25T3 COMP1521 Week 1

TUTORIAL 1

REVISION/RECURSION

Jimmy Kirkpatrick

INTRODUCTION/ADMIN

My name is Jimmy Kirkpatrick. I'm a 3rd year Computer Science / Statistics student. I'm one of the course admins for COMP1521 and your tutor for this term. I like League of Legends.

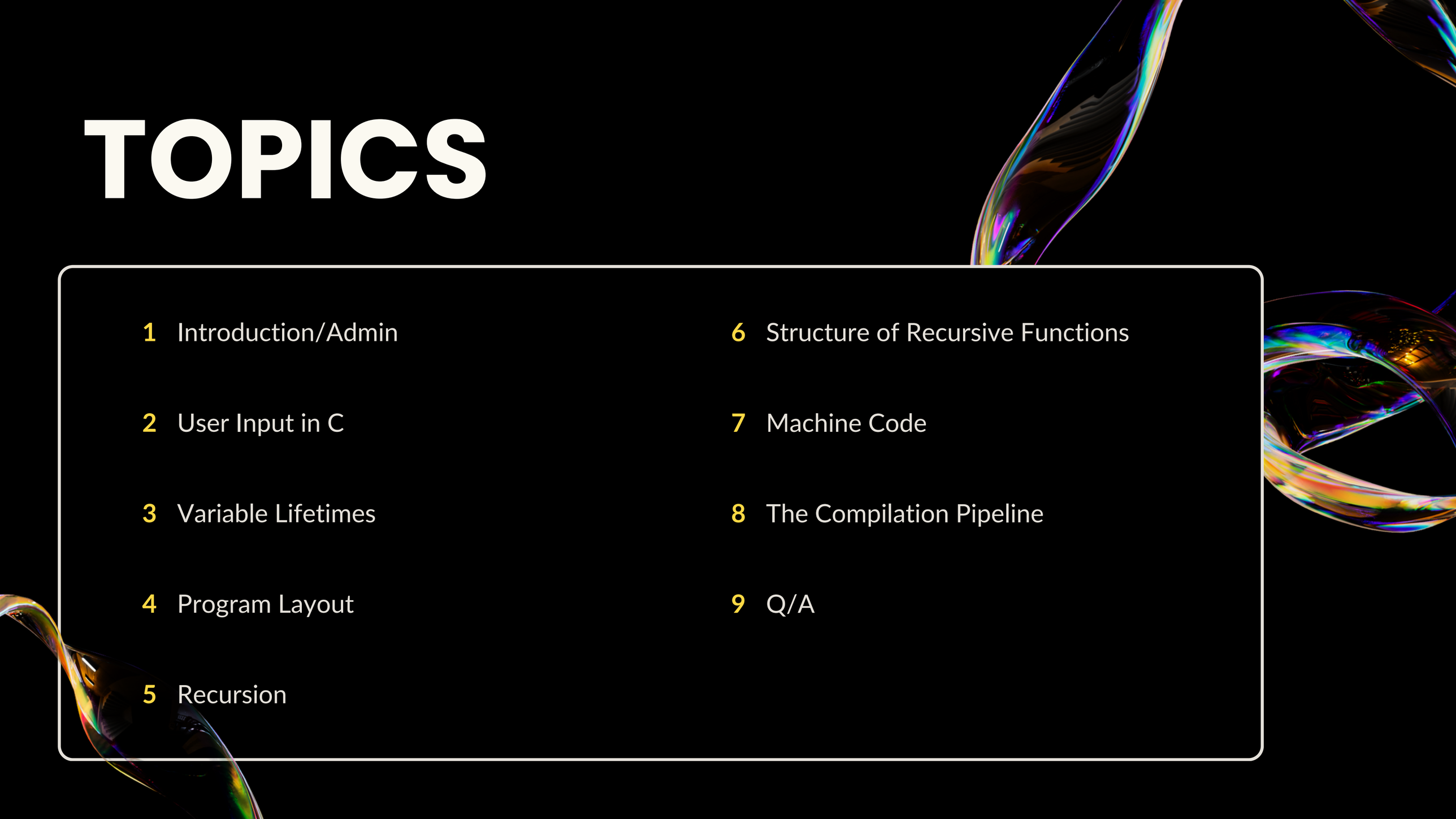
If you have any questions:

- Related to course content: Ask on the forum
- Related to enrolment/personal problems: Email the course account cs1521@cse.unsw.edu.au
- Related to my tutorial/labs/exercises: Email me at james.kirkpatrick@student.unsw.edu.au

Course homepage:
https://cgi.cse.unsw.edu.au/~cs1521/25T3/

TOPICS

1. Introduction/Admin
2. User Input in C
3. Variable Lifetimes
4. Program Layout
5. Recursion
6. Structure of Recursive Functions
7. Machine Code
8. The Compilation Pipeline
9. Q/A



COURSE OUTLINE / ASSESSMENT

- 9 weeks worth of labs with challenge exercises
- 8 weeks worth of weekly tests (starting week 3)
- 2 assignments
- 1 final exam with 40% hurdle

You must complete 2.5 weeks of challenge exercises to gain full marks for the lab component. Without challenge exercises, you can only obtain 95% of the lab marks.

Weekly tests are timed lab exercises. You will be given 1 hour to complete 3 (easy-ish) exercises. Your best 6 weekly tests will count towards the marked component.

The weekly tests are worth 10% together. The labs are worth 15% together. Both assignments are worth 15%, and the final exam is worth 45%.

https://cgi.cse.unsw.edu.au/~cs1521/25T3/outline/

C REVISION

C REVISION

Before we proceed:
Is everyone familiar with for loops?
Strings? Addresses/Memory? Functions?
Command line arguments? Structs?

USER INPUT

- 3 main different ways to constantly obtain input in C

```c
#include <stdio.h>

int main(void) {

    int ch;
    int count = 0;

    while ((ch = fgetc(stdin)) != EOF) {
        count += 1;
    }

    printf("Characters entered = %d\n", count);

}
```

fgetc()

- Reads a single character from a stream
- Returns the read character or EOF

USER INPUT

- 3 main different ways to constantly obtain input in C

```c
#include <stdio.h>

int main(void) {

    char ch;
    int count = 0;

    while (scanf("%c", &ch) != EOF) {
        count += 1;
    }

    printf("Characters entered = %d\n", count);

}
```

fgetc()

- Reads a single character from a stream
- Returns the read character or EOF

scanf()

- Scans input according to a format string
- Saves the read character into a buffer
- Returns amount of read items or EOF

USER INPUT

- 3 main different ways to constantly obtain input in C

```c
#include <stdio.h>

int main(void) {

    char arr[2];
    int count = 0;

    while (fgets(arr, 2, stdin) != NULL) {
        count += 1;
    }

    printf("Characters entered = %d\n", count);

}
```

fgetc()

- Reads a single character from a stream
- Returns the read character or EOF

scanf()

- Scans input according to a format string
- Saves the read character into a buffer
- Returns amount of read items or EOF

fgets()

- Reads multiple characters into an buffer array with a terminating null byte ‘\0’
- Returns a pointer to the array or NULL

VARIABLE LIFETIMES

- 2 types of variables
  - Local variables
    - Exist only in the current scope
    - e.g. Functions, for loops
  - Global variables
    - Exist for the entire program
    - malloc / free do NOT create variables, they allocate memory which is then referenced by a local/global variable

VARIABLE LIFETIMES

```c
#include <stdio.h>

int global = 1;

void my_function(int local) {
    local = 3;
    global = 3;
    return;
}

int main(void) {
    int local = 1;

    local = 2;
    global = 2;

    my_function(local);

    printf("local: %d\nglobal: %d\n", local, global);

    return 0;
}
```

- What is the expected output?

VARIABLE LIFETIMES

```c
#include <stdio.h>

int global = 1;

void my_function(int local) {
    local = 3;
    global = 3;
    return;
}

int main(void) {
    int local = 1;

    local = 2;
    global = 2;

    my_function(local);

    printf("local: %d\nglobal: %d\n", local, global);

    return 0;
}
```

- What is the expected output?

  local: 2
  global: 3

VARIABLE LIFETIMES

```c
#include <stdio.h>

int global = 1;

void my_function(int local) {
    local = 3;
    global = 3;
    return;
}

int main(void) {
    int local = 1;

    local = 2;
    global = 2;

    my_function(local);

    printf("local: %d\nglobal: %d\n", local, global);

    return 0;
}
```

- What is the expected output?

  local: 2
  global: 3

- What's the difference?

VARIABLE LIFETIMES

```c
#include <stdio.h>

int global = 1;

void my_function(int local) {
    local = 3;
    global = 3;
    return;
}

int main(void) {
    int local = 1;

    local = 2;
    global = 2;

    my_function(local);

    printf("local: %d\nglobal: %d\n", local, global);

    return 0;
}
```

- What is the expected output?

  local: 2
  global: 3

- What's the difference?

- How do we fix the program?

VARIABLE LIFETIMES

```c
#include <stdio.h>

int *get_var(void) {
    int local = 1;
    return &local;
}

int main(void) {
    int *local_pointer = get_var();
    printf("%d\n", *local_pointer);

    return 0;
}
```

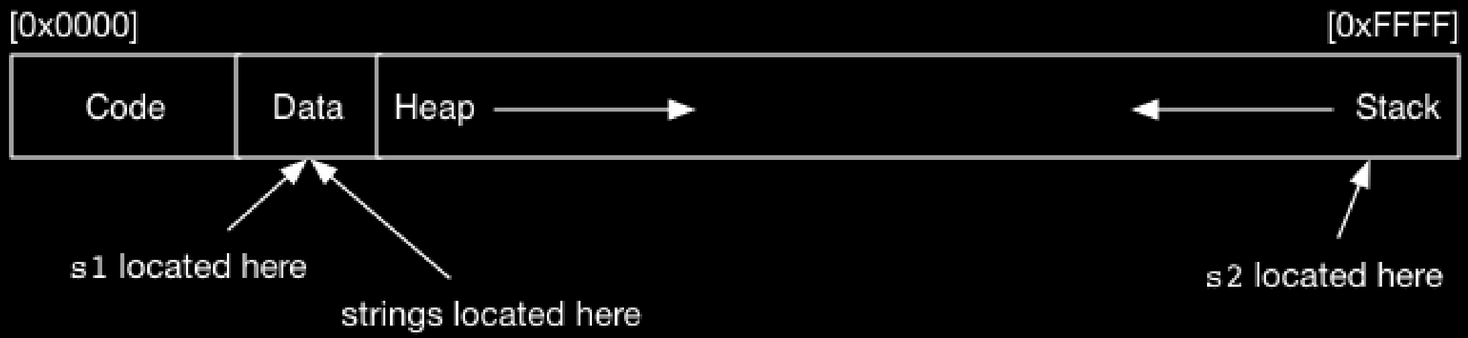
- What's wrong with this program?

VARIABLE LIFETIMES

```c
#include <stdio.h>

int *get_var(void) {
    int local = 1;
    return &local;
}

int main(void) {
    int *local_pointer = get_var();
    printf("%d\n", *local_pointer);

    return 0;
}
```

- What's wrong with this program?
- How do we fix the program?

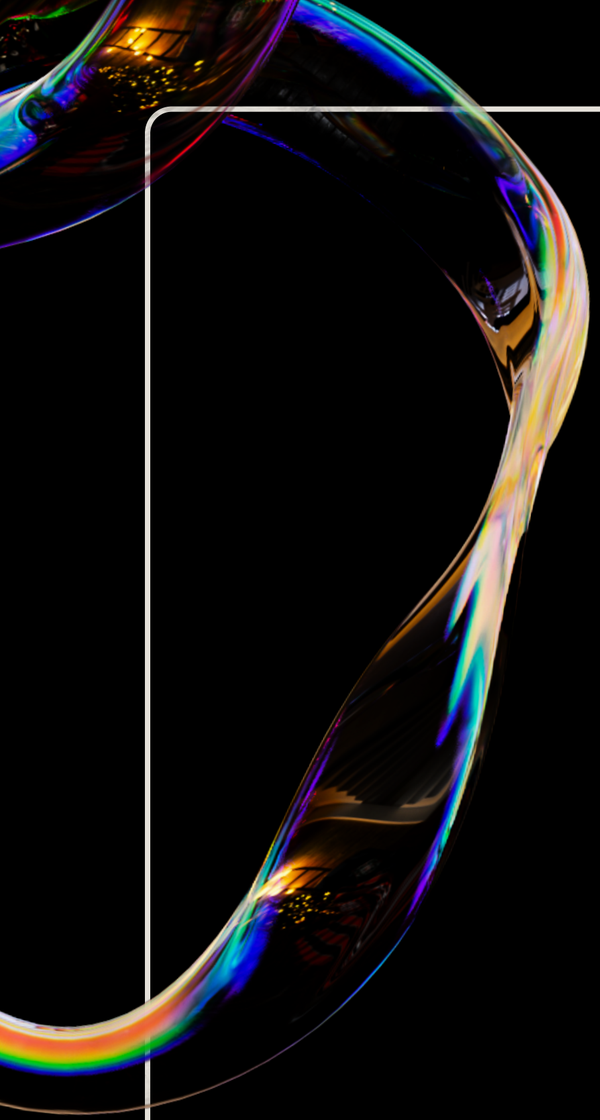
PROGRAM LAYOUT



- ‘local’ exists in the stack
- ‘global’ exists in the data segment

More on the stack in Week 4, where we manipulate it directly in assembly

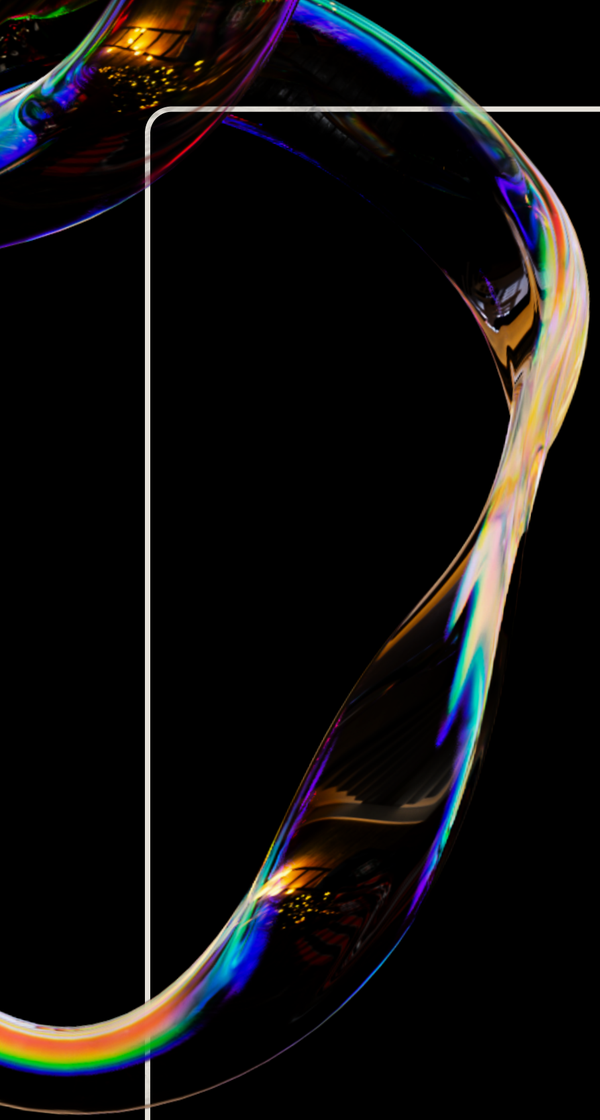

RECURSION



RECURSION

- Definition

A recursive function is a function that calls itself

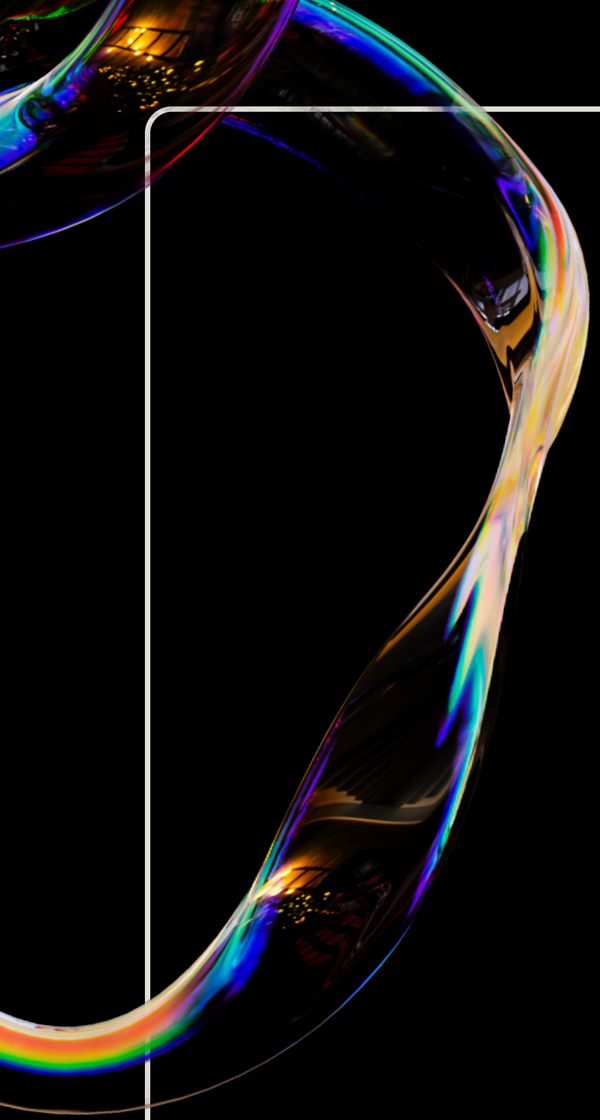


RECURSION

- Definition

A recursive function is a function that calls itself

```
void foo(void) {
    return foo();
}
```



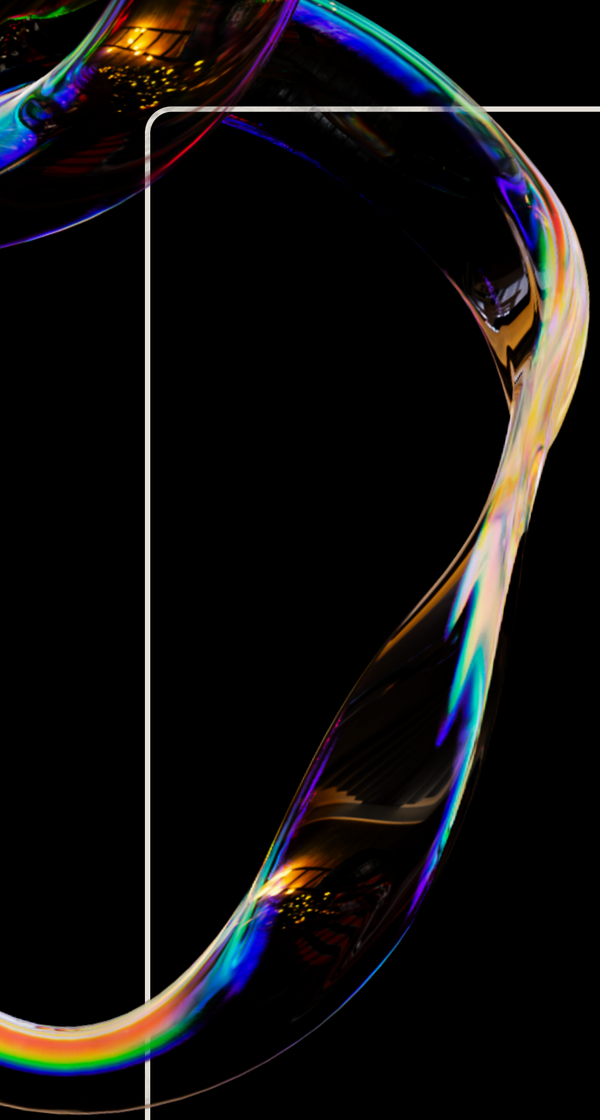
RECURSION

- Definition

A recursive function is a function that calls itself

```
void foo(void) {
    return foo();
}
```

- What happens if we try to run this?



RECURSION

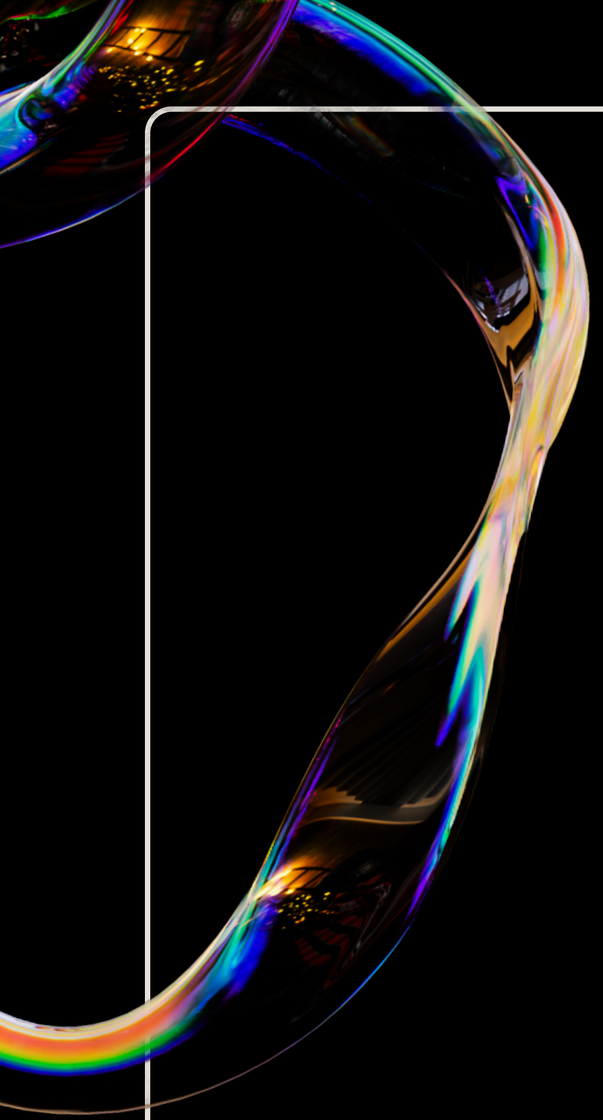
- Definition

A recursive function is a function that calls itself

```
void foo(void) {
    return foo();
}
```

- What happens if we try to run this?

  Stack overflow.



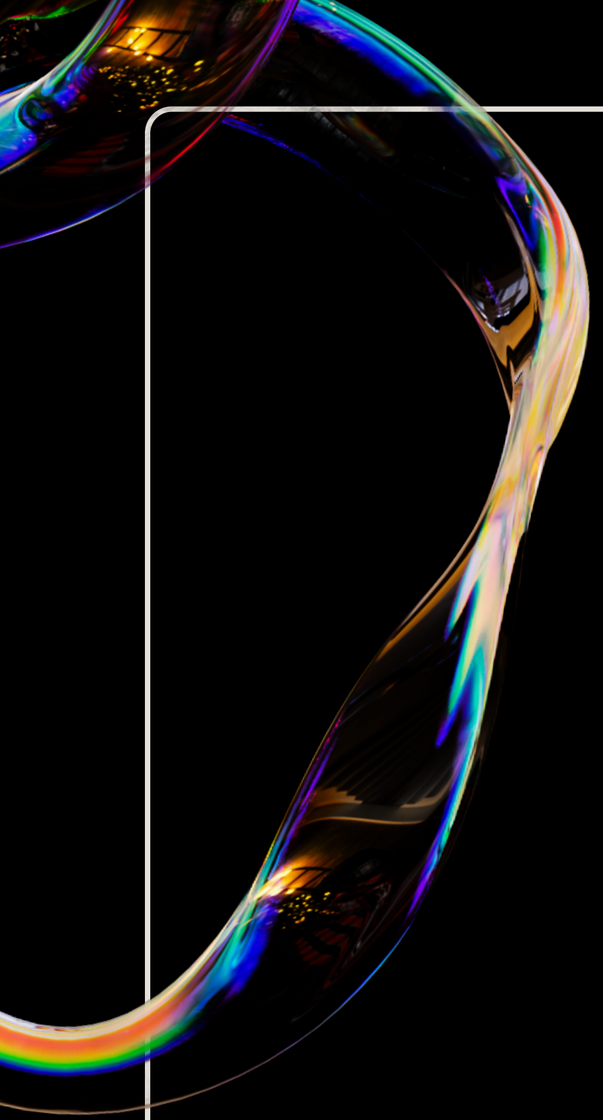
RECURSION

- Definition

A recursive function is a function that calls itself

```
int factorial(int n) {
    return n * factorial(n - 1);
}
```

- Let’s look at an example of a practical recursive function



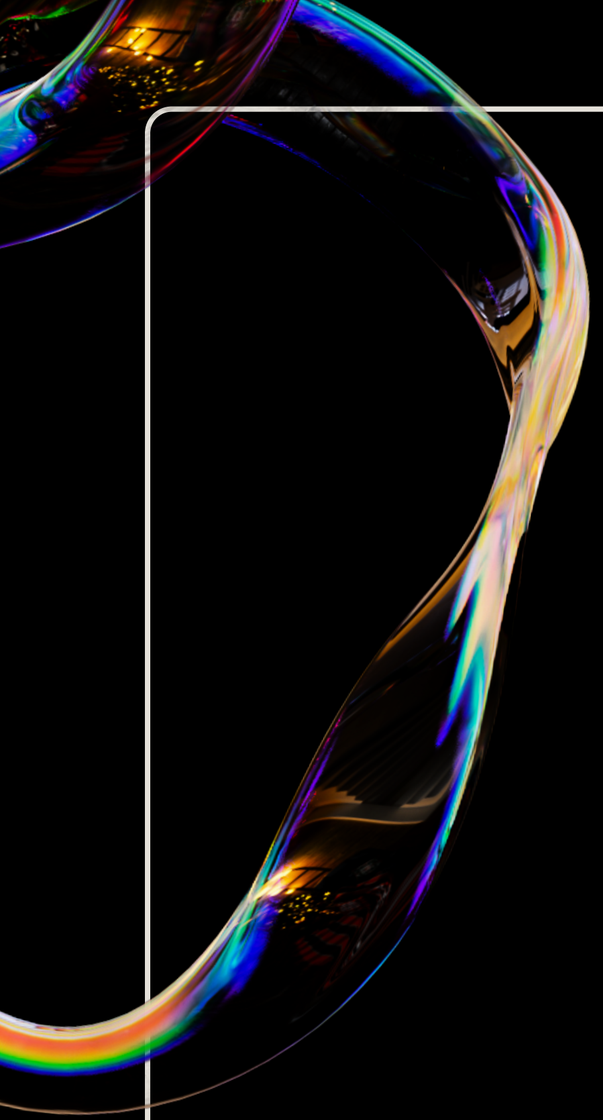
RECURSION

- Definition

A recursive function is a function that calls itself

```
int factorial(int n) {
    return n * factorial(n - 1);
}
```

- Let’s look at an example of a practical recursive function
- Does this work?



RECURSION

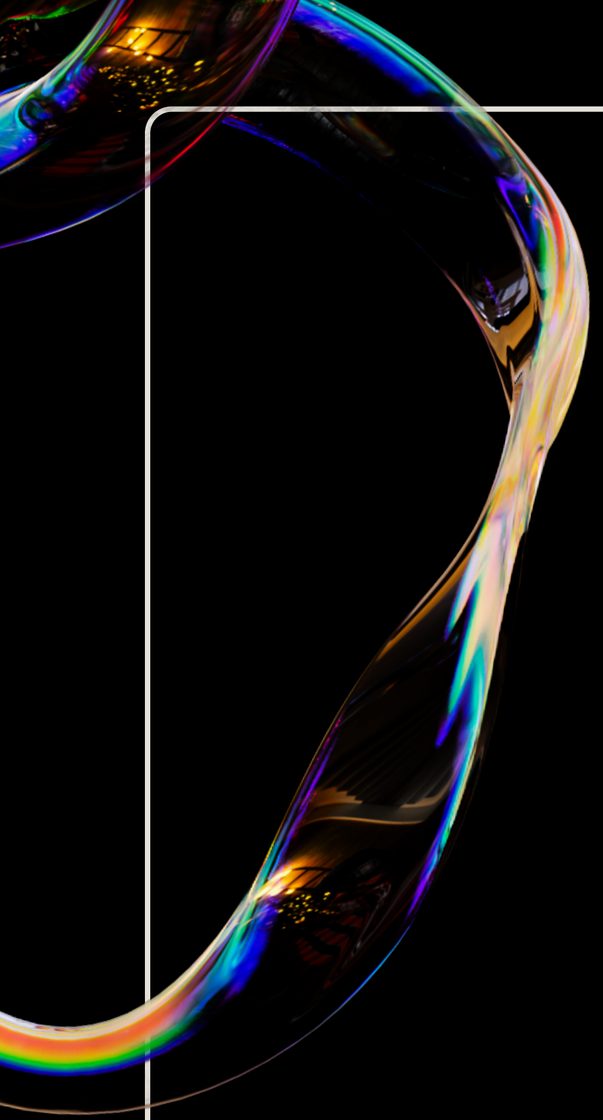
- Definition

A recursive function is a function that calls itself

```
int factorial(int n) {
    return n * factorial(n - 1);
}
```

- Let’s look at an example of a practical recursive function
- Does this work?

  No, it recurses forever.
- How do we fix it?



RECURSION

- Definition

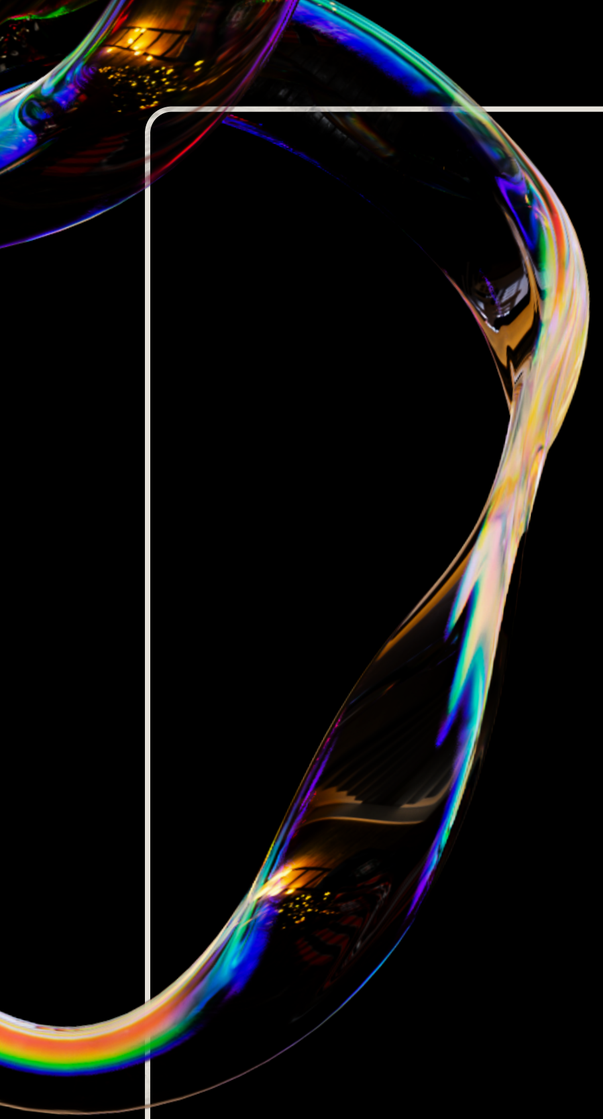
A recursive function is a function that calls itself

```
// note: undefined for negative numbers
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

- Let’s look at an example of a practical recursive function
- Does this work?

  No, it recurses forever.
- How do we fix it?

  Base cases.

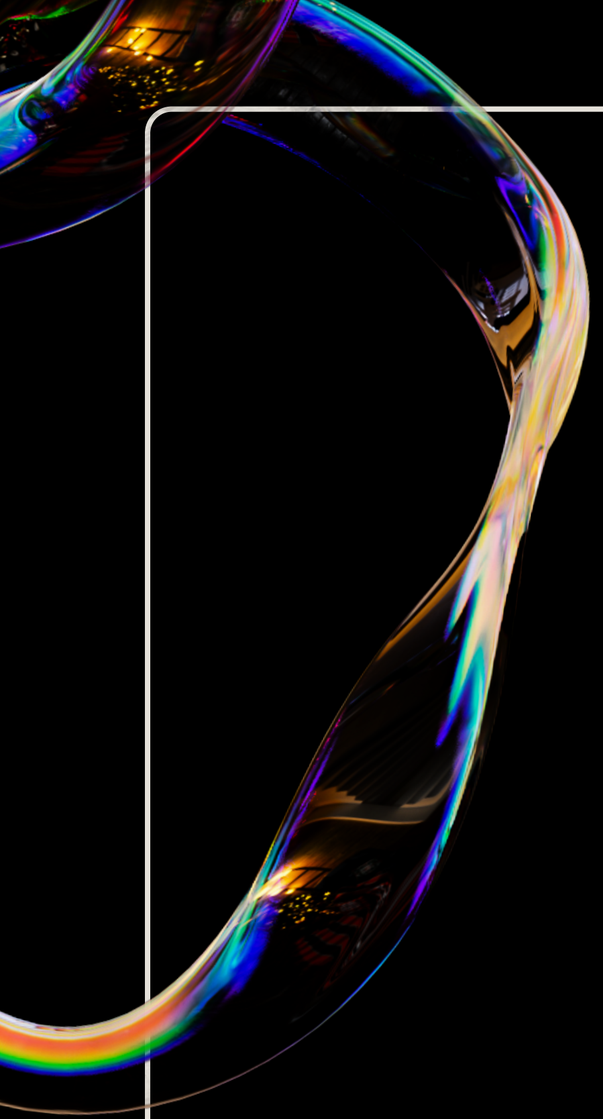


STRUCTURE OF A RECURSIVE FUNCTION

A recursive function must have:

- Base cases
  - As in, to stop recursing somewhere
- Recursive step
  - A function call to itself

Otherwise, doesn’t work / not a recursive function.

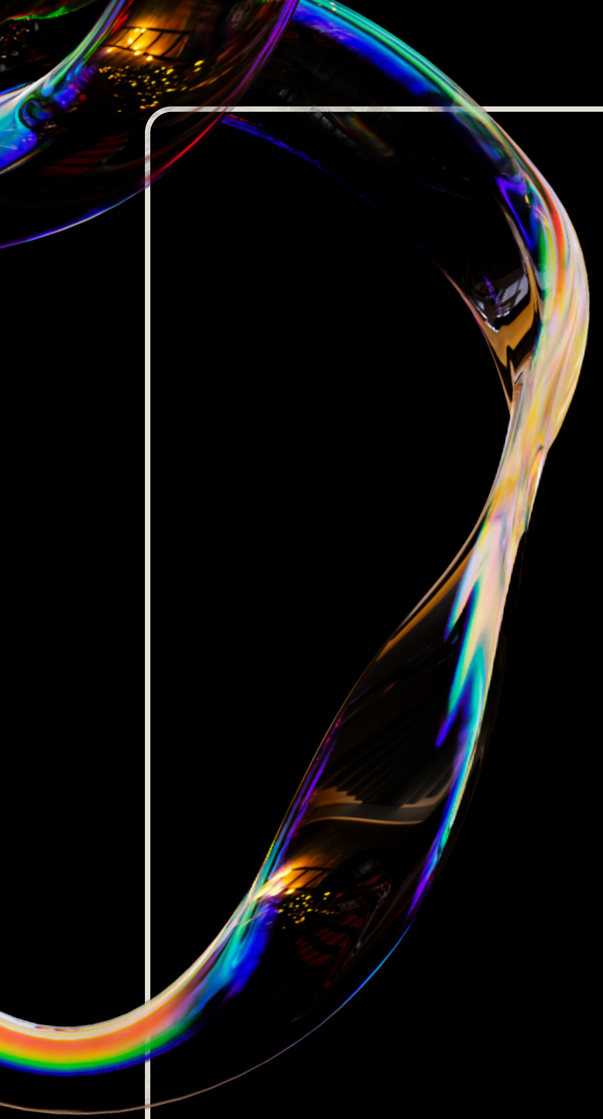


STRUCTURE OF A RECURSIVE FUNCTION

A recursive function must have:

- Base cases
  - As in, to stop recursing somewhere
- Recursive step
  - A function call to itself
  - Always appears after the base cases

Otherwise, doesn't work / not a recursive function.



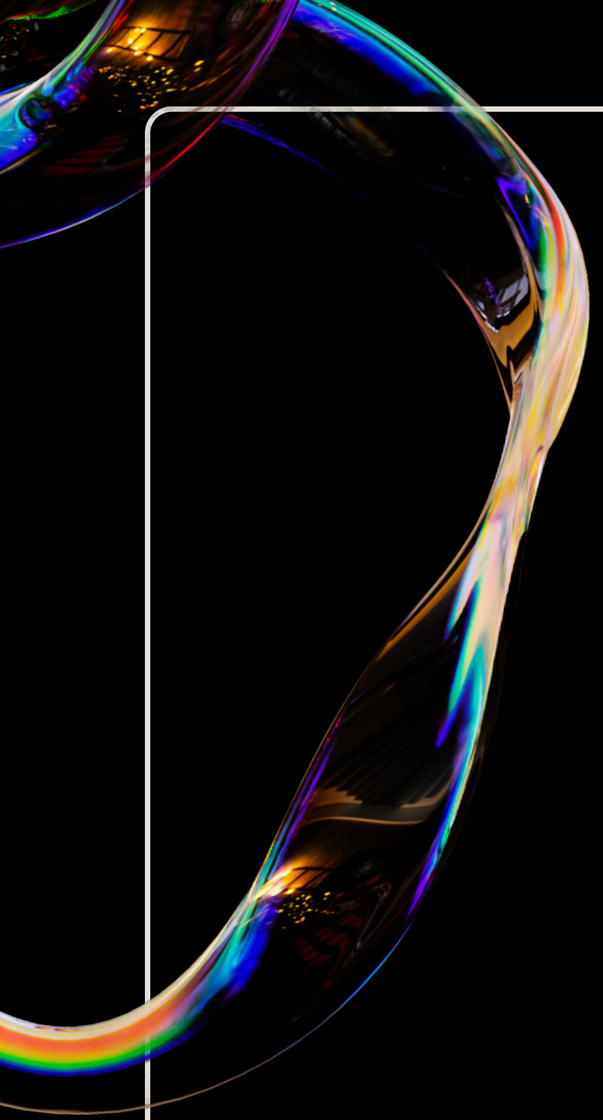
STRUCTURE OF A RECURSIVE FUNCTION

A recursive function must have:

- Base cases
  - As in, to stop recursing somewhere
- Recursive step
  - A function call to itself
  - Always appears after the base cases

Otherwise, doesn't work / not a recursive function.

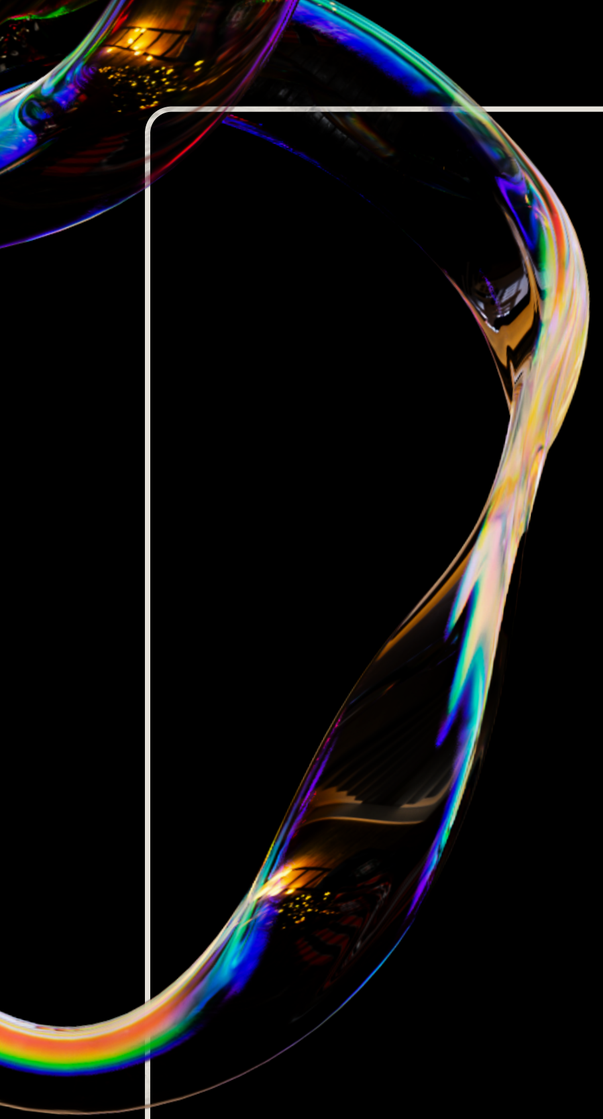
Useful: Recursive functions can do anything a loop can do.



STRUCTURE OF A RECURSIVE FUNCTION

Recursive functions can do anything a loop can do.

```c
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

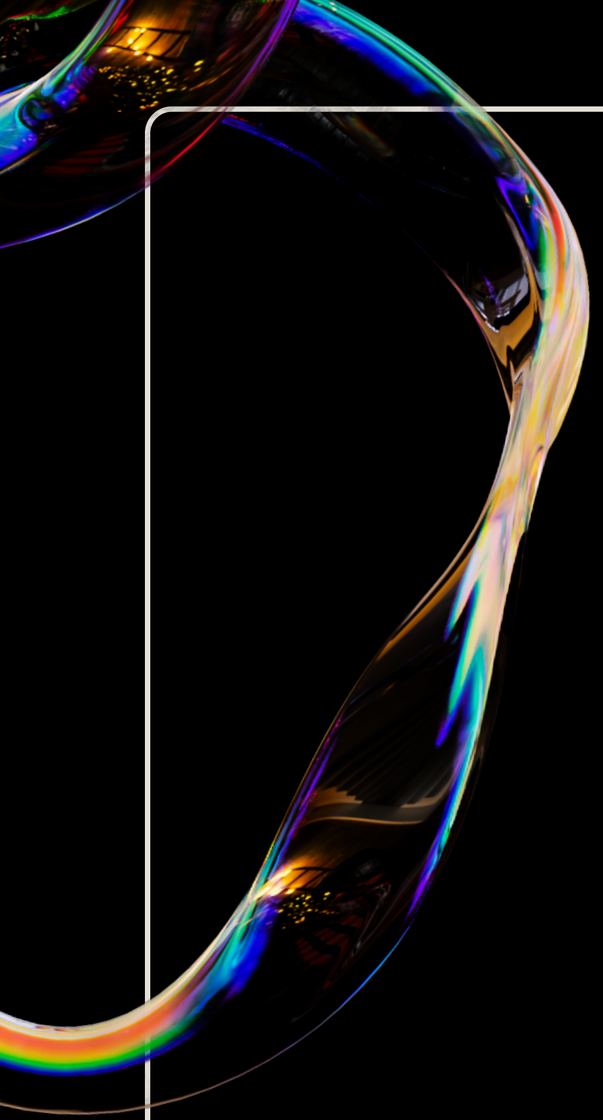


STRUCTURE OF A RECURSIVE FUNCTION

Recursive functions can do anything a loop can do.

```c
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

- Convert the loop into a recursive function



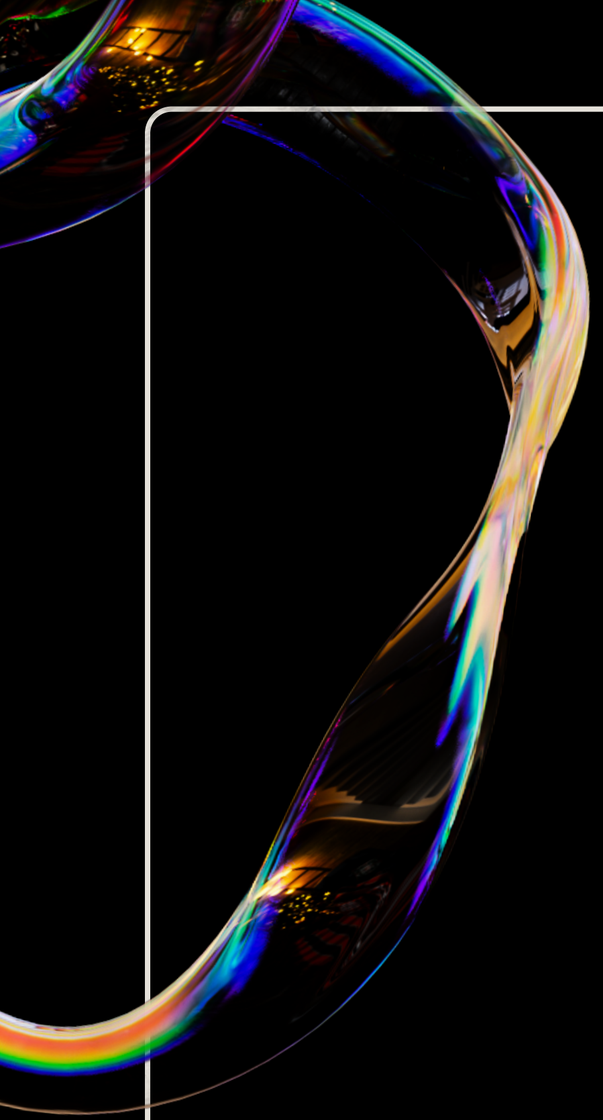
STRUCTURE OF A RECURSIVE FUNCTION

Recursive functions can do anything a loop can do.

```c
void recursive(int n) {
    if (n >= 10) return;
    printf("%d\n", n);
    recursive(n + 1);
}

int main(void) {
    recursive(0);
    return 0;
}
```

- Convert the loop into a recursive function

Done! But it looks more complicated.



STRUCTURE OF A RECURSIVE FUNCTION

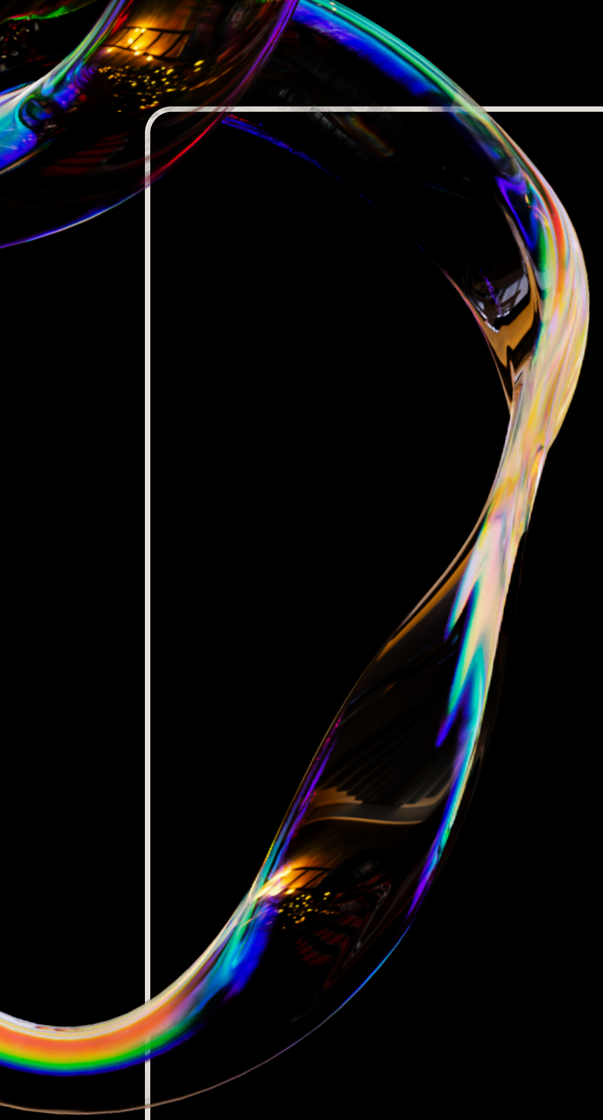
Recursive functions can do anything a loop can do.

```c
void recursive(int n) {
    if (n >= 10) return;
    printf("%d\n", n);
    recursive(n + 1);
}

int main(void) {
    recursive(0);
    return 0;
}
```

- Convert the loop into a recursive function

  Done! But it looks more complicated.

- When do we want to use recursion instead of loops?



STRUCTURE OF A RECURSIVE FUNCTION

Recursive functions can do anything a loop can do.

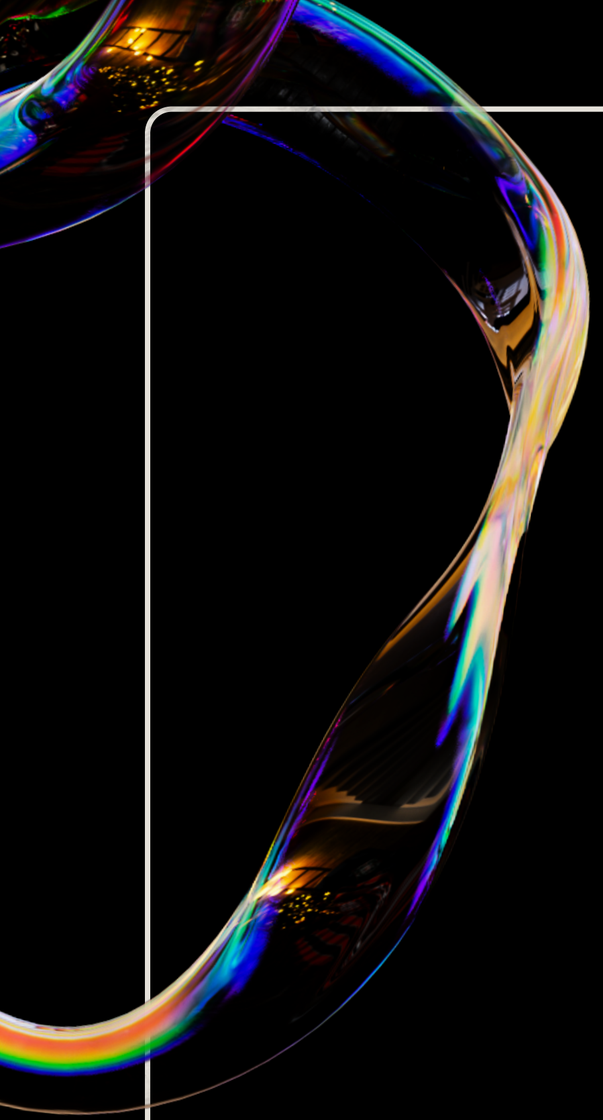
```c
void recursive(int n) {
    if (n >= 10) return;
    printf("%d\n", n);
    recursive(n + 1);
}

int main(void) {
    recursive(0);
    return 0;
}
```

- Convert the loop into a recursive function

  Done! But it looks more complicated.

- When do we want to use recursion instead of loops?

  Backtracking, file systems, etc...



STRUCTURE OF A RECURSIVE FUNCTION

Recursive functions can do anything a loop can do.

```c
void recursive(int n) {
    if (n >= 10) return;
    printf("%d\n", n);
    recursive(n + 1);
}

int main(void) {
    recursive(0);
    return 0;
}
```

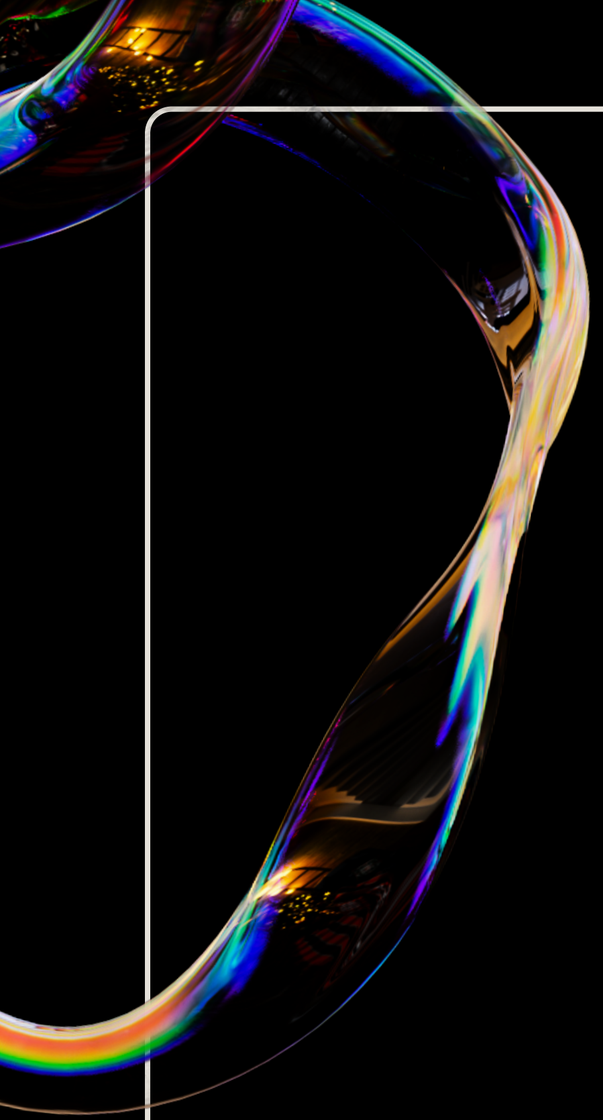
- Convert the loop into a recursive function

  Done! But it looks more complicated.

- When do we want to use recursion instead of loops?

  Backtracking, file systems, etc...

- Recursive function are more powerful that loops



STRUCTURE OF A RECURSIVE FUNCTION

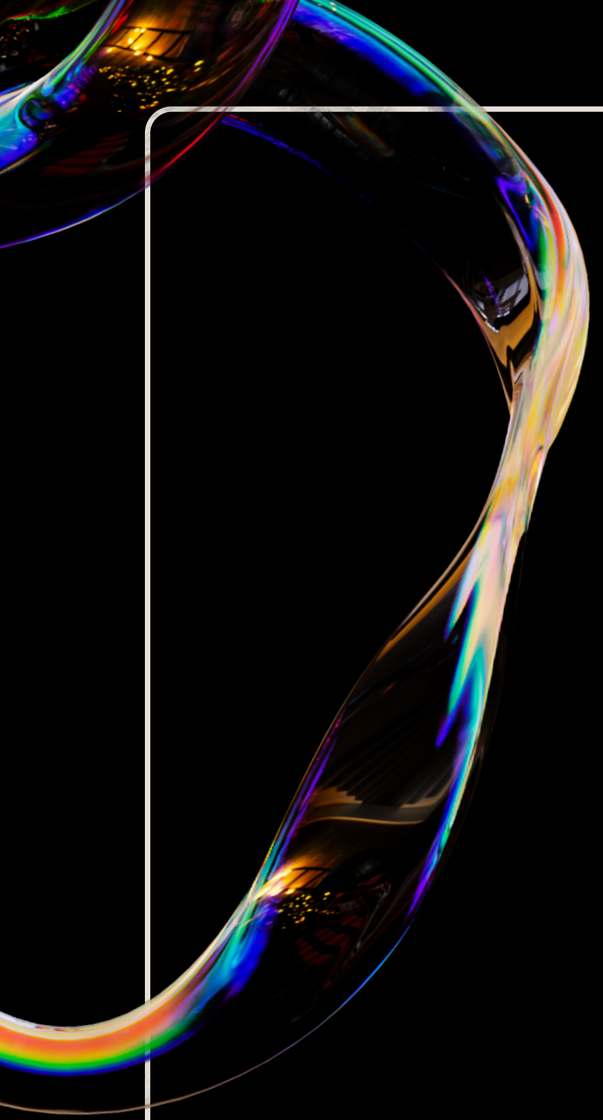
Recursive functions can do anything a loop can do.

```c
void recursive(int n) {
    if (n >= 10) return;
    printf("%d\n", n);
    recursive(n + 1);
}

int main(void) {
    recursive(0);
    return 0;
}
```

Most important!

- Base cases
- Recursive statements

That’s all you need to remember for recursive functions.



STRUCTURE OF A RECURSIVE FUNCTION

Recursive functions can do anything a loop can do.

```c
void recursive(int n) {
    if (n >= 10) return;
    printf("%d\n", n);
    recursive(n + 1);
}

int main(void) {
    recursive(0);
    return 0;
}
```

Most important!

- Base cases
- Recursive statements

That's all you need to remember for recursive functions.

$$f(x) = \begin{cases} g(x) & \text{if } x \in B \\ h(x, f(p(x))) & \text{otherwise} \end{cases}$$


MACHINE
CODE

MACHINE CODE

- Computers speak in terms of 0's and 1's. These are called bits; the alphabet is called binary, the language is called machine code.

MACHINE CODE

- Computers speak in terms of 0's and 1's. These are called bits; the alphabet is called binary, the language is called machine code.

- Programmer’s can’t read 0's and 1's, or understand it well. This is why we program in source code.

MACHINE CODE

- Computers speak in terms of 0's and 1's. These are called bits; the alphabet is called binary, the language is called machine code.
- Programmer’s can’t read 0's and 1's, or understand it well. This is why we program in source code.
- At one point, we need to convert between source code and machine code.

MACHINE CODE

- Computers speak in terms of 0's and 1's. These are called bits; the alphabet is called binary, the language is called machine code.
- Programmer’s can’t read 0's and 1's, or understand it well. This is why we program in source code.
- At one point, we need to convert between source code and machine code.
- Compilers make this conversion for us, parsing a source code program and emitting machine code which then can be executed by the computer.

COMPILATION PIPELINE

- Pre-processor (.c file → .c file)

Expands #include and #define directives, to create a “whole” source code file

COMPILATION PIPELINE

- Pre-processor (.c file → .c file)

Expands #include and #define directives, to create a “whole” source code file

- Compilation (.c file → .s file)

Creates an assembly file based on the source code file

COMPILATION PIPELINE

- Pre-processor (.c file → .c file)

Expands #include and #define directives, to create a “whole” source code file

- Compilation (.c file → .s file)

Creates an assembly file based on the source code file

- Assembler (.s file → .o file)

Creates a object (machine code) file based on the assembly file

COMPILATION PIPELINE

- Pre-processor (.c file → .c file)

Expands #include and #define directives, to create a “whole” source code file

- Compilation (.c file → .s file)

Creates an assembly file based on the source code file

- Assembler (.s file → .o file)

Creates a object (machine code) file based on the assembly file

- Linking (.o file → executable)

Creates an executable based off of multiple object files and resolves library calls, symbols, metadata, etc.s

GG

FILES/SLIDES SENT OUT AFTER CLASS

Any questions?