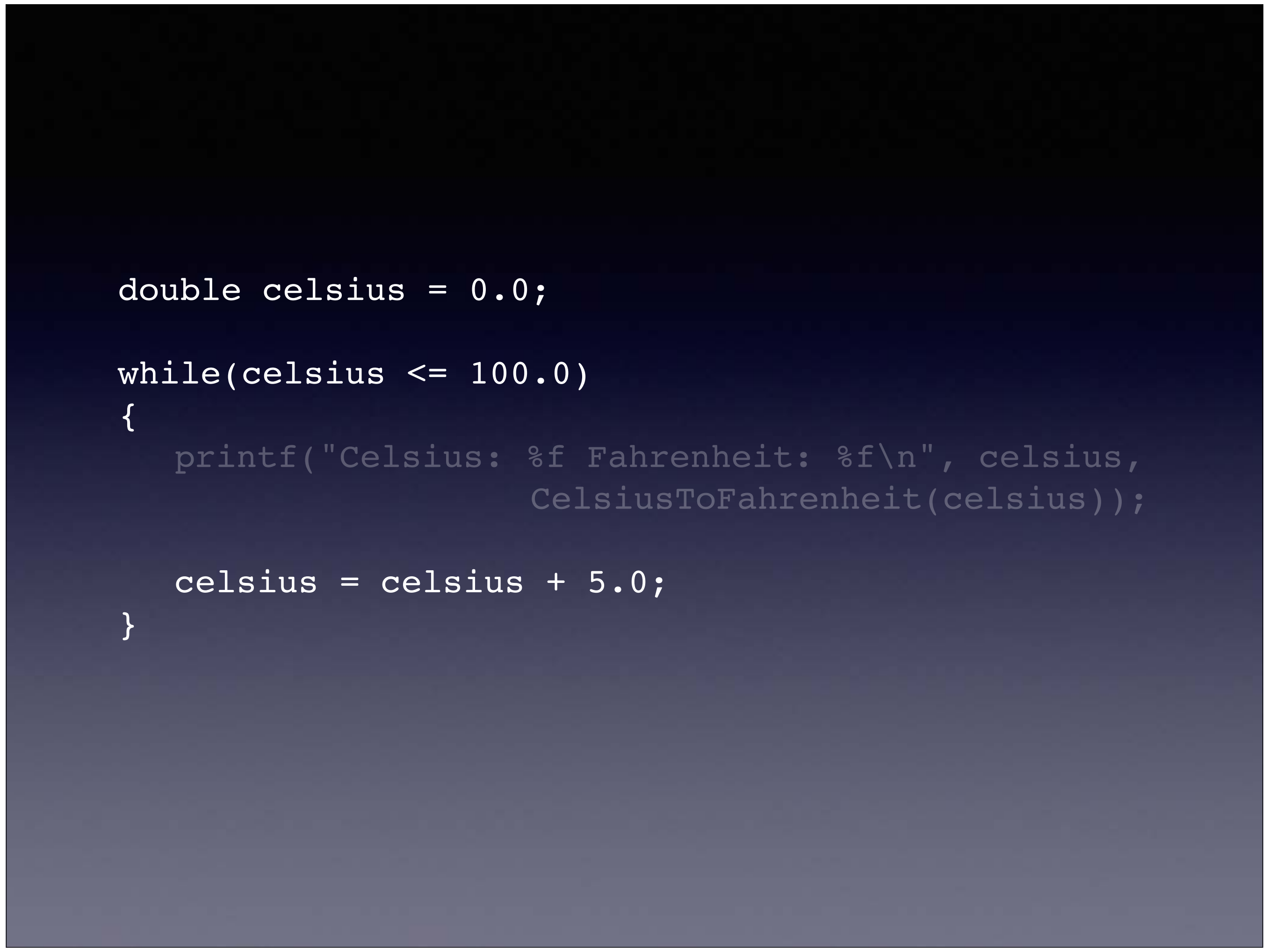
Increment

- Both increment the value *stored* in `x` by one
- Preincrement gives the new value as the result
- Postincrement gives the old value as the result
- Can see this by using `printf()`
- Equivalent for decrement (`x--`/`--x`)

Loops

- Loops are very useful
- Allow us to repeatedly do some code
- Until a specific condition is met
  e.g. counting characters in a file
- Often used to count through a sequence of values

```
double celsius = 0.0;

while(celsius <= 100.0)
{
   printf("Celsius: %f Fahrenheit: %f\n", celsius,
                     CelsiusToFahrenheit(celsius));

   celsius = celsius + 5.0;
}
```

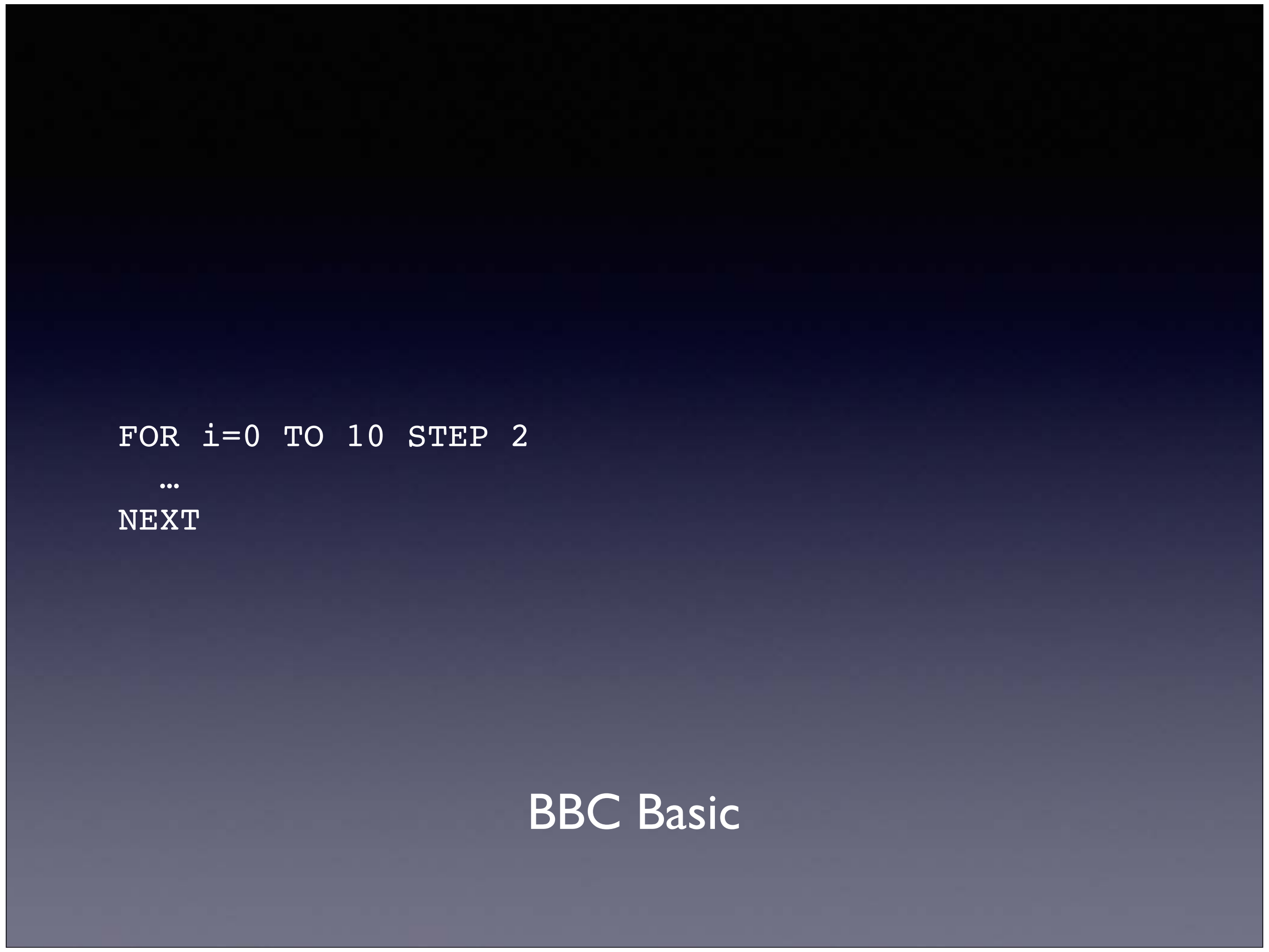


```
double celsius = 0.0;

while(celsius <= 100.0)
{
   printf("Celsius: %f Fahrenheit: %f\n", celsius,
                       CelsiusToFahrenheit(celsius));

   celsius = celsius + 5.0;
}
```

Tend to write lots of loops with this structure

Counting Loops

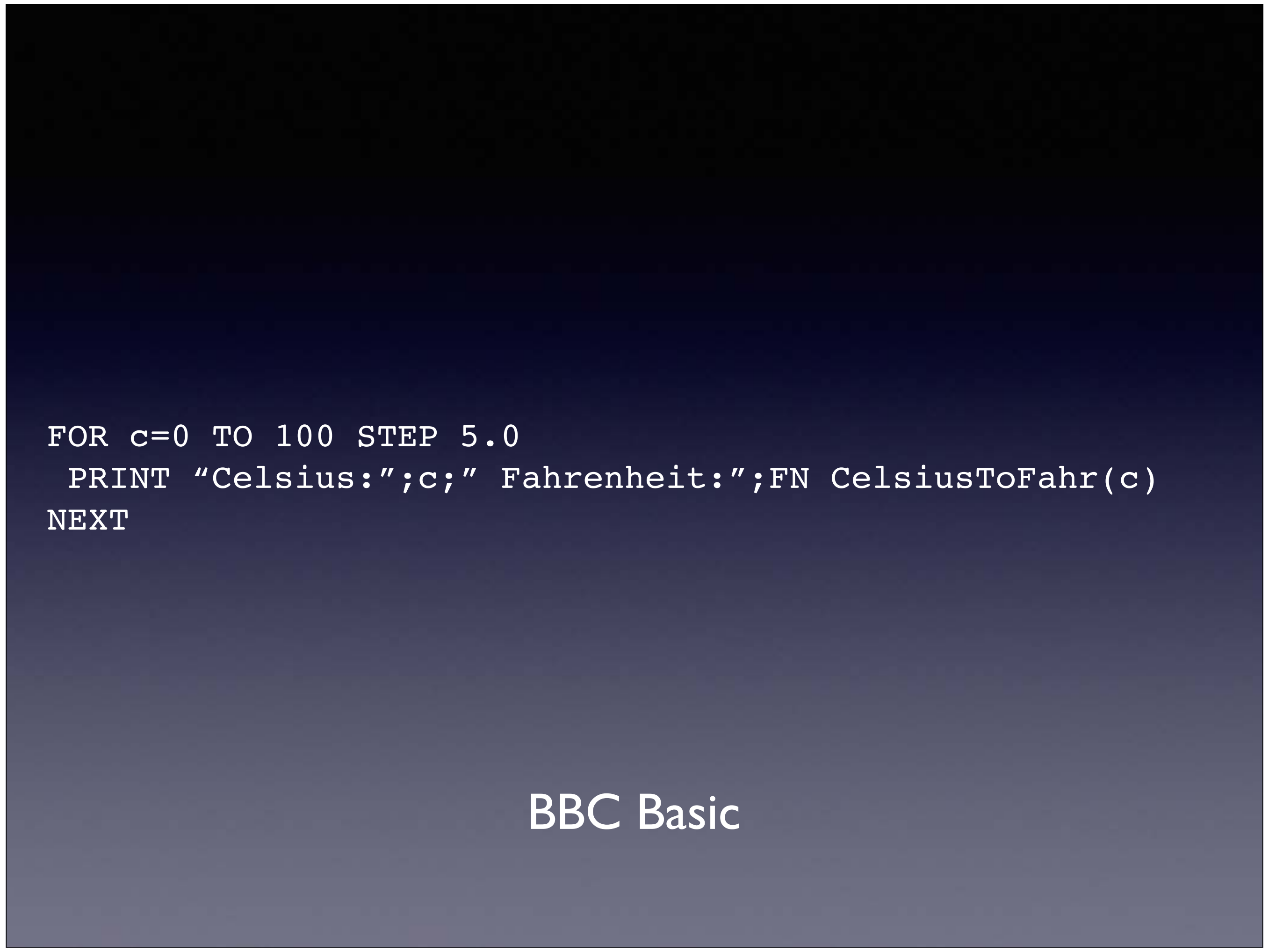
- Very common structure
- Start at one value
- Count up (or down) in steps
- Until you reach some value
- Lots of programming languages provide direct support for this — the `for` loop



```
FOR i=0 TO 10 STEP 2
  …
NEXT
```

BBC Basic

The BBC Basic FOR loop, which counts in twos from 0 to 10 giving 6 values (0,2,4,6,8,10)



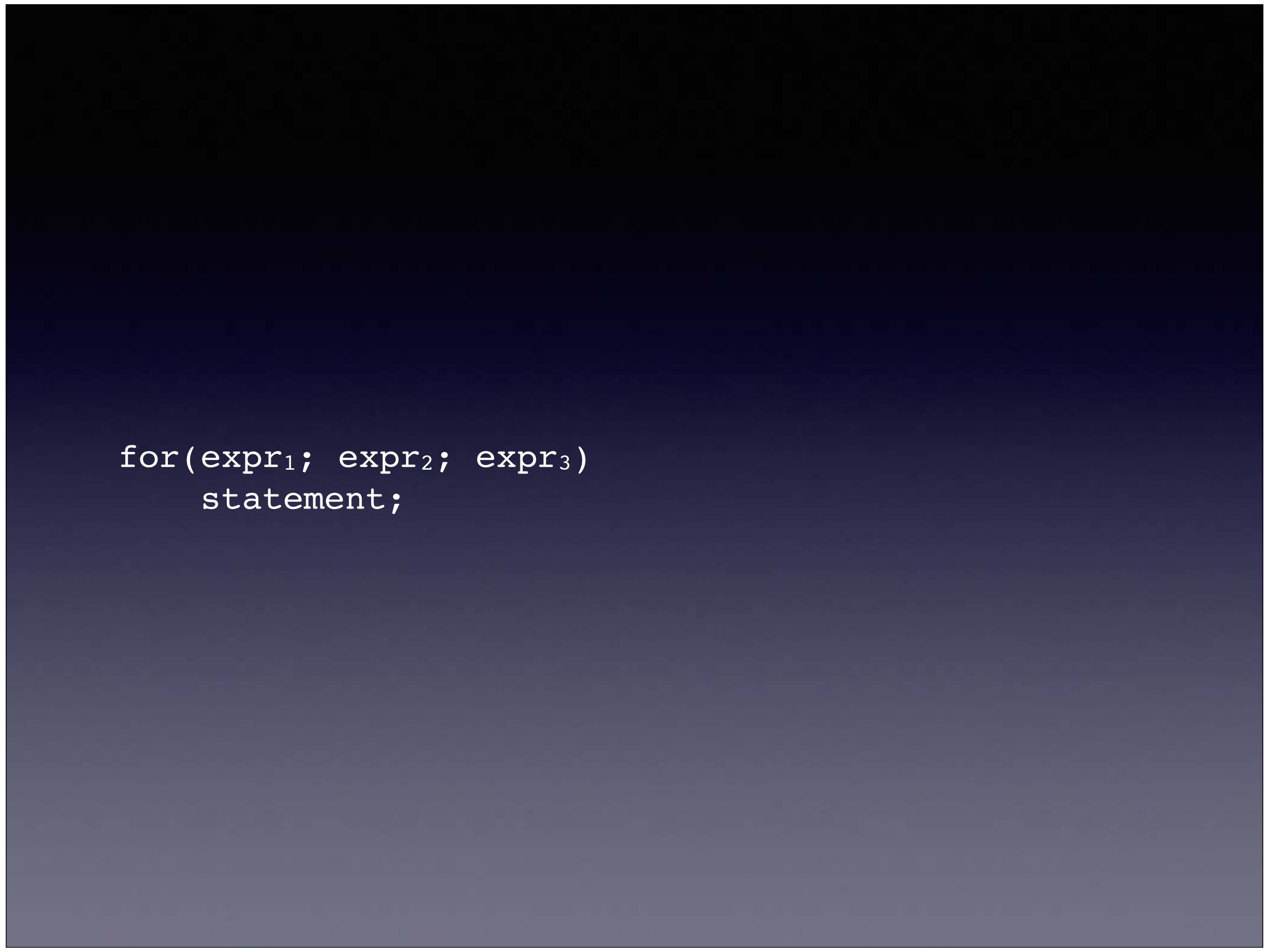
```
FOR c=0 TO 100 STEP 5.0
 PRINT “Celsius:”;c;” Fahrenheit:”;FN CelsiusToFahr(c)
NEXT
```

BBC Basic

BBC BASIC equivalent of our while loop...

C’s for loop

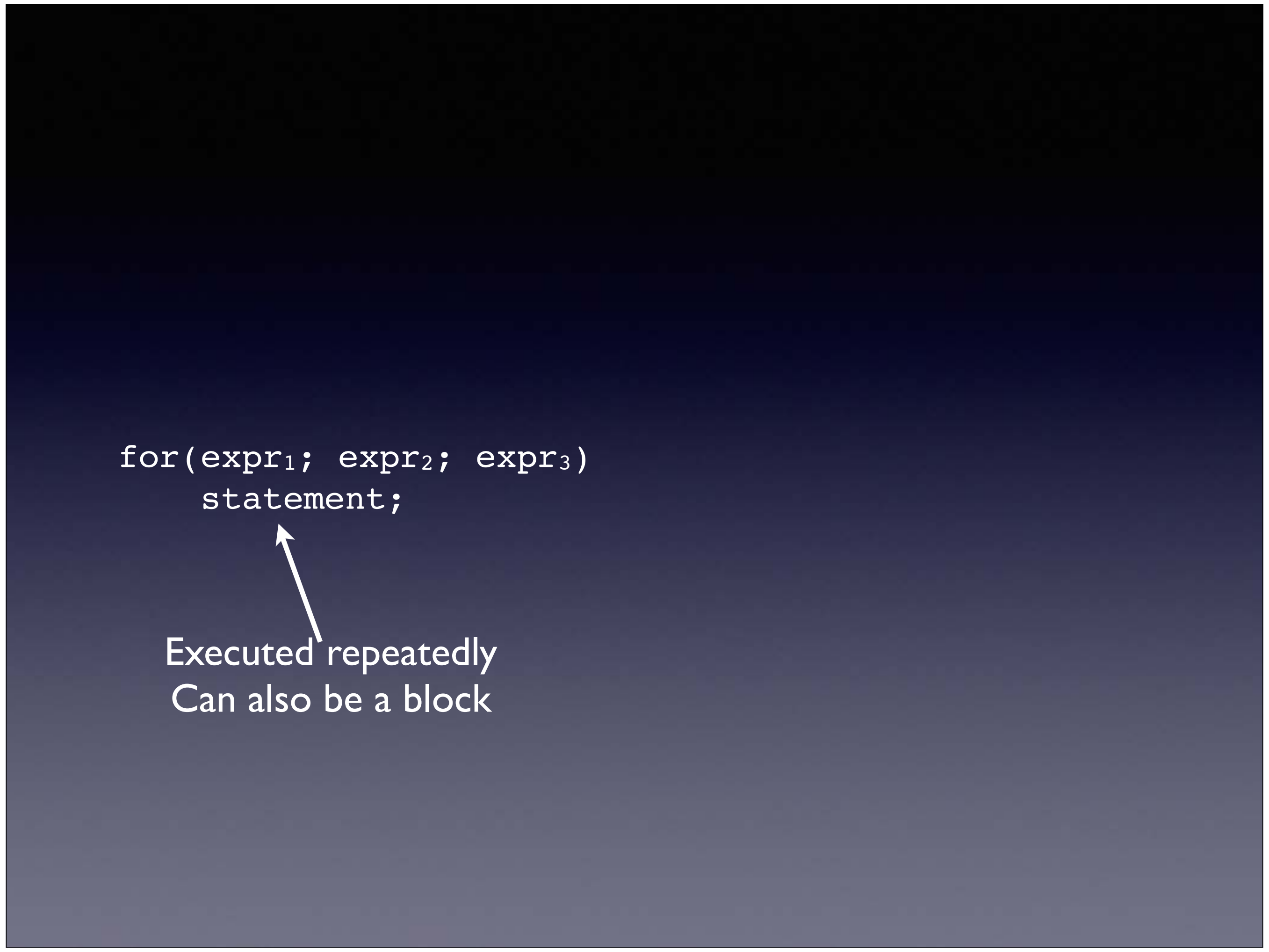
- C also provides a `for` loop…
- It’s syntax is more general than that in other languages



```
for(expr1; expr2; expr3)
    statement;
```

The C for loop...

expr is just a C expression, any valid C instruct, an assignment, a conditional a function call...

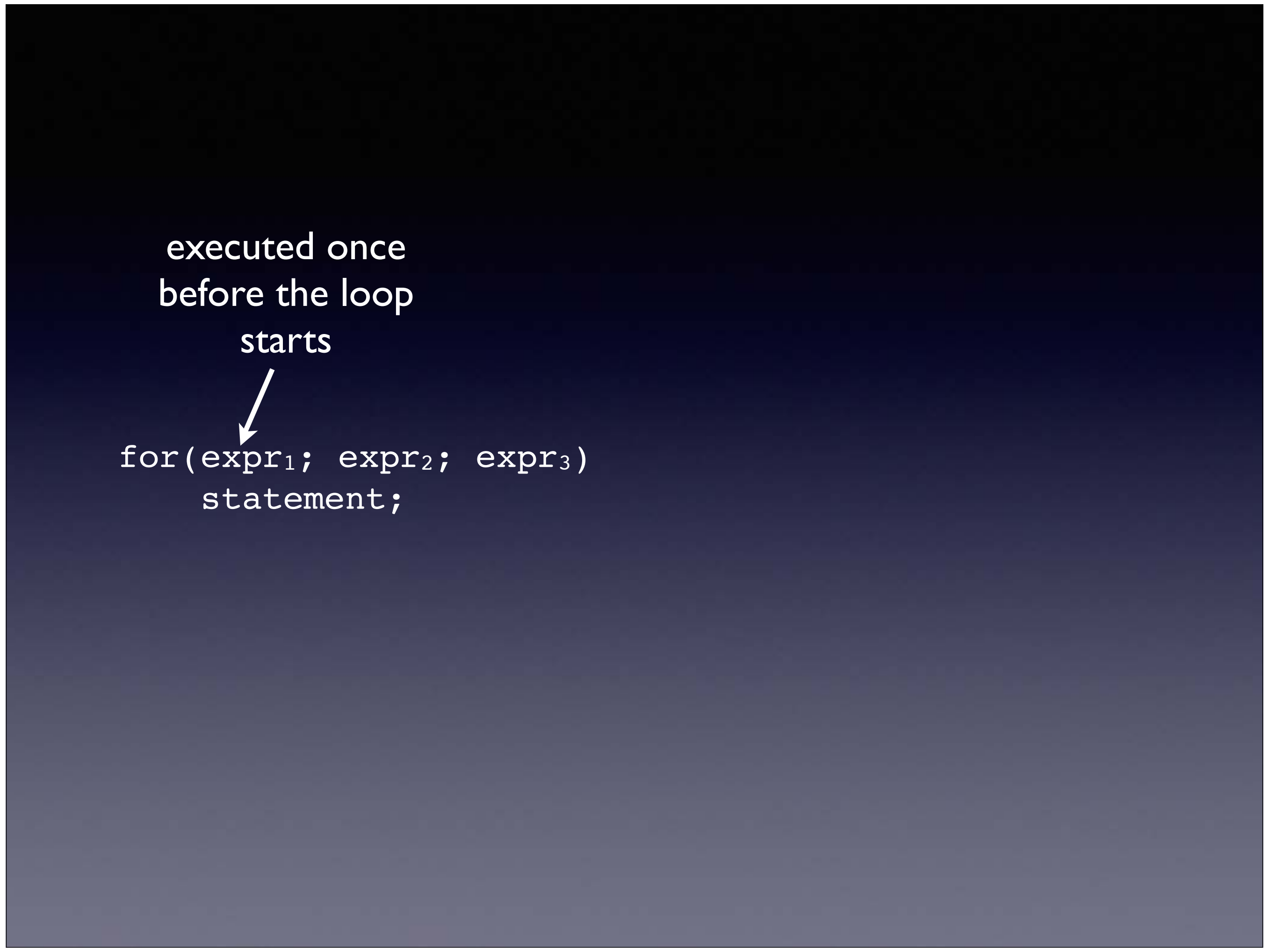


```
for(expr1; expr2; expr3)
    statement;
```

Executed repeatedly
Can also be a block

The C for loop...

expr is just a C expression, any valid C instruct, an assignment, a conditional a function call...



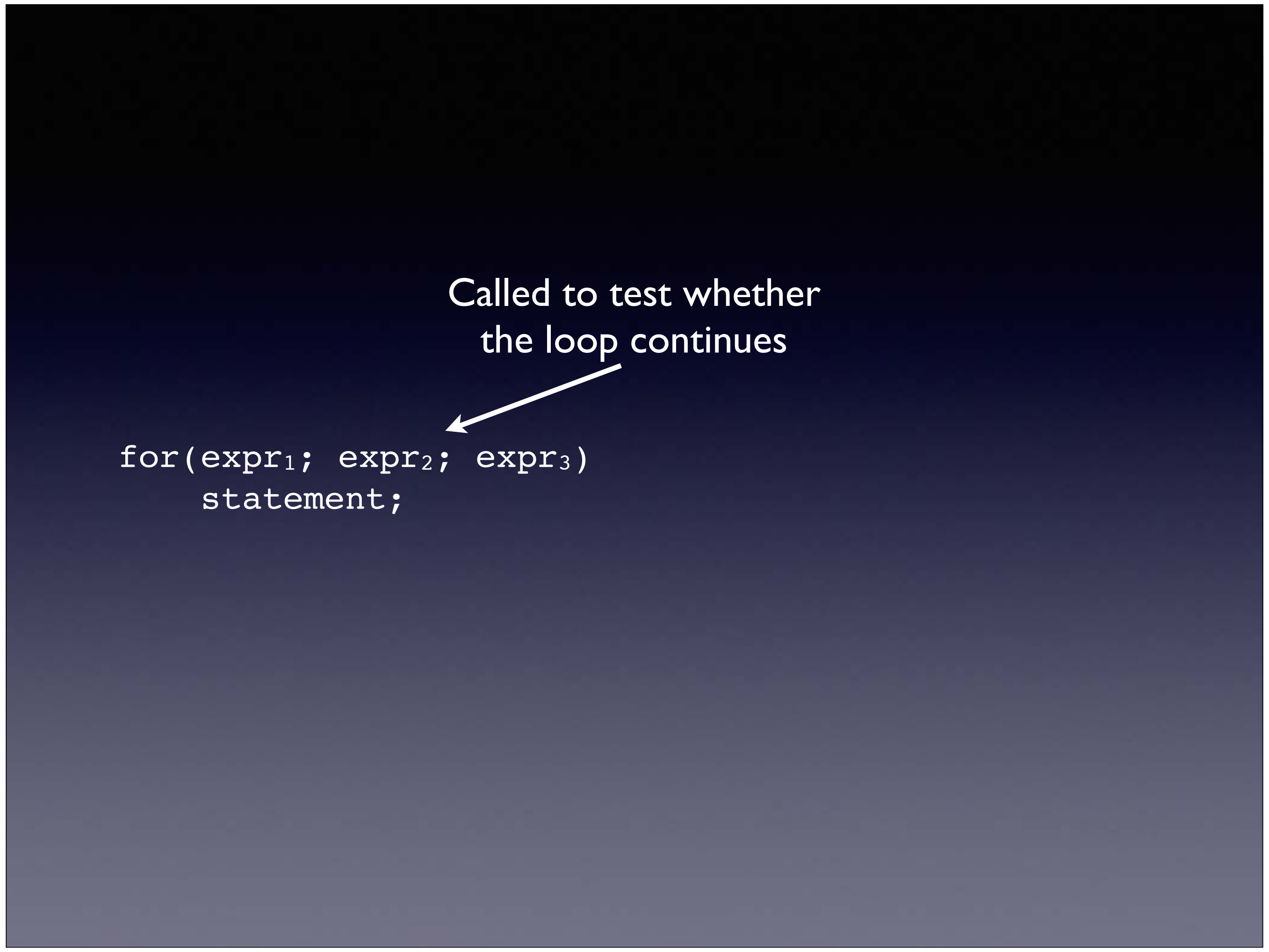
executed once before the loop starts

$\texttt{for(expr}_1\texttt{; expr}_2\texttt{; expr}_3\texttt{)}$

$\quad\texttt{statement;}$

The C for loop...

expr is just a C expression, any valid C instruct, an assignment, a conditional a function call...

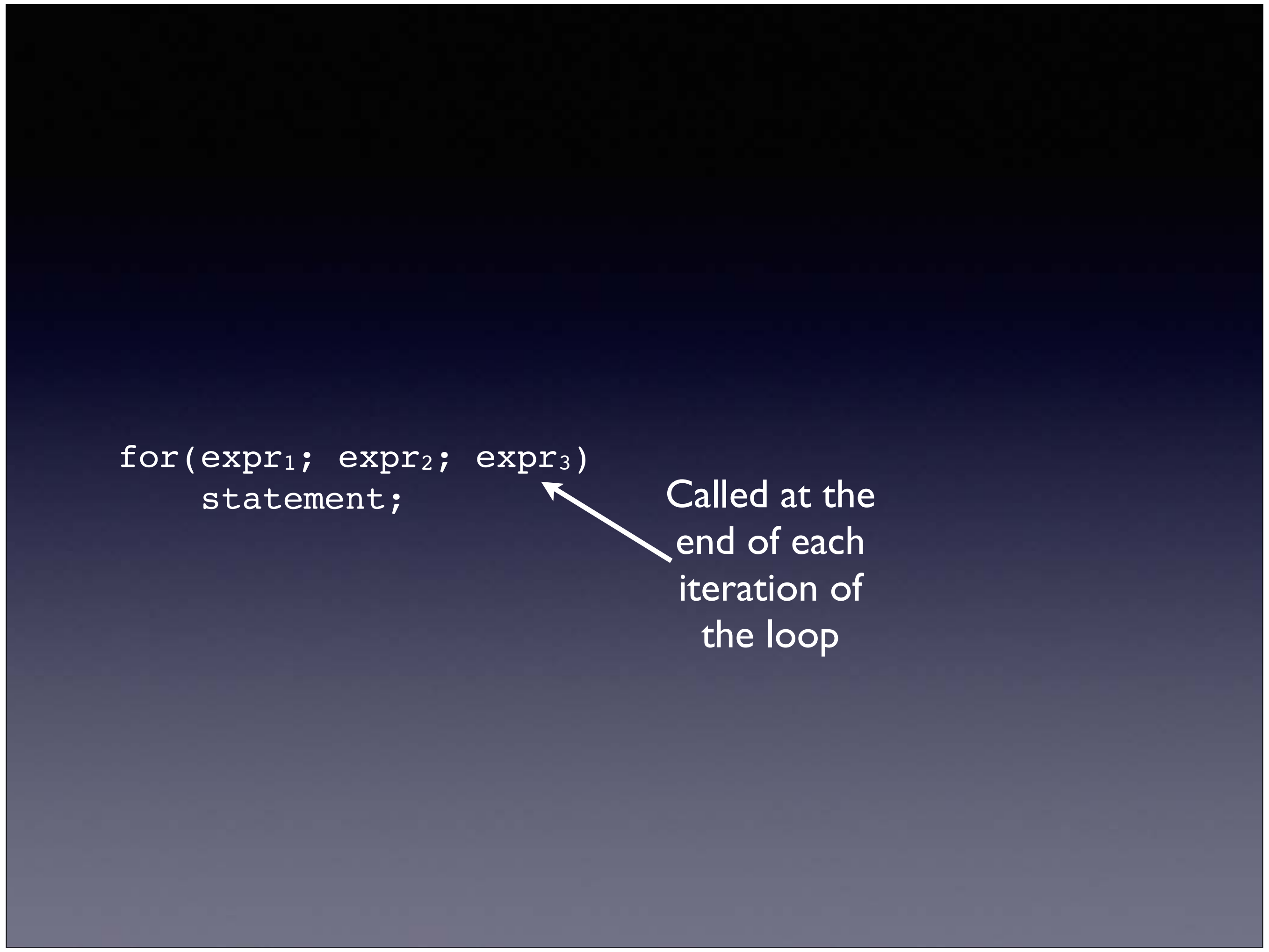


Called to test whether the loop continues

```
for(expr1; expr2; expr3)
    statement;
```

The C for loop...

expr is just a C expression, any valid C instruct, an assignment, a conditional a function call...

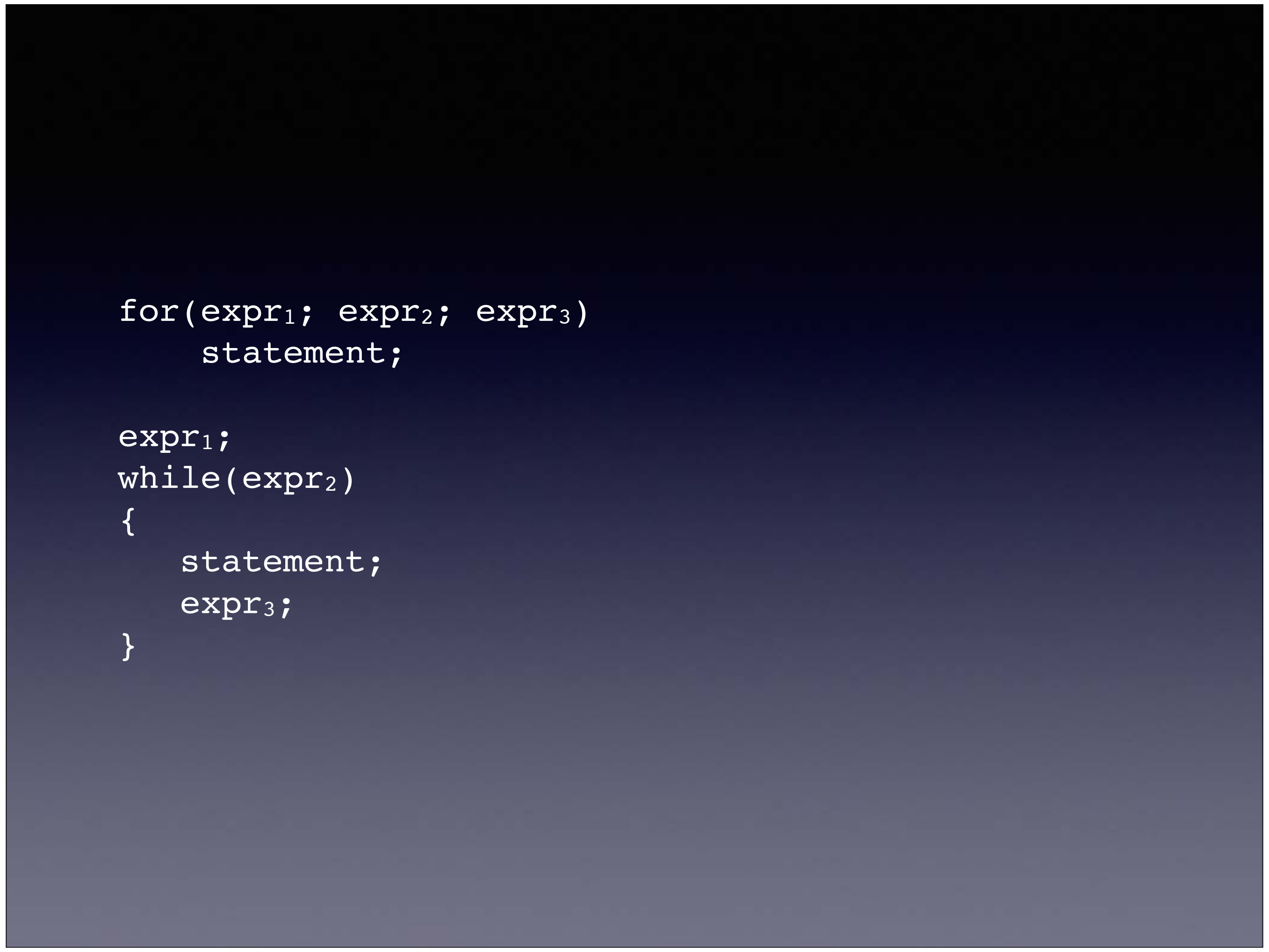


```
for(expr1; expr2; expr3)
     statement;
```

Called at the end of each iteration of the loop

The C for loop...

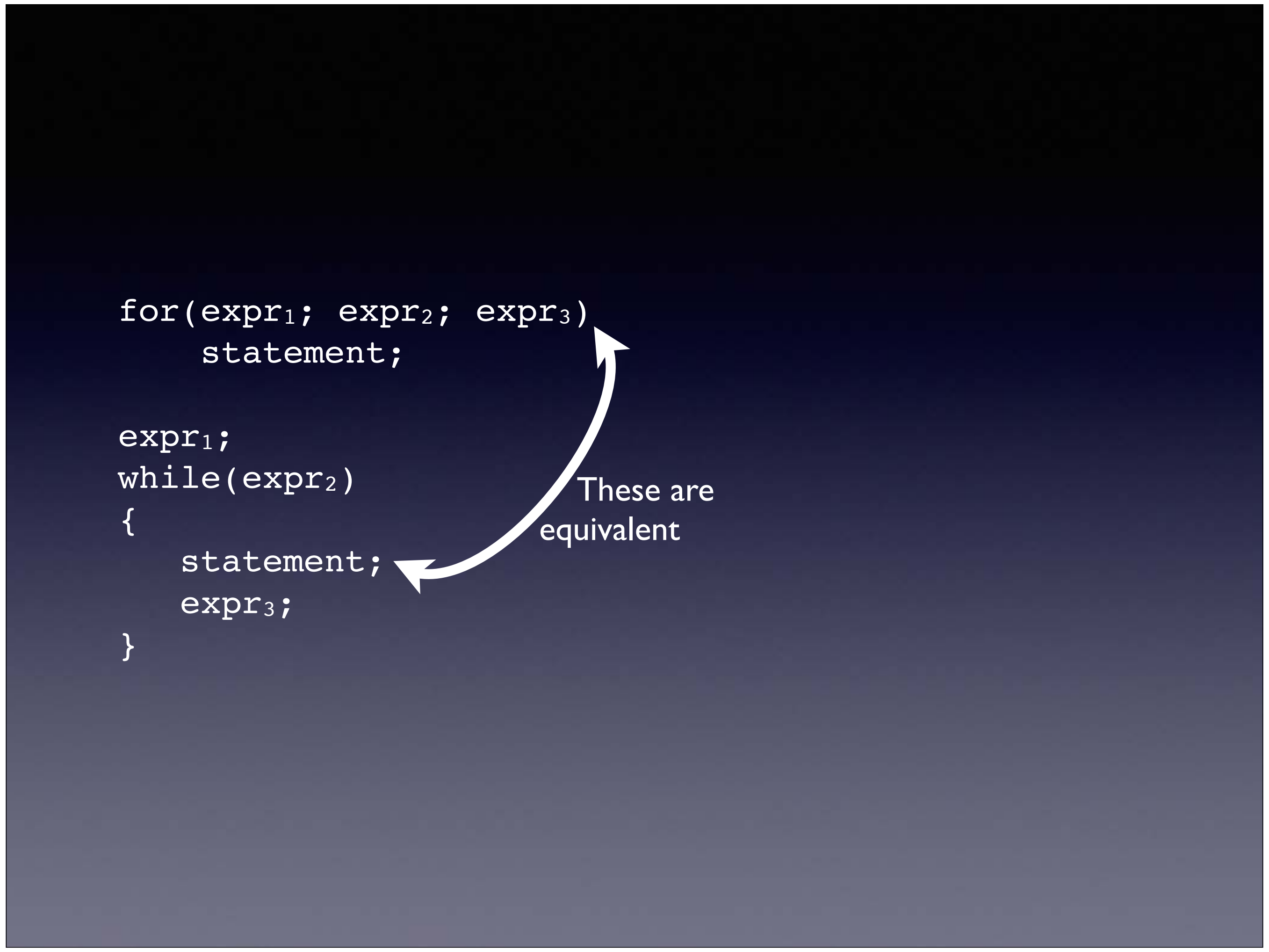
expr is just a C expression, any valid C instruct, an assignment, a conditional a function call...



```
for(expr1; expr2; expr3)
    statement;

expr1;
while(expr2)
{
   statement;
   expr3;
}
```

The equivalent while loop to any for loop

Go show how to rewrite the celsius program



```
for(expr1; expr2; expr3)
    statement;

expr1;
while(expr2)
{
   statement;
   expr3;
}
```

The equivalent while loop to any for loop

Go show how to rewrite the celsius program

`for` loop

- Four parts to the `for` loop
  - Block of code execute
  - Expression evaluated once at the start usually an initial assignment
  - An expression (conditional) to test whether the loop ends
  - An expression executed at the end of every iteration (another assignment)

for loops

- Expressions are optional
- But semi-colons aren’t
- What does `for(;;)` mean?

for loops

- Expressions are optional
- But semi-colons aren’t
- What does `for(;;)` mean?
- Loop forever…

Escaping…

- C provides away to break out of a loop early
- Including infinite loops
- Uses the `break` instruction
- Almost always placed inside a conditional…
- Escapes from the current loop only

`for` or `while`?

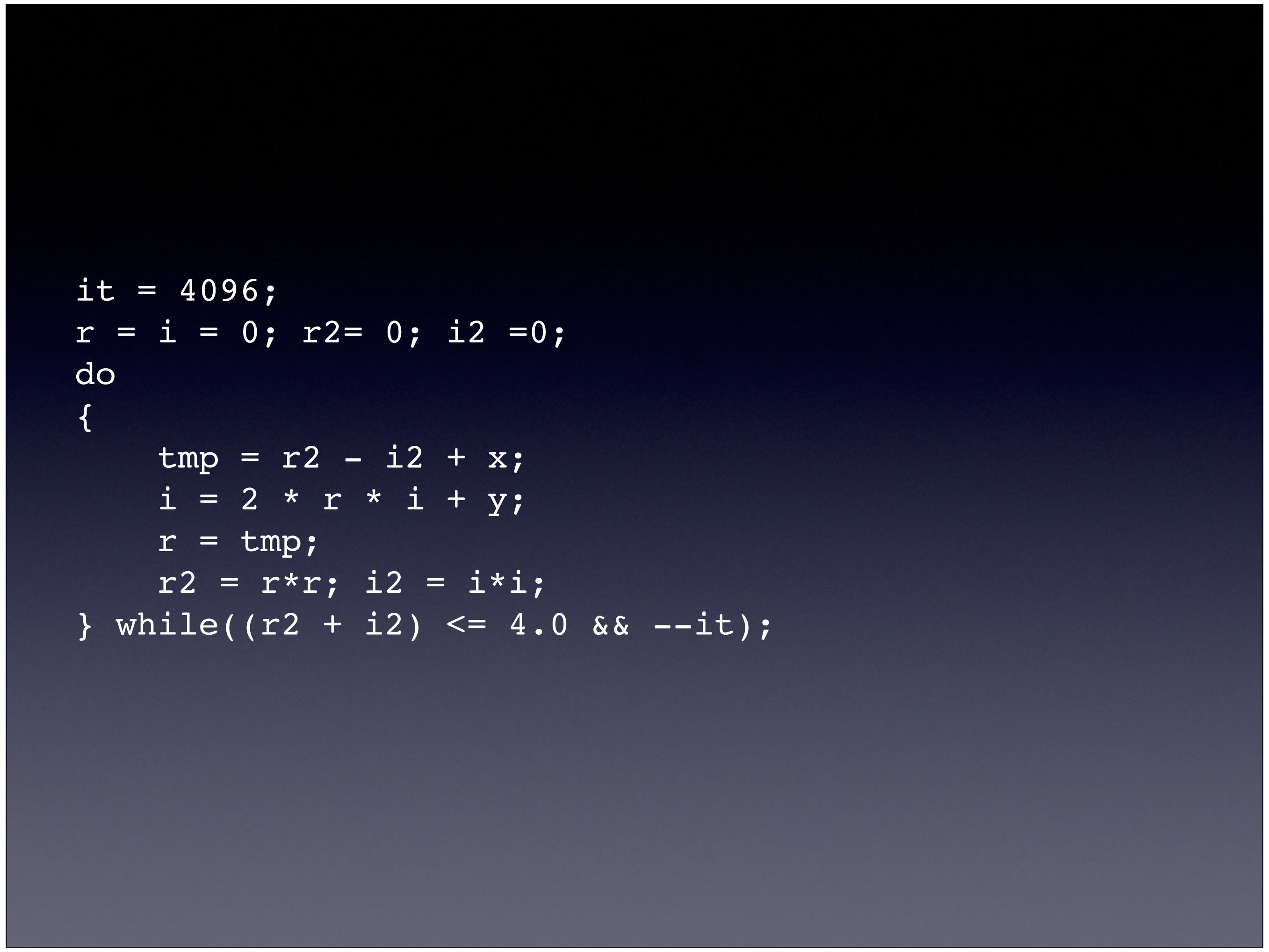
- Which do you use?
- Large personal preference, but some rules of thumb
- If there's a simple initialization, and increment then use `for`
- If there's no initialization or increment, use `while`

Nested loops

- You can embed one loop inside another
- Including of different types
- The inner loop will be executed inside every iteration of the outer loop
- Can be very useful for traversing two-dimensional things (e.g. images)

Do-while loop

- `while` loop tests the conditional first
- This means it is possible for the code in the loop never to execute
- C also provides another version
  `do { … } while(condition)`
- The test in this is done at the end so loop always executes at least once



```
it = 4096;
r = i = 0; r2= 0; i2 =0;
do
{
    tmp = r2 - i2 + x;
    i = 2 * r * i + y;
    r = tmp;
    r2 = r*r; i2 = i*i;
} while((r2 + i2) <= 4.0 && --it);
```

Mandlebrot loop…