Recursion and other Bits and Pieces

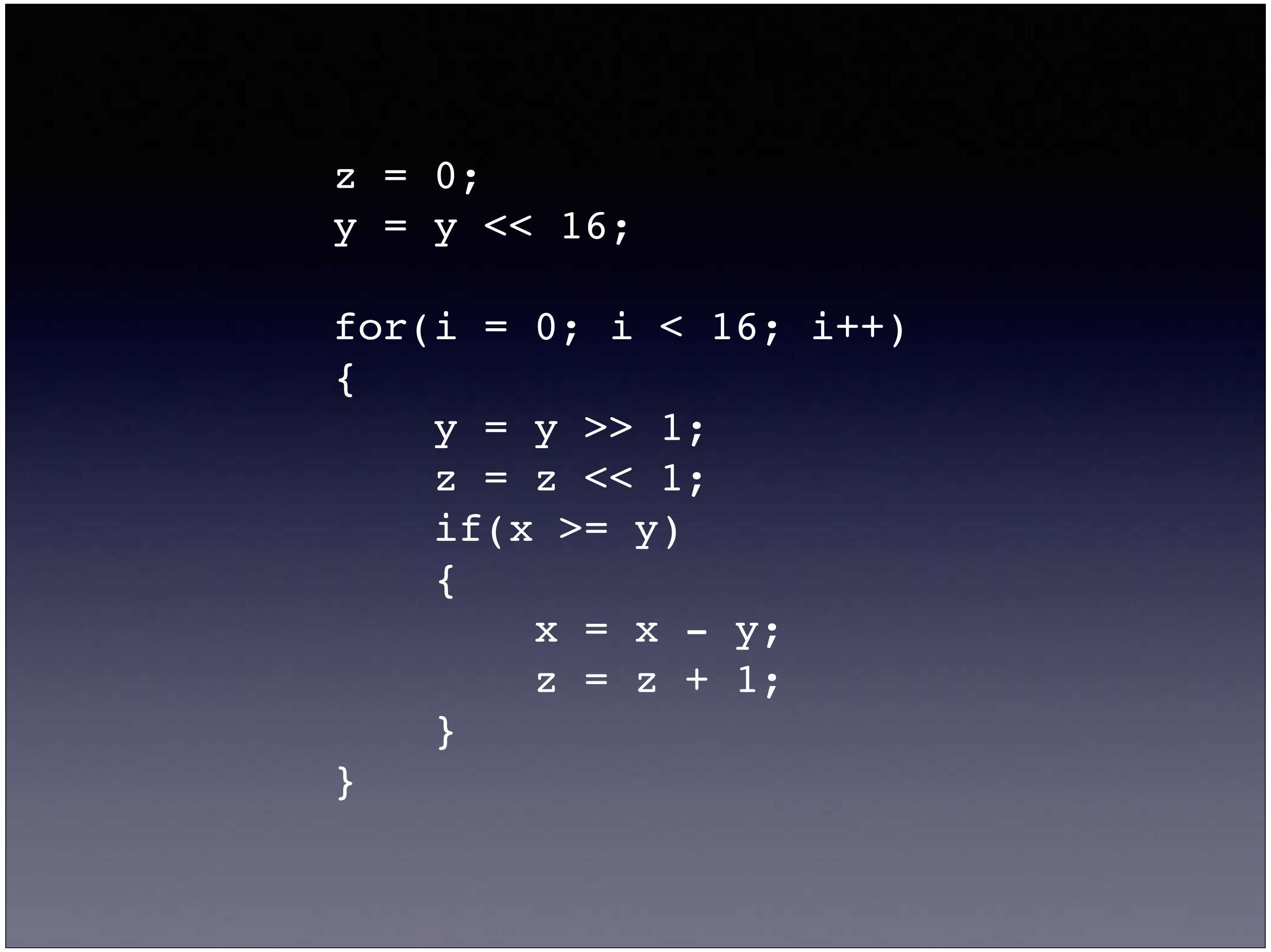
Steven R. Bagley

Recap

- Seen how we can control program flow
- Functions
- Conditionals
- Loops
- But there is another method we haven't looked at yet…

Iteration

- Iteration repeatedly does something until
  - We reach a target
  - Have done it a set number of times
- Fancy name for a ‘loop’
- Used a lot in Computer Science



```
z = 0;
y = y << 16;

for(i = 0; i < 16; i++)
{
    y = y >> 1;
    z = z << 1;
    if(x >= y)
    {
        x = x - y;
        z = z + 1;
    }
}
```

An example of iteration -- any loop would have done.

Repetition

- While the loop is a common method of repeating a task in a program
- It is not the only method
- We can create loops by using a function…

Functions

- Suppose we have a function…

```
void MyFunc()
{
    printf(“Hello World\n”);

}
```

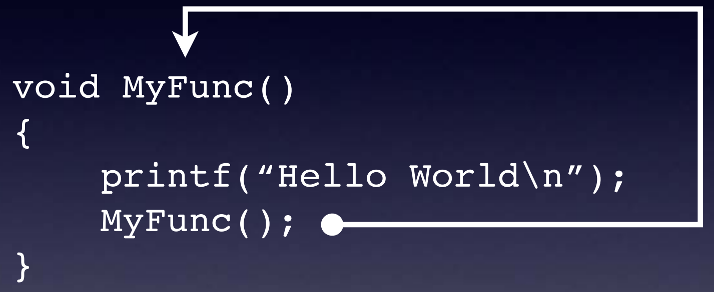
- When call it will print out ‘Hello World’
- Then return to whatever called it

Functions

- But what if we make the function call itself?

```
void MyFunc()
{
    printf(“Hello World\n”);
    MyFunc();
}
```

- What happens now?



```
void MyFunc()
{
    printf(“Hello World\n”);
    MyFunc();
}
```

Recursive Function

- Each time the function calls itself
- Its starts again from the beginning
- Executing the `printf` repeatedly
- Effectively creating an infinite loop…
- Won’t stop unless the program is killed

Recursion

- This is known as recursion…
- We've defined the function, `MyFunc()`, in terms of itself
- Very powerful…
- Used a lot in computer science
- In fact, some languages don't have loops at all, they just use recursion…

Base-Case

- Generally, we need to be able to stop the recursion
- This is done by providing a **Base Case**
- Some case during which the function won't call itself
- This usually depends on some input value

Recursive Step

- Still not that useful, function either loops infinitely, or does nothing
- However, if we change the recursive function call to pass `n-1` instead of `n`
- Then we find that the function prints `‘Hello World’` out `n` times

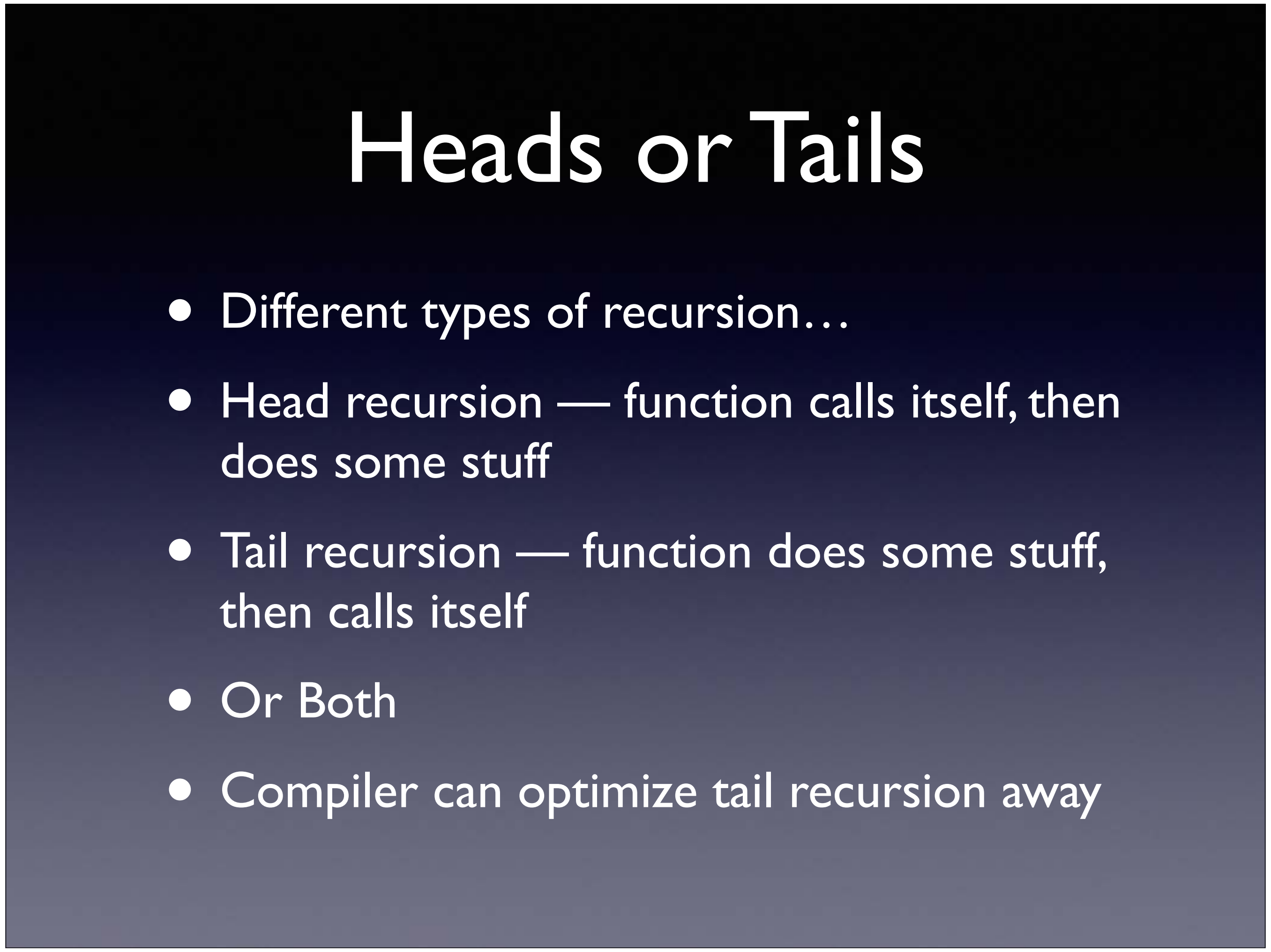
Recursion

- Suppose we call `MyFunc(3)`
- `MyFunc()` starts `n` does not equal `0`, so function runs
- Prints `Hello World`
- Calls `MyFunc(3-1)` (i.e. `MyFunc(2)`)
- And so on…

Recursion

- Eventually, `MyFunc()` will be called with `n` equal to `0`
- `MyFunc()` just returns passing control back to the previous function
- Which is also `MyFunc()`
- This then ends, passing control back up…
- Eventually, control gets passed back to whatever called `MyFunc()` in the first place

Modify MyFunc to show this by adding printfs everywhere



Heads or Tails

- Different types of recursion…
- Head recursion — function calls itself, then does some stuff
- Tail recursion — function does some stuff, then calls itself
- Or Both
- Compiler can optimize tail recursion away

Tail recursion — call to itself is the last thing in the function
Whether you want to use head or tail recursion depends on the order you need to do things

Warning

- Recursion can be dangerous
- If you don’t get the recursive step and the base case right
- You’ll end up in an infinite loop
- Well, infinite until the stack runs out…

Why recursion?

- So what…
- Why not just use a loop, this seems far too overcomplicated
- It is for a simple loop, but for other tasks the recursive implementation can be easier to write
- A good example of this is Fibonacci series

Fibonacci series

- `1 1 2 3 5 8 13 21 34 55 89 144 233…`
- Common mathematical series, that occurs in nature *a lot*
- But its defined in terms of itself
- Any Fibonacci number is defined as the sum of the two previous Fibonacci numbers
- `fibb n = fibb n-1 + Fibb n-2`

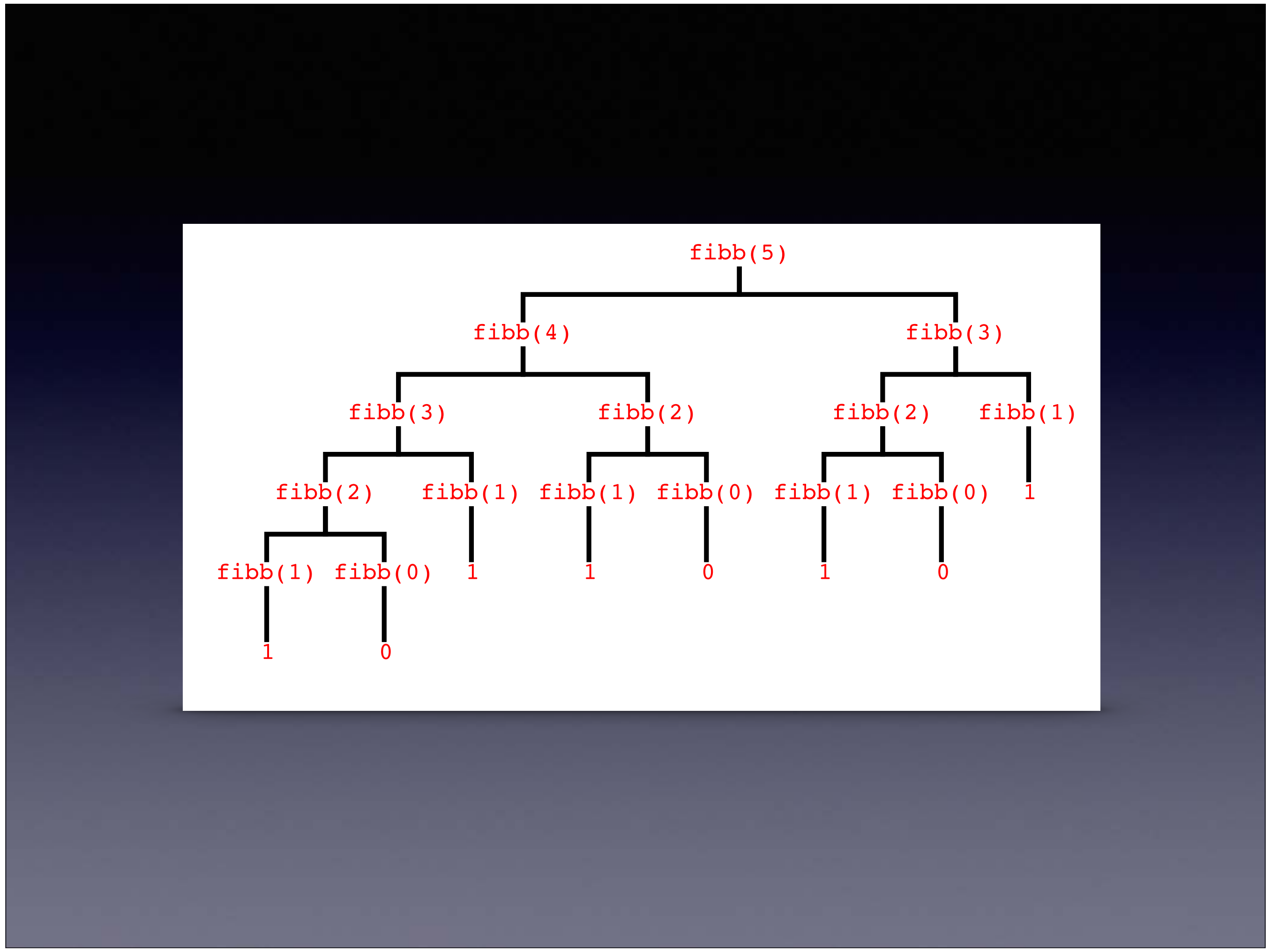
Fibonacci function

- This is a classic example of recursion…
- We can implement this in a computer program either using iteration (i.e. a loop)
- Or recursively…
- But the recursive version is much easier to write, and simpler to understand

Show how to write a recursive fibonacci
The show code for iterative solution

Speed

- In this case, the recursive solution would be slower
- It will end up calculating some values multiple times
- This becomes clear if we draw a graph of how the functions call each other…



See how fibb 3 is called twice
fibb 2 is called 3 times
fibb 1 is called 5 times

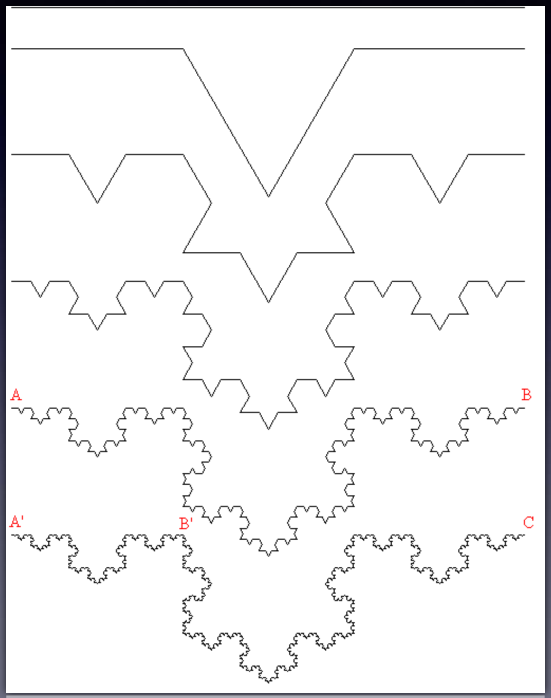
Speed

- Can speed this up by cacheing the answer
- For example, in an array
- If the value has been calculated before, extract answer from the array
- If not, calculate and store for future use
- Only works if the result is invariant (never changes)

Uses of recursion

- Koch Snowflake (and other fractals)
- Divide-and-conquer routines
- Backtracking
- Processing data structures…
- Parsers

Koch Snowflake



Divide and Conquer

- A classic approach used in designing algorithms is ‘divide-and-conquer’
- Problem is too hard to solve in its entirety
- Split into two simpler problems and solve them separately and combine the result
- And so on, until you end up with something solvable

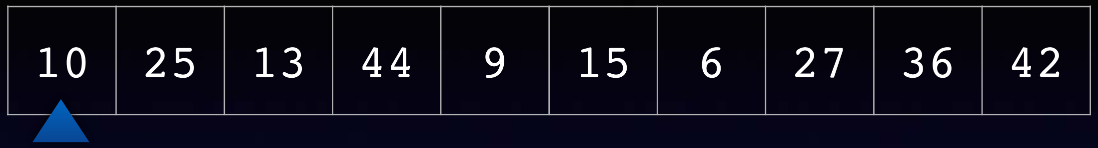
Sorting

- A classic use of divide-and-conquer is the quicksort algorithm for sorting an array
- This is only a brief overview, you will see this in much more detail later in the course
- Takes the array and picks a value from the array
- This becomes the pivot value

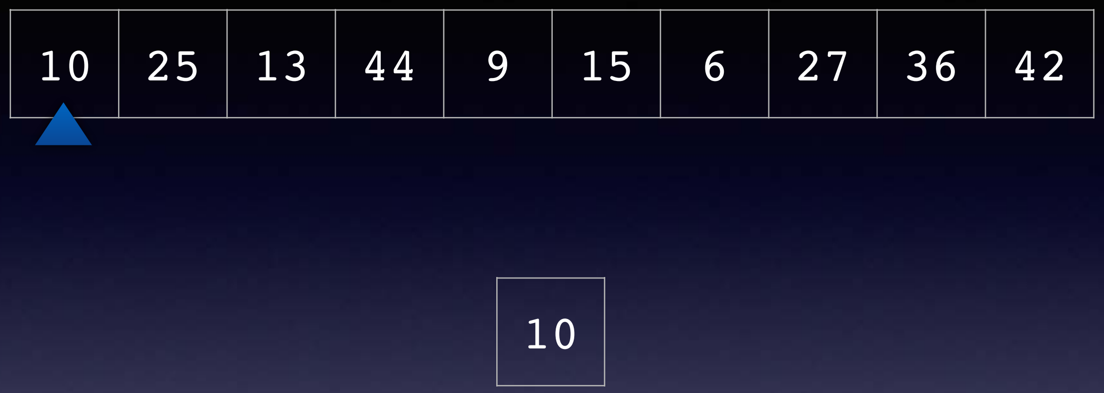
Quicksort shuffle

- Shuffle the values in the array about so that it is in three parts
- All the values less than the pivot (`A`)
- The pivot
- All the values greater than the pivot (`B`)
- Now call quicksort on `A` and `B`

10	25	13	44	9	15	6	27	36	42

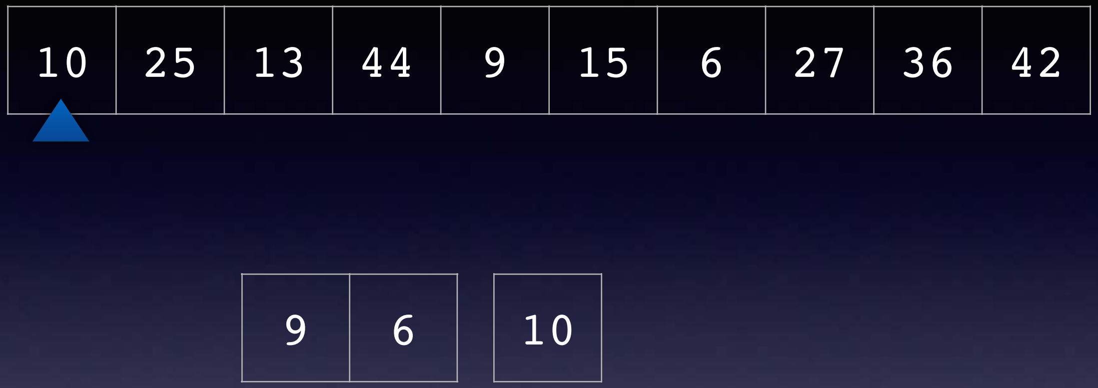


10	25	13	44	9	15	6	27	36	42



10	25	13	44	9	15	6	27	36	42

10



10	25	13	44	9	15	6	27	36	42

9	6

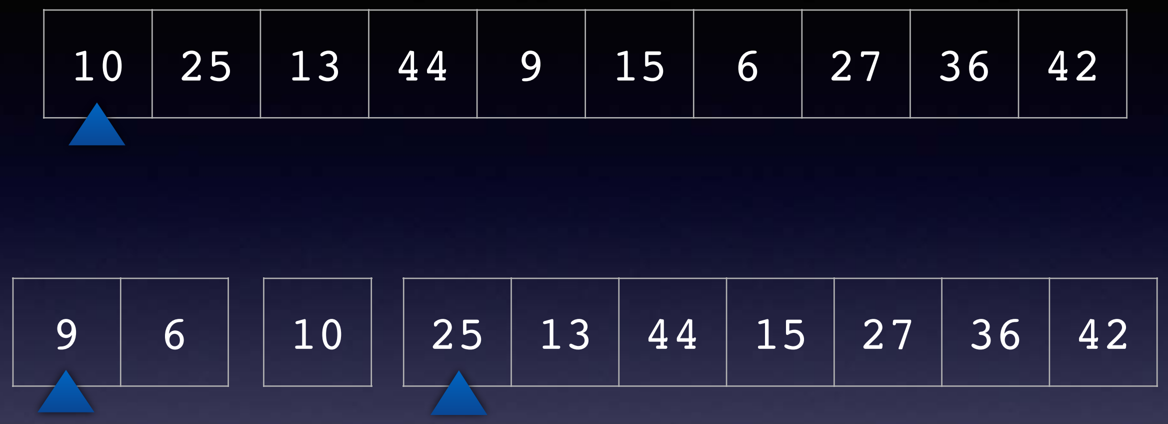
10

10	25	13	44	9	15	6	27	36	42

9	6

10

25	13	44	15	27	36	42

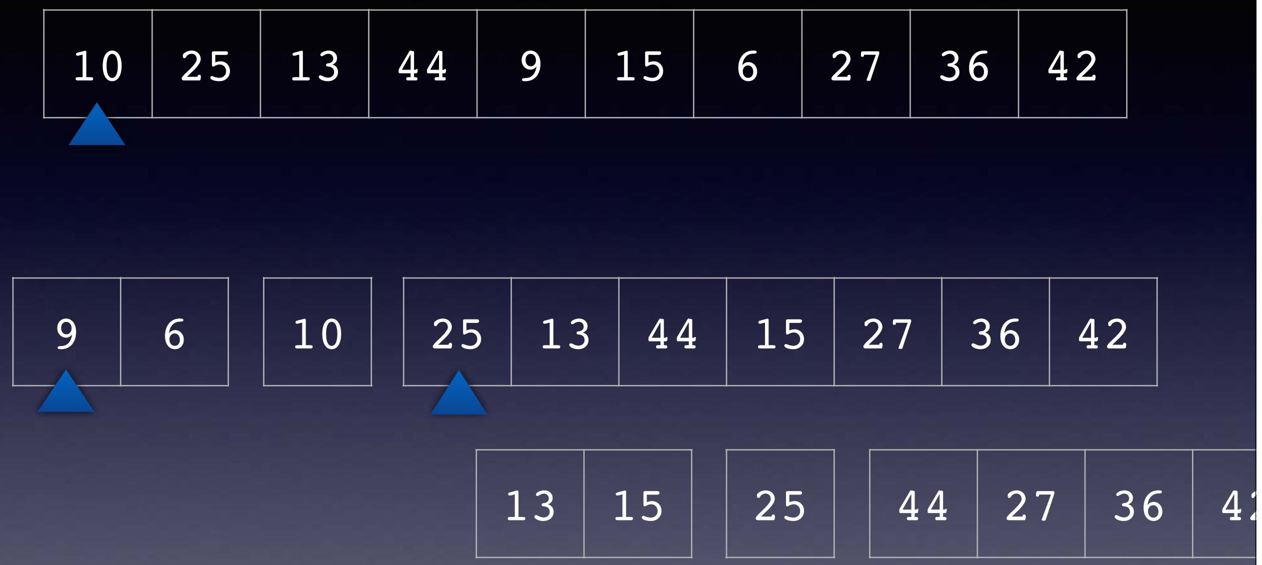


10	25	13	44	9	15	6	27	36	42

9	6

10

25	13	44	15	27	36	42



10 25 13 44 9 15 6 27 36 42

9 6 | 10 | 25 13 44 15 27 36 42

13 15 | 25 | 44 27 36 4

Quicksort base case

- Eventually, quicksort will be called on an array containing zero or only one element
- This is, by definition, sorted
- But, that means the quicksort call above has also sorted its bit of the array out
- Because its `A` and `B` must now be sorted, and the pivot is between the two, all of them are in order

Backtracking

- Because each recursive function has its own local variables
- When you return to a point it still has the same values
- This makes recursion very useful if you ever need to backtrack and try a different example, e.g. solving Sudoku puzzles

Recursion or Iteration

- Some problems model better as recursion
- Others work better iteratively
- Best to chose the implementation that best represents the task

Finally

- Look at a few final bits and pieces we haven't covered yet
- Generally, new ways of doing things we've already seen

Comparisons

- Often you will want to do different things depending on a variable having a specific value
- We have seen how we can implement this
- Using the if…else statement
- Chaining them together to test for a specific value

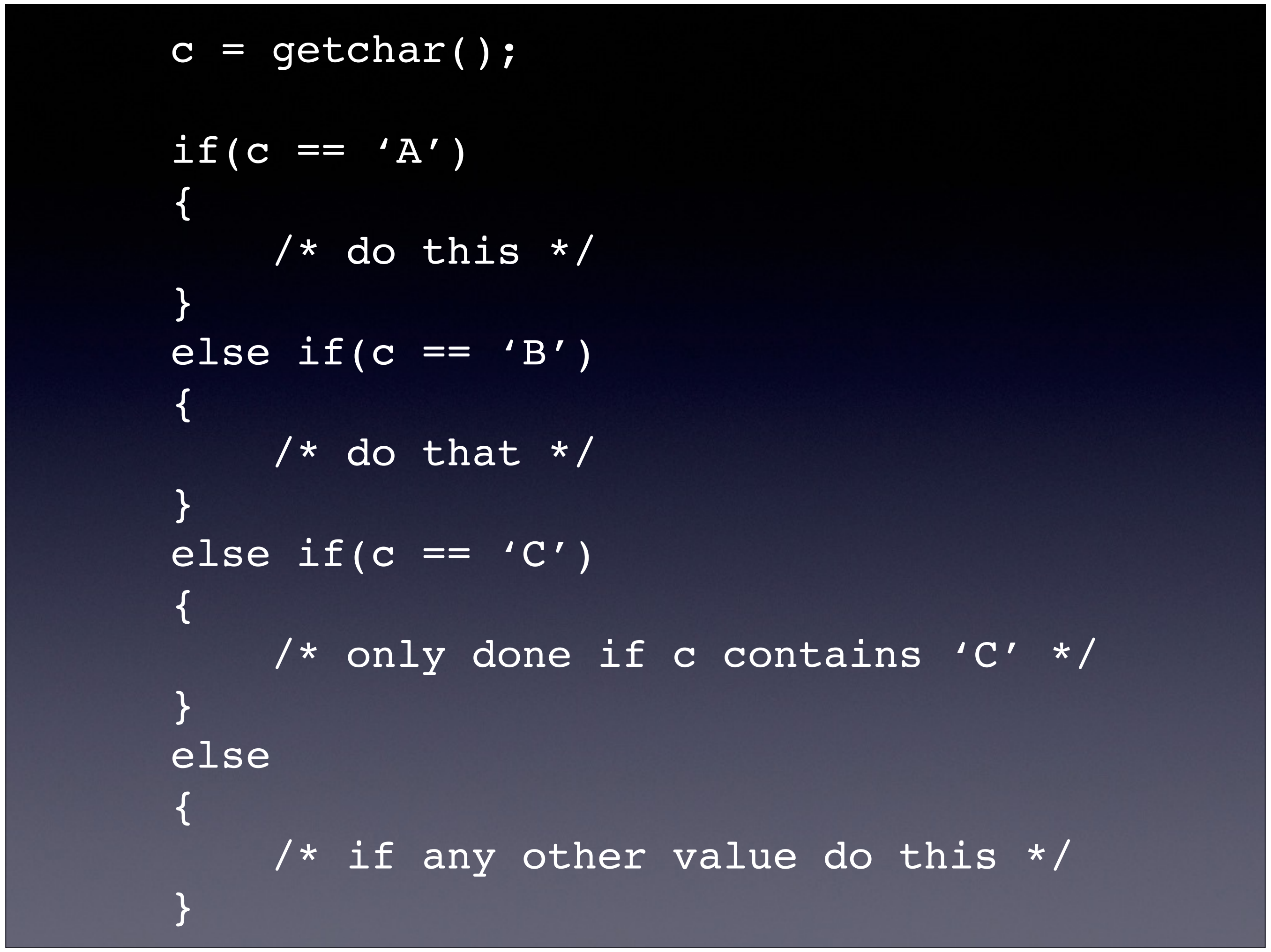
```
c = getchar();

if(c == ‘A’)
{
    /* do this */
}
else if(c == ‘B’)
{
    /* do that */
}
else if(c == ‘C’)
{
    /* only done if c contains ‘C’ */
}
else
{
    /* if any other value do this */
}
```

`switch` statements

- C provides another way to implement this
- Only works with integer values
- Using the `switch` statement
- Each case that can happen is labelled using the case instruction
- Program jumps to the right case based on the value

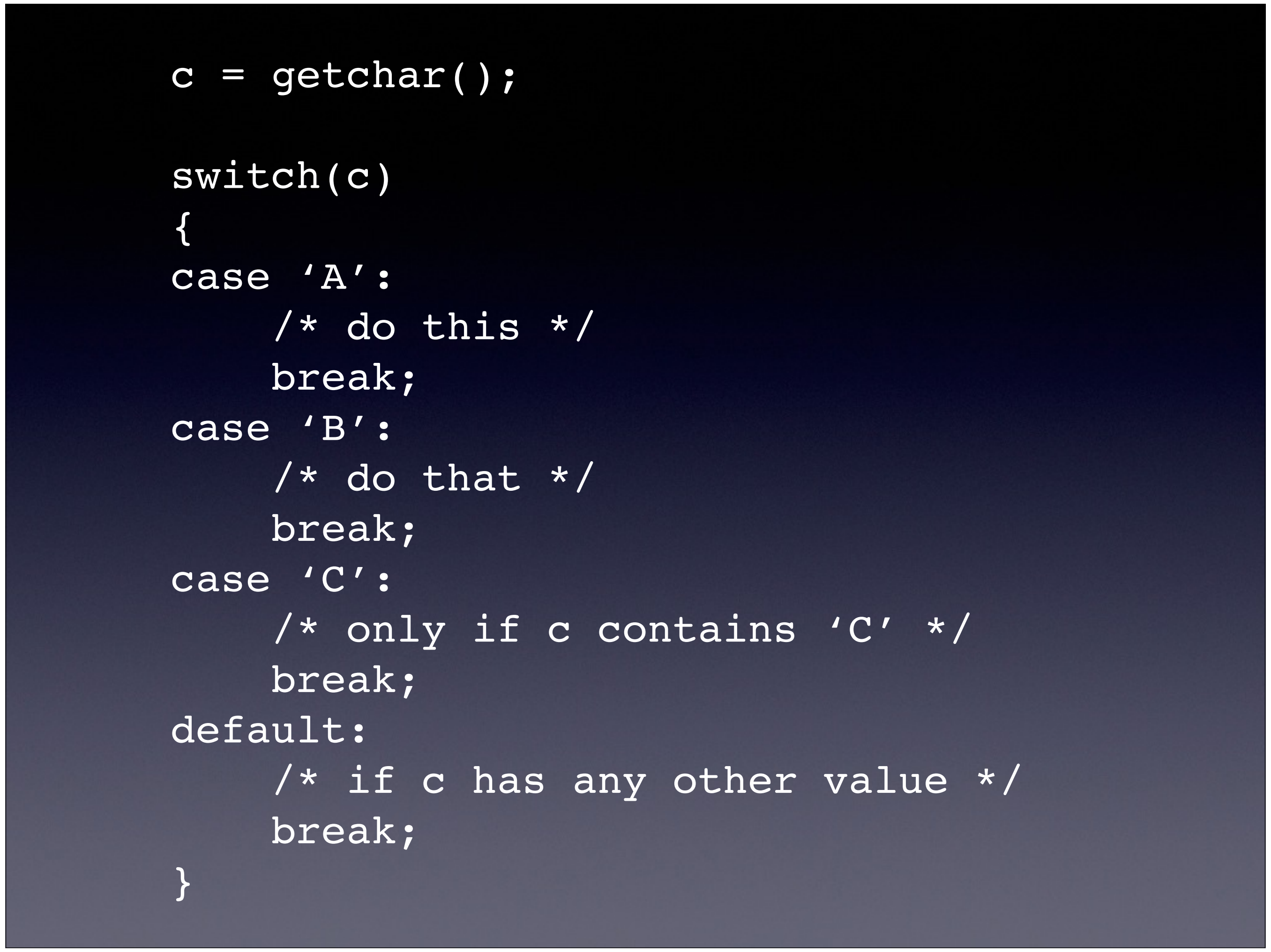
And characters since they are considered integers…



```
c = getchar();

if(c == ‘A’)
{
    /* do this */
}
else if(c == ‘B’)
{
    /* do that */
}
else if(c == ‘C’)
{
    /* only done if c contains ‘C’ */
}
else
{
    /* if any other value do this */
}
```

Can rewrite this as…



```
c = getchar();

switch(c)
{
case ‘A’:
    /* do this */
    break;
case ‘B’:
    /* do that */
    break;
case ‘C’:
    /* only if c contains ‘C’ */
    break;
default:
    /* if c has any other value */
    break;
}
```

this using a switch statement

Which case?

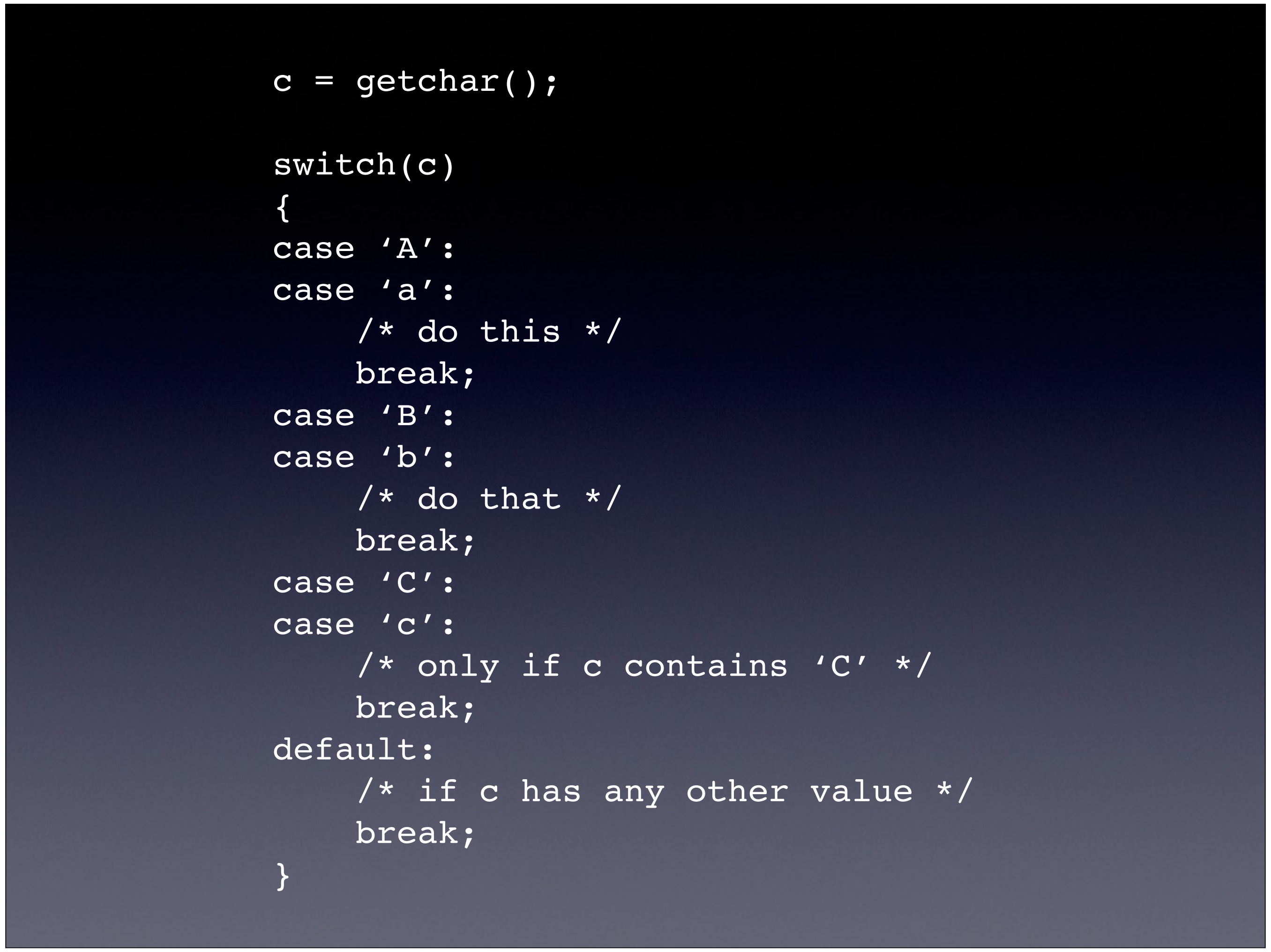
- Define a specific case to jump to using
  `case <value>:`
- `<value>` must be a constant integer value (not a variable), e.g:
  `case 42:`
- Can also use the optional `default:` label to specify a default condition (if needed)

`case` Execution

- Switch jumps to the `case` (or `default`) that matches the value in the variable
- If no match found, carries on after switch
- Each line of code executed *until end of* `switch`
- Not the next `case` statement
- Use `break` to stop it before the next `case`

Falling through

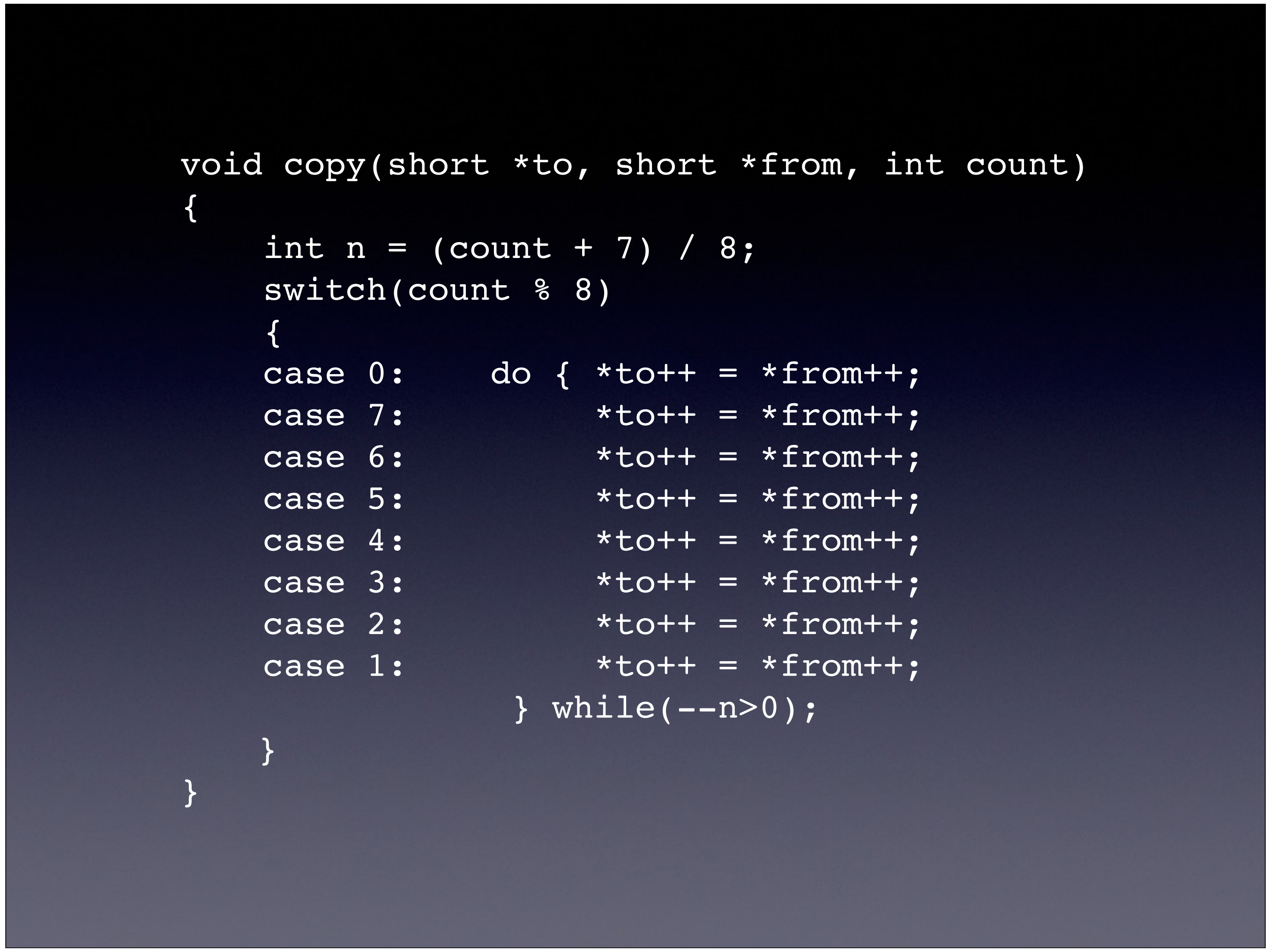
- This is called ‘falling-through’
- Can be a pain if you forget to put the break statement in
- But can also be useful
- For example, if we wanted to make the previous example work for upper and lower case characters



```
c = getchar();

switch(c)
{
case ‘A’:
case ‘a’:
    /* do this */
    break;
case ‘B’:
case ‘b’:
    /* do that */
    break;
case ‘C’:
case ‘c’:
    /* only if c contains ‘C’ */
    break;
default:
    /* if c has any other value */
    break;
}
```

this using a switch statement



```
void copy(short *to, short *from, int count)
{
    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);
    }
}
```

Duff’s device, yes this is valid C and yes it’s crazy. However, it made sense when they wrote it

It’s an optimized routine to copy count bytes from from and output to to

Bit-twiddling

- Sometimes necessary to manipulate individual bits
  e.g. for handling images
- C lets us do this
- Use a combination of the boolean logical operators and/or/xor/not
- And bit-shifting

Bitshifting

- Shown you bit-shifts in ARM assembler
- C provides the same ability to shift bits, using the << and >> operator
- `n << y`, means shift the value `n` `y` places to the left
- `n >> y`, means shift the value `n` `y` places right

Undefined whether it is an arithmetic or logical shift

Logic

- As well as `&&` and `||` used to combine conditionals, C also provides bitwise logical operators
- Applies the logical operator to each and every bit in the variable

Bitwise operators

&	And all the bits together individually
|	Or all the bits together individually
^	Exclusive-or the bits together individually
~	Invert all the bits

Bitwise operations

&

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	0	1	1	1	1	

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	0	1	1	1	1	

0							

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

&

0	0						

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

&

0	0	0					

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	0	1	1	1	1	

0	0	0	0				

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

&

0	0	0	0	1			

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

&

0	0	0	0	1	0		

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	0	1	1	1	1	

0	0	0	0	1	0	1	

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	0	1	1	1	1	

0	0	0	0	1	0	1	0

Bitwise operations

`|`

Bitwise operations

1	0	1	0	1	0	1	0	
0	0	0	0	1	1	1	1	|

Bitwise operations

<table>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td rowspan="2">|</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

|

1	0						

Bitwise operations

1	0	1	0	1	0	1	0	|
0	0	0	0	1	1	1	1	

1	0	1					

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

|

1	0	1	0				

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

\|

1	0	1	0	1			

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

|

1	0	1	0	1	1		

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

|

1	0	1	0	1	1	1	

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

`|`

1	0	1	0	1	1	1	1

Bitwise operations

^

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	

1							

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	
1	0							

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

^

1	0	1					

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	0	1	1	1	1

^

1	0	1	0				

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	

1	0	1	0	0			

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	

1	0	1	0	0	1		

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	

1	0	1	0	0	1	0	

Bitwise operations

1	0	1	0	1	0	1	0	^
0	0	0	0	1	1	1	1	
1	0	1	0	0	1	0	1	

Testing a bit

- Can use and (`&`) to see if a bit (or bits) are set
- And the value to be tested with the number for just that bit
- Result is non-zero if its set, and 0 if not, i.e. true, or false
- Use bitshifts to find right value (`1 << 7`)

Testing a bit

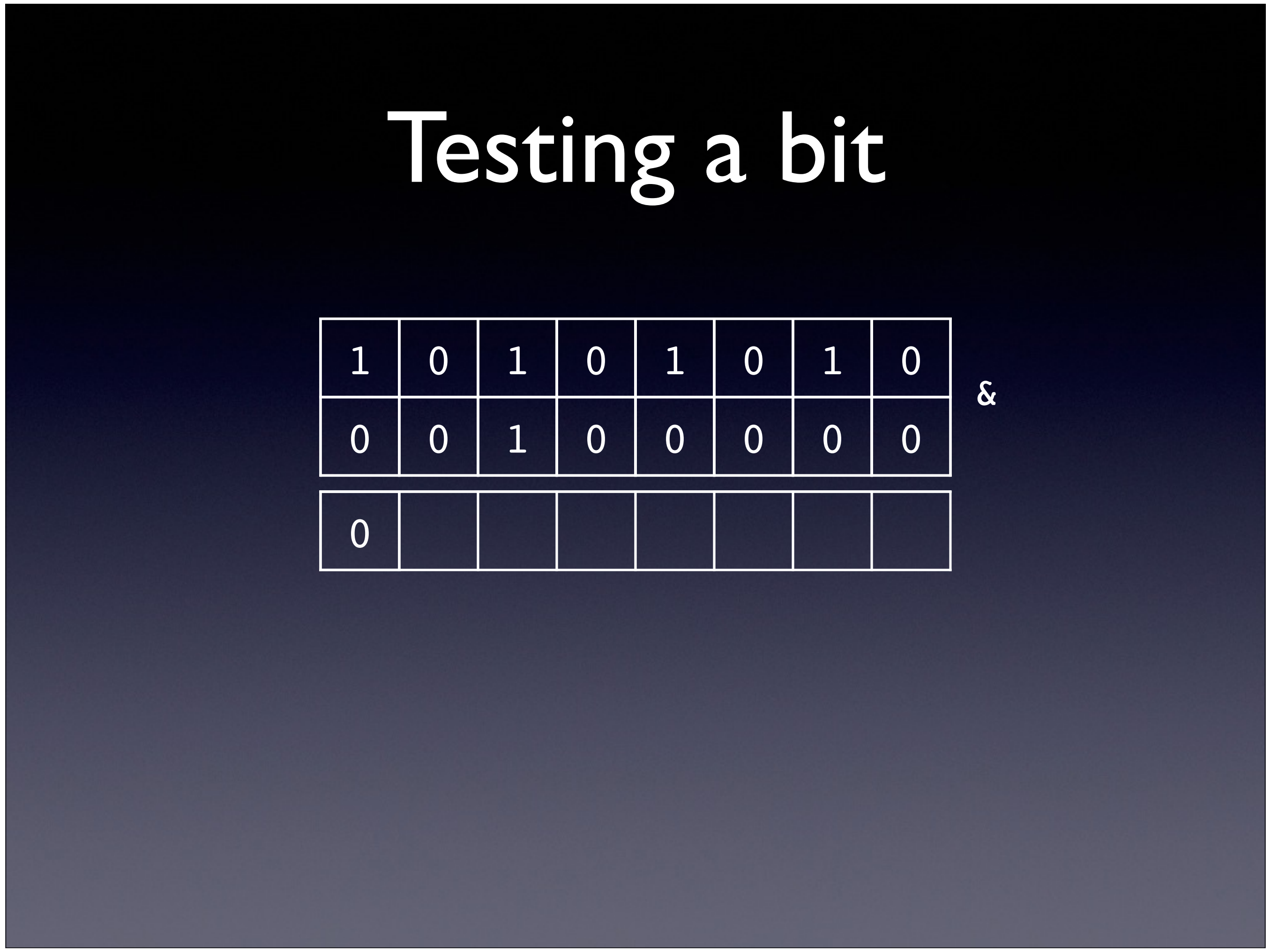
&

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0	&
0	0	1	0	0	0	0	0	

Result is true, bit is set



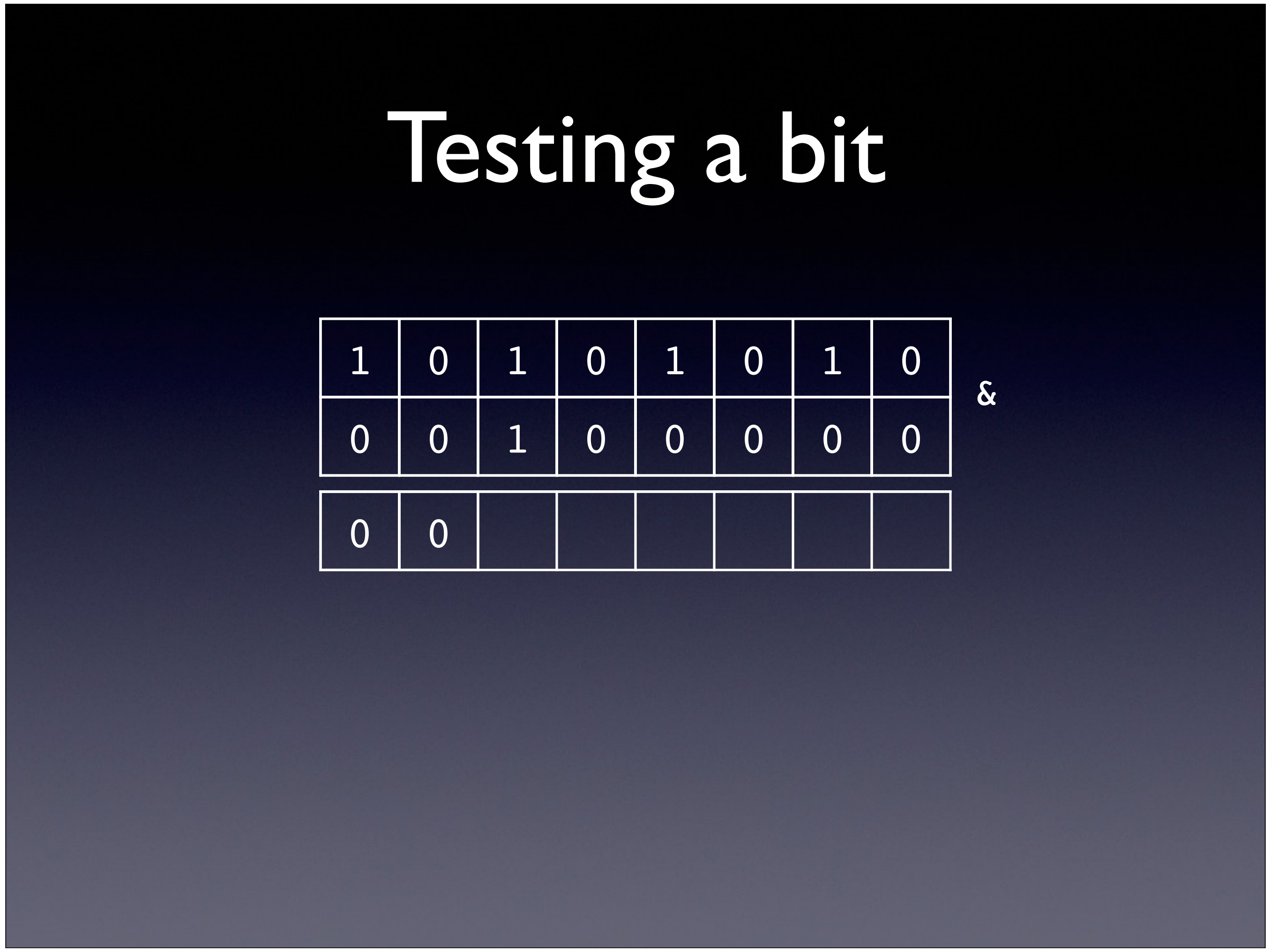
Testing a bit

1	0	1	0	1	0	1	0
0	0	1	0	0	0	0	0

&

0							

Result is true, bit is set



Testing a bit

1	0	1	0	1	0	1	0	&
0	0	1	0	0	0	0	0	
0	0							

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0
0	0	1	0	0	0	0	0

&

0	0	1					

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0	&
0	0	1	0	0	0	0	0	

0	0	1	0				

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0
0	0	1	0	0	0	0	0

&

0	0	1	0	0			

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0		

&

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0
0	0	1	0	0	0	0	0

&

0	0	1	0	0	0	0	

Result is true, bit is set

Testing a bit

1	0	1	0	1	0	1	0
0	0	1	0	0	0	0	0

&

0	0	1	0	0	0	0	0

Result is true, bit is set

Bitwise operations

&

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	
0								

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	
0	0							

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	
0	0	0						

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	

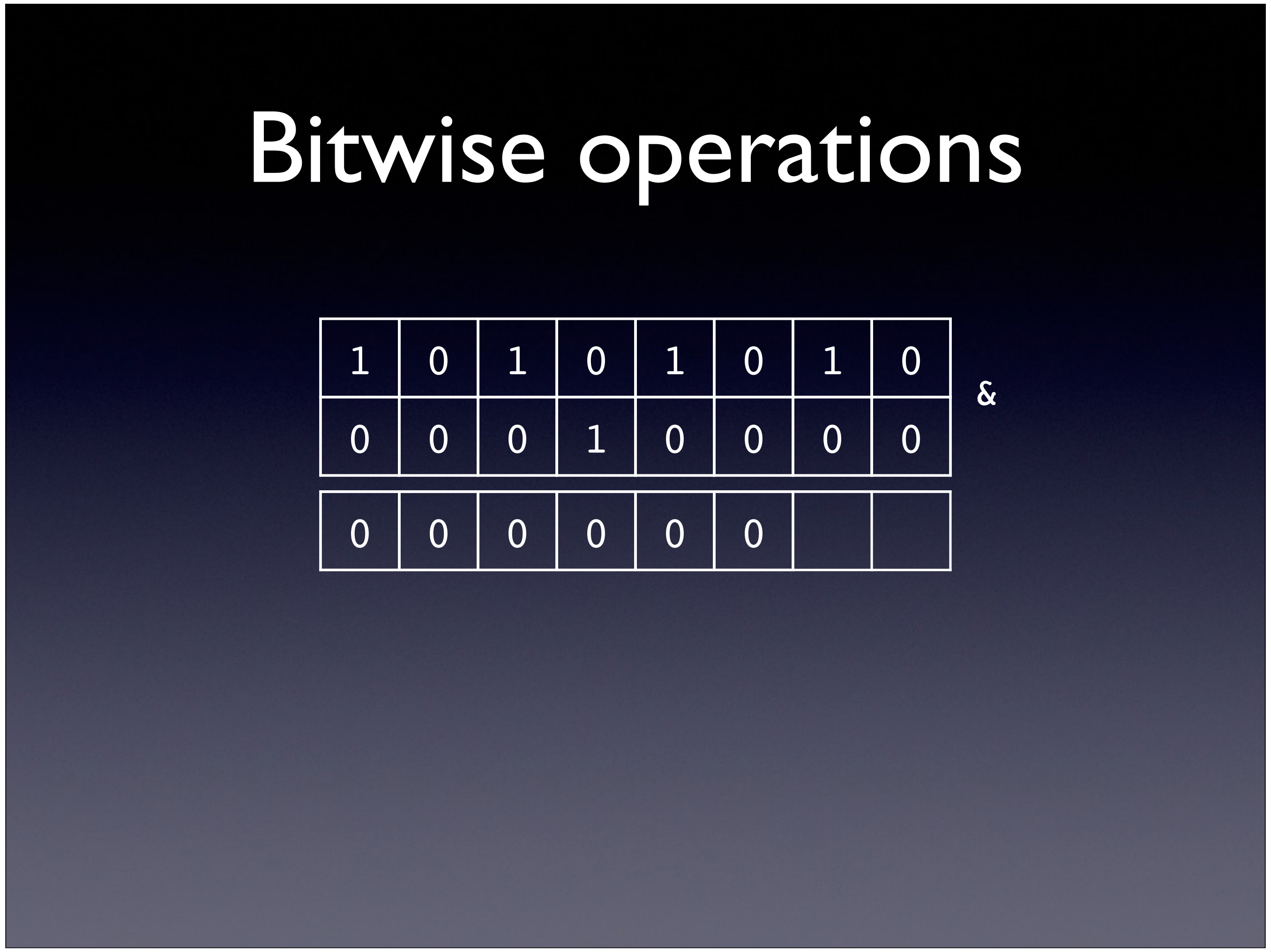
0	0	0	0				

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	
0	0	0	0	0				

Zero so false



Bitwise operations

1	0	1	0	1	0	1	0	&
0	0	0	1	0	0	0	0	
0	0	0	0	0	0			

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	1	0	0	0	0

&

0	0	0	0	0	0	0	

Zero so false

Bitwise operations

1	0	1	0	1	0	1	0
0	0	0	1	0	0	0	0

&

0	0	0	0	0	0	0	0

Zero so false

Conditional operator

- `x = (y == 1) ? 0 : 1;`
- Believe it or not, the above is valid C
- It’s a compact way of writing a conditional
- If the condition `(y==1)` is true the `?` operator selects the first value before `:`
- If false, it selects the second value after `:`

That’s all folks…

- Will use PRG lecture slots to cover CSA
- And the adventure continues
  - G51ISO — Programming in Java
  - G51FUN — Functional Programming

That's all almost folks…

- Will use PRG lecture slots to cover CSA
- And the adventure continues
    - G51ISO — Programming in Java
    - G51FUN — Functional Programming

That's all almost folks…

- Syntax is simple, but understanding how to fit it together is hard…
- It's like learning to play a musical instrument, it takes practice…

Keep Programming