School of Computing & Communications | Lancaster University

SCC.111 Software Development – Lecture 2: Writing Code

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

"C"

This term we start with introducing an imperative programming language called **C**. First created in 1970's to build UNIX. C is compact, low-level, and is used to generate fast, efficient code that exploits hardware features well.

Imperative programming

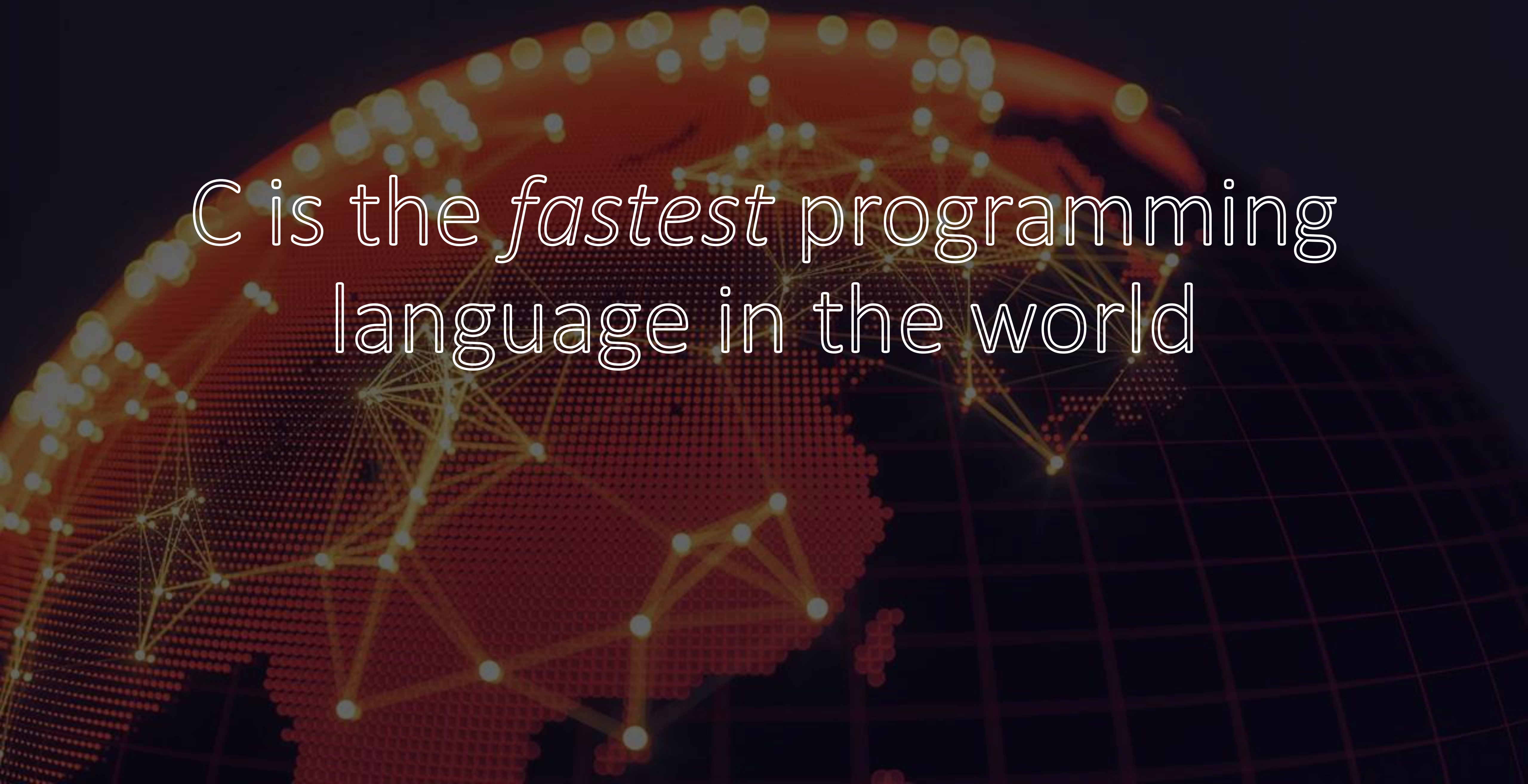
- is a *programming paradigm* that uses statements that change a program's state
- So, a critical step is to think through *what* to represent to solve our problem
  - – a total, a brightness, a geo-location, a time, an audio sample, a command for a robot…
- As well as, *how* our program should manipulate this

Facts about C

The C Language was not called C at the beginning. It was first known as New B!

It was released in 1972 and still ranks 2nd on the TIOBE Index and 4th in IEEE Spectrum Index

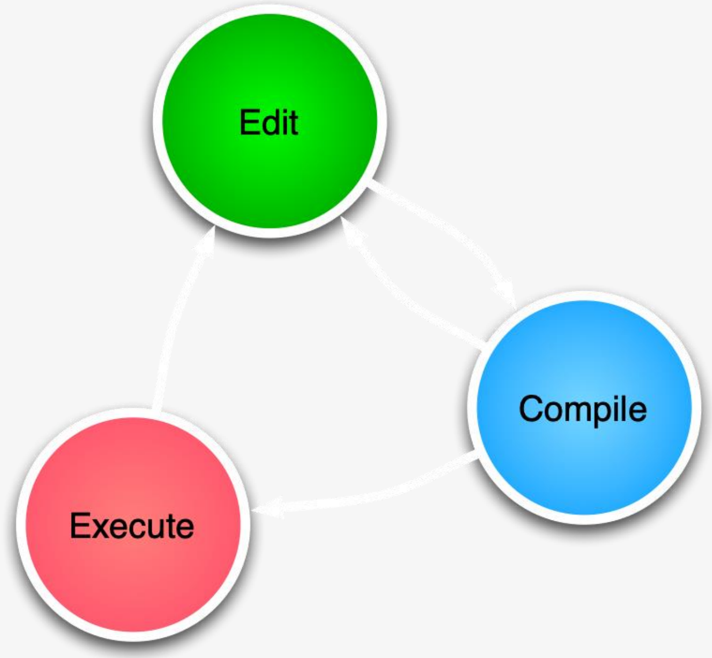
C is the mother of all modern programming languages. C++, Java, JavaScript, Go, C#, PHP, Python, Perl, Rust, etc. all borrow the syntax and many C concepts



C is the *fastest* programming language in the world

C is a ‘compiled language’

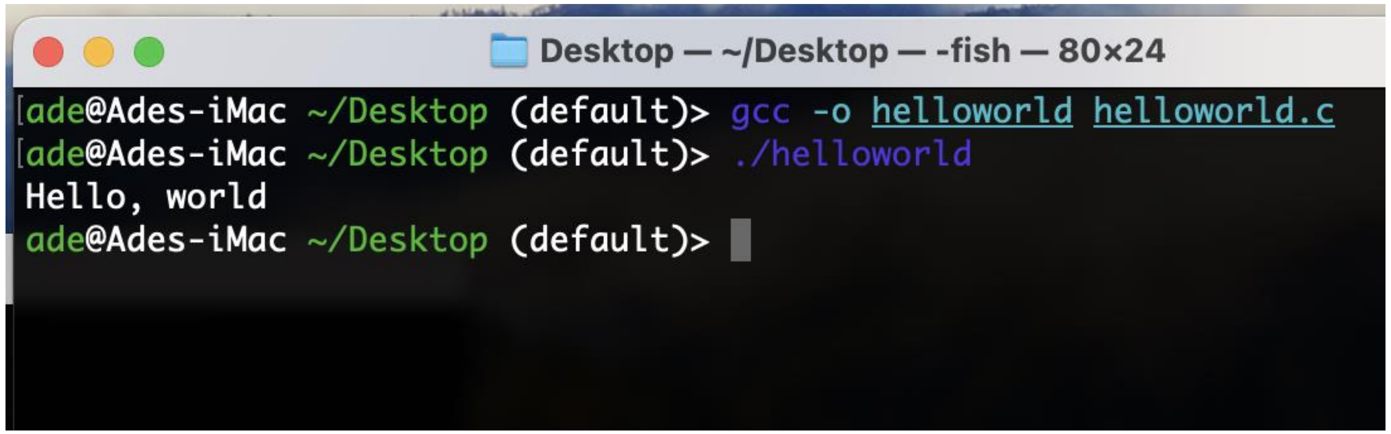
- In C we will use standard tools to form a “tool chain”:
  - use **a text editor** to write and edit code
  - An *explicit* compilation phase using a **compiler** translates C into assembler then machine code (*iff* the program syntax is correct!)
  - a resulting **executable** we can run on *the target platform*



Our first C program

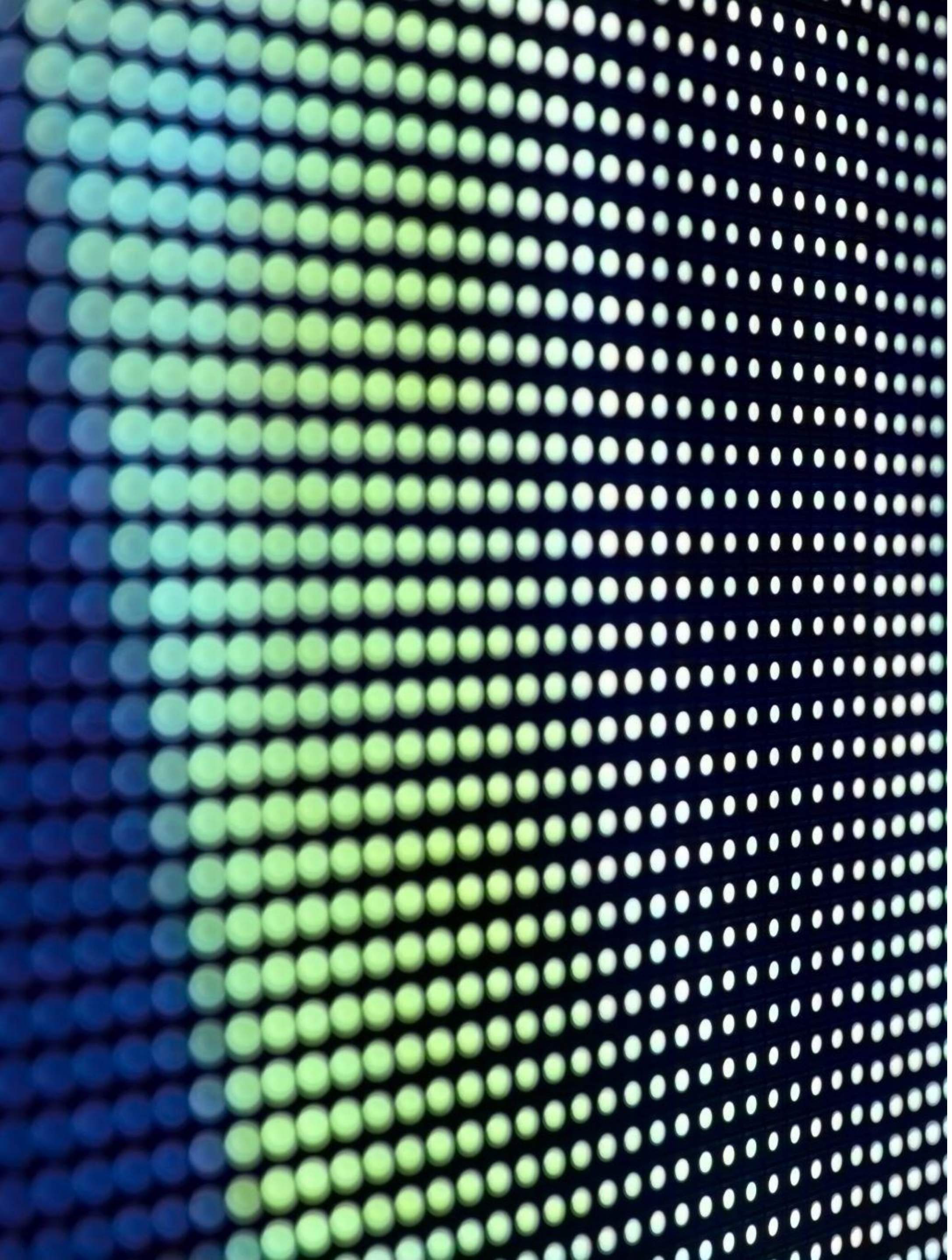
```
#include <stdio.h>

int main()
{
  printf("Hello, world\n");
}
```

```
$ gcc -o helloworld helloworld.c
```



Desktop — ~/Desktop — -fish — 80×24

```
ade@Ades-iMac ~/Desktop (default)> gcc -o helloworld helloworld.c
ade@Ades-iMac ~/Desktop (default)> ./helloworld
Hello, world
ade@Ades-iMac ~/Desktop (default)> 
```



It solves *a very simple* problem

- Producing a single line of output on the screen (in the terminal output)
  - "Hello, world\n"
  - Or more strictly, outputs the sequence of characters ‘H’, ‘e’, ‘l’ etc. followed by a newline character \n to the ‘output stream’ (e.g. terminal)

Even this program has complexity

- And adheres to a **strict set of rules** or **syntax** that make up the language “C” – and it’s not valid C if it doesn’t follow those rules!

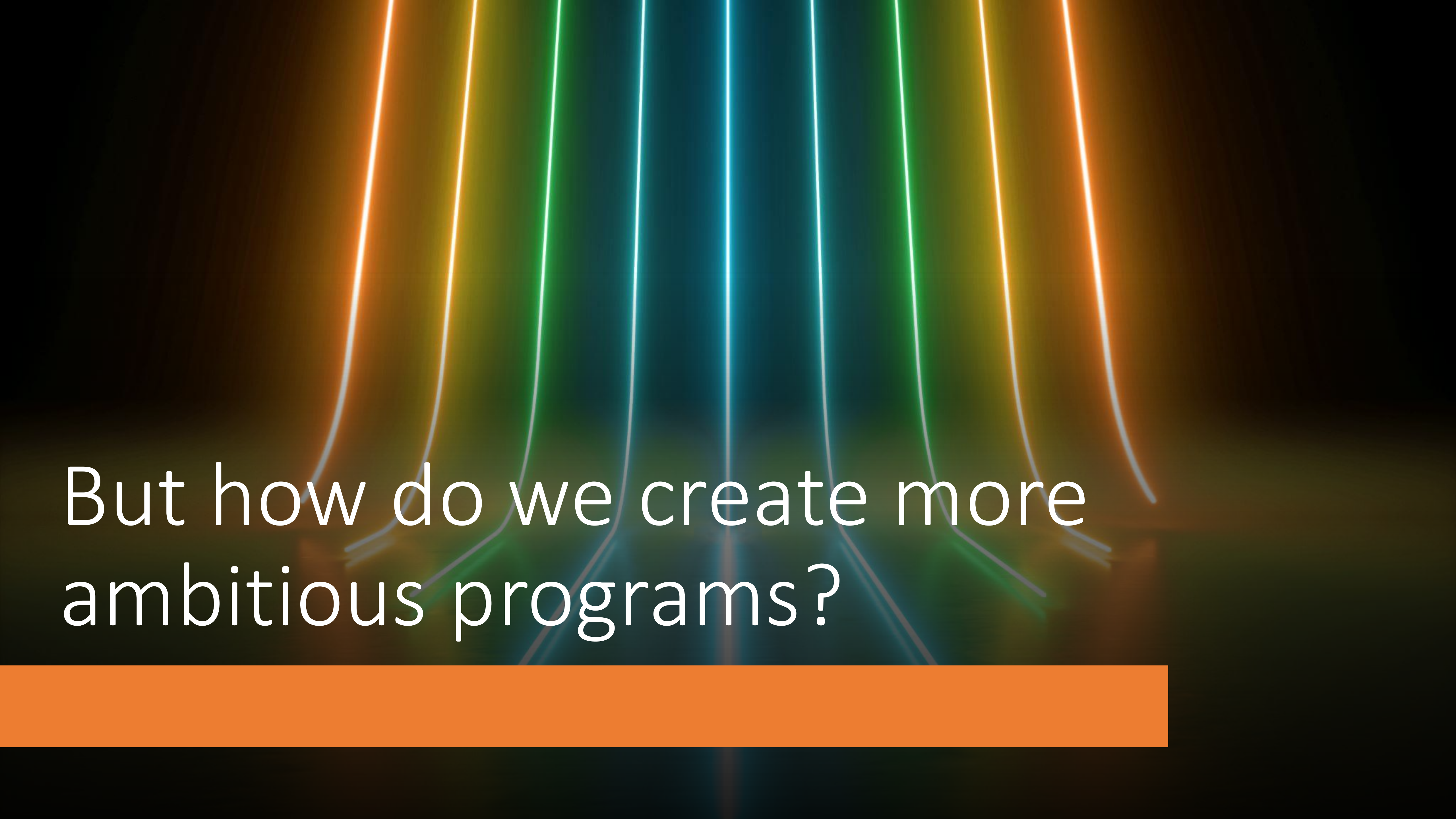
```
#include <stdio.h>

int main()
{
  printf("Hello, world\n");
}
```

Copy the contents of file ‘stdio.h’ into this file (we’ll see why later in the course)

Declare ‘main’ a function that is the entry point/ starting point for execution when the program runs (c.f. ‘program flow’)

{} make up the ‘body’ of the function

Call a function printf, passing in “Hello, world\n” – **printf** does the actual output for us (we’ll meet this again later on)

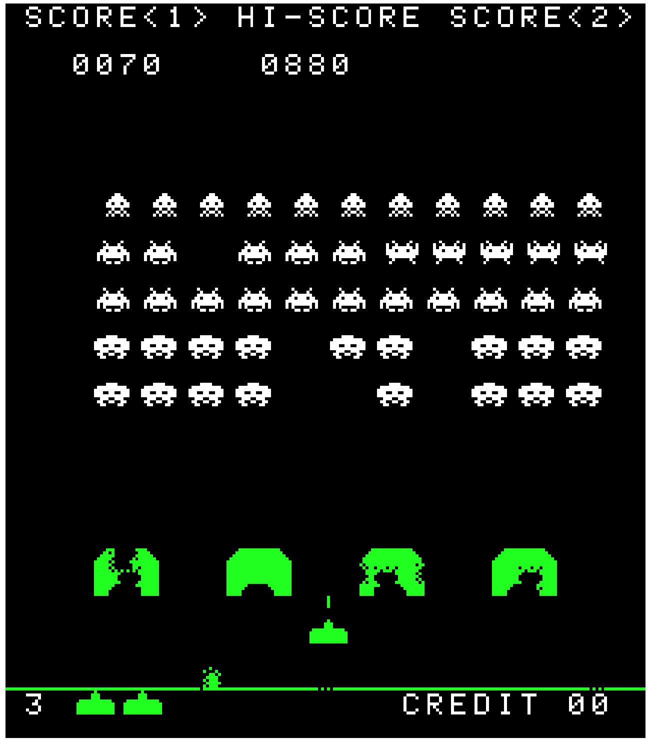


But how do we create more ambitious programs?

I want space invaders please…

High level functionality

Top down



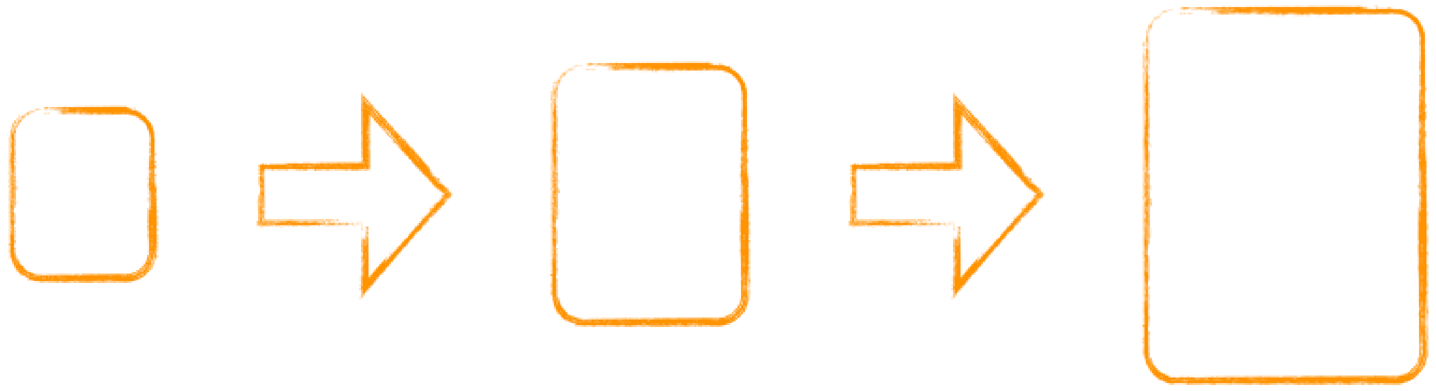
Bottom up

C code

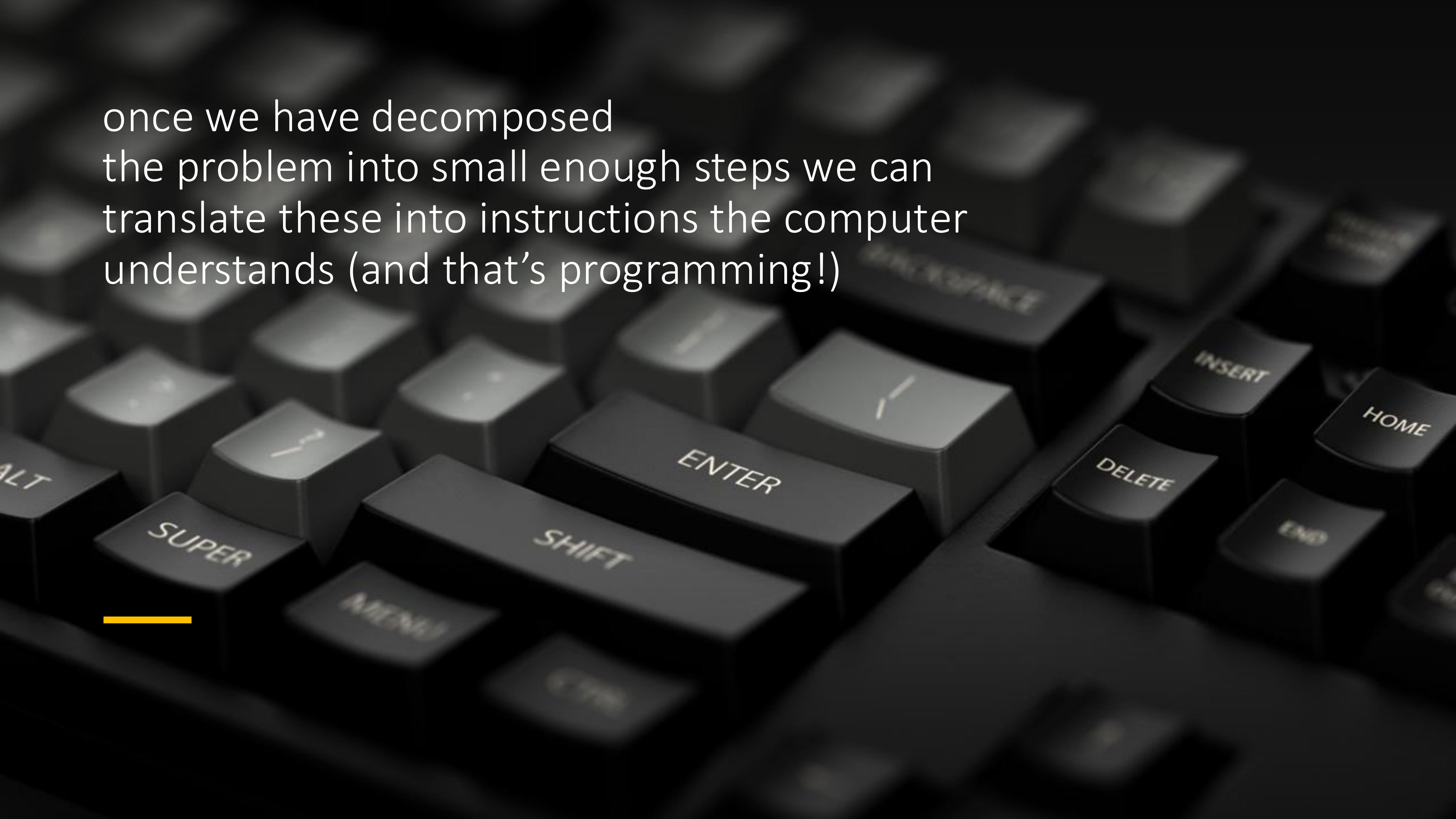
Low level code

Decomposing problems

- We solve pretty much all problems by repeatedly decomposing them into a series of smaller steps until the problem is tractable.

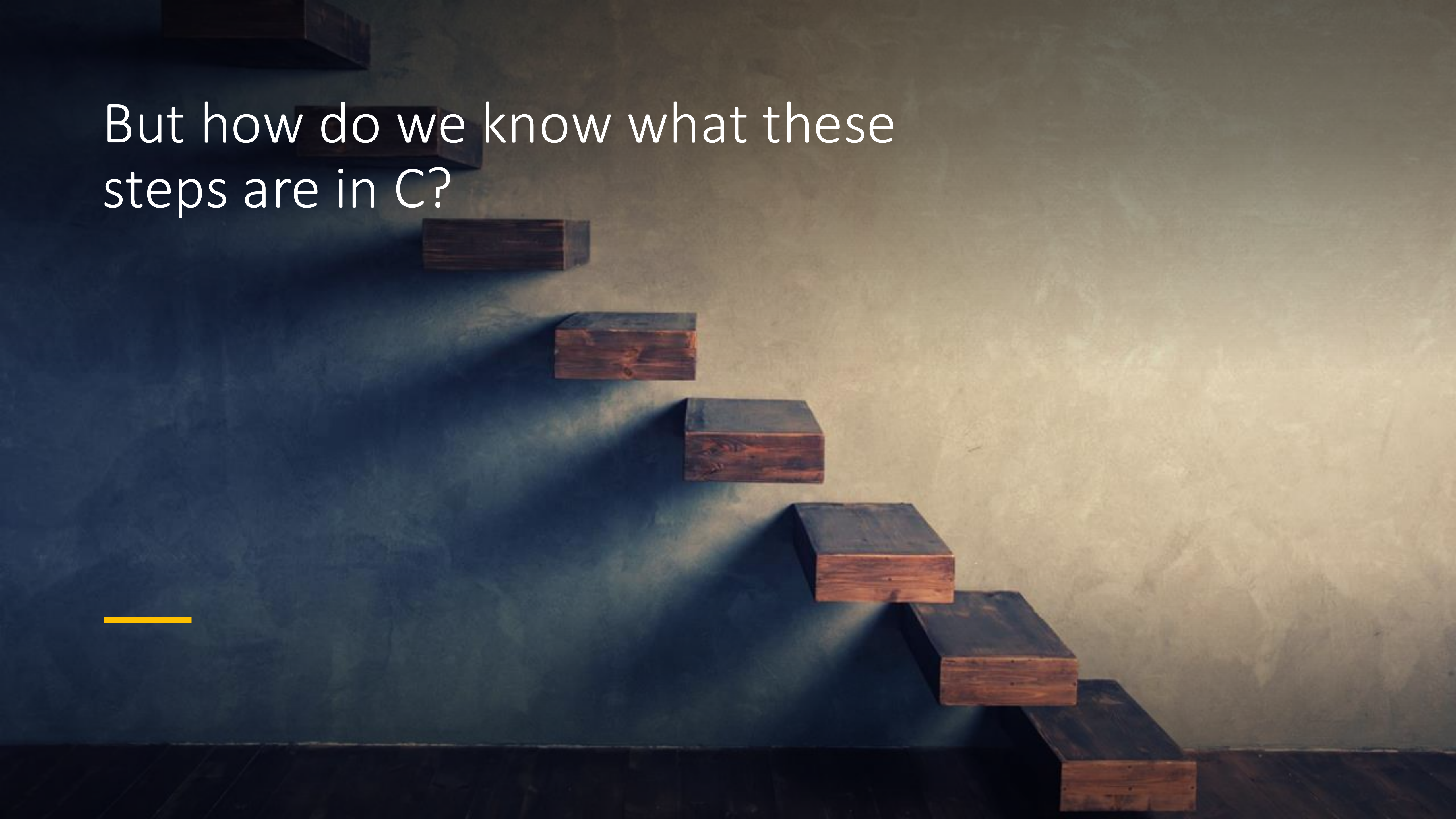


Iterative refinement



once we have decomposed
the problem into small enough steps we can
translate these into instructions the computer
understands (and that’s programming!)

But how do we know what these steps are in C?



English?

1

More than a million total words

2

170,000 words in current use

3

20,000-30,000 words (native speaker, 5,000 by age 4!)

C total vocabulary = 32 words

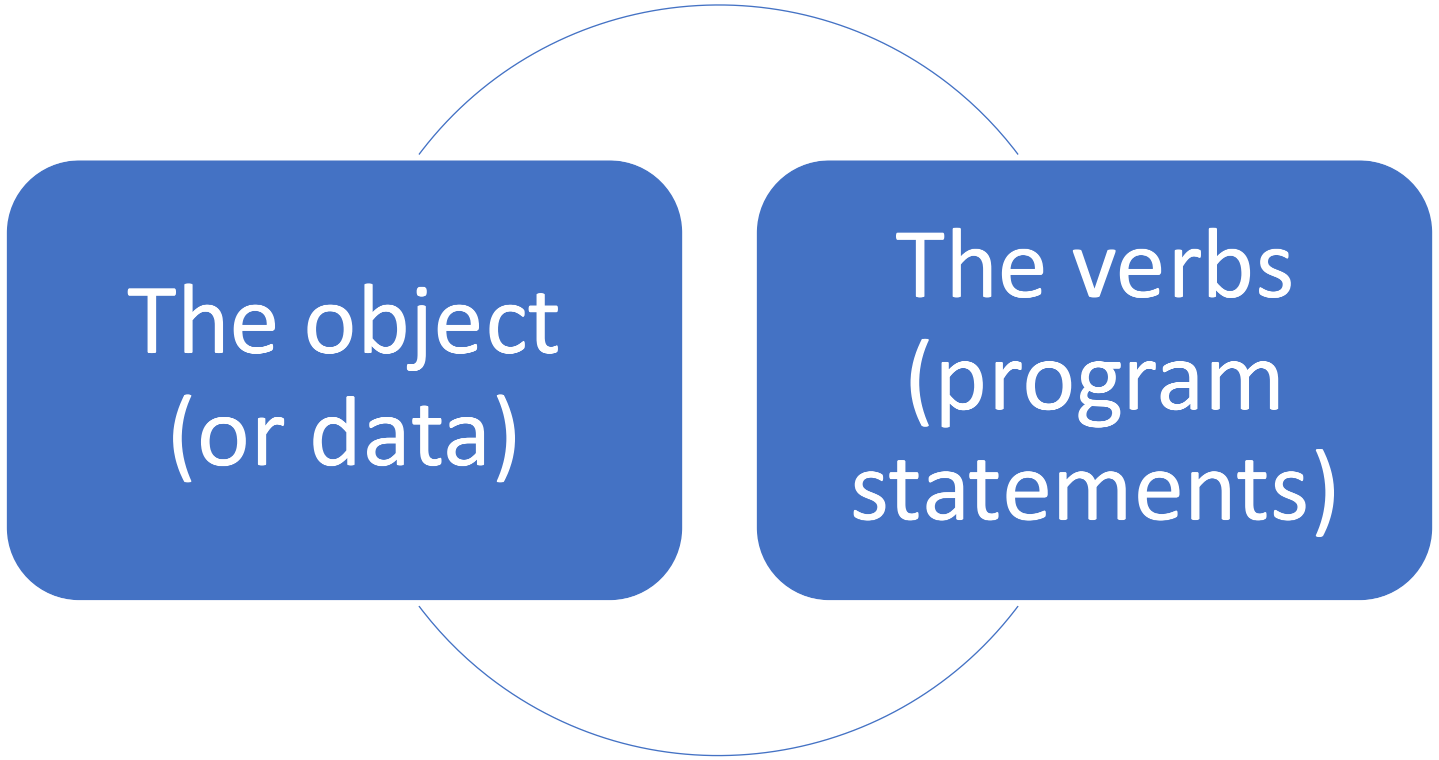
```
auto        extern      sizeof
break       float       static
case        for         struct
char        goto        switch
const       if          typedef
continue    int         union
default     long        unsigned
do          register    void
double      return      volatile
else        short       while
enum        signed
```

English has some rules… as do all languages

SUBJECT VERB OBJECT (SVO)

Students (S) write (V) programs (O)

Similarly, when we write programs…



Programming languages also have **rigid rules**

- Syntax, program statements follow rules on how keywords or reserved words are used and combined (the '**how**')
- Variables (named stores) hold our data (the '**what**')
- Operators allow us **to manipulate** this data to produce the output or effect we want

see K&R Appendix A, pg.191

Let’s look at a few C essentials

Understanding our second C program

```
#include <stdio.h>

int main()
{
  // I'm declaring 'counter' a variable
  // and setting it to 0

  int counter = 0;

  while (counter < 10) {
    // while counter is less than 10, add one
    counter++;
  }

  printf("The variable counter is now %d\n", counter);
}
```

Now we have a ‘counter’ variable (named space for a whole number)

We can manipulate it, here adding one each time round a ‘loop’ (operator ++)

There’s quite a lot to cover in even this simple program…

Variables

- We store data in *variables*, and change their value to calculate the answer we want
- Variables in C are `typed’, this affects how operators work (e.g. precision, size)

```
int main()
{
  int i;
  char c;
  float f;
  double d;
  short s;
  unsigned char b;
  unsigned int u;
}
```

Variables have a ‘type’ such as ‘int’. Depending on the type you can only store specific things, e.g. whole numbers, floating point

Normally variables can store positive and negative (signed) values, unless you say they’re unsigned

The rule here is that type reserved words refer to the variable on the right hand side, e.g. ‘i’ is the variable and it’s an ‘int’. When ‘main’ finishes it returns an ‘int’ value.

Operators

```c
#include <stdio.h>

int main()
{
  int a, b;

  a = -1;
  a = a + 3;
  b = (a * 5) + 2;
  b = b / 2 + 2;

  printf("b is %d\n", b);

  return 0;
}
```

Binary operators	Unary operators
+ Addition	++ Increment
- Subtraction	-- Decrement
/ Division	+= Add
* Multiplication	*= Multiply
% Remainder (mod)	
<< Shift left, >> right	

Brackets first
Orders (i.e. Powers and Square Roots, etc.)
Division and **M**ultiplication (left-to-right)
Addition and **S**ubtraction (left-to-right)

Variable naming rules

- Begins with a letter or underscore symbol
  - Consists of letter, digits or underscore only (no spaces)
  - Cannot be a reserved word like ‘`if`’, ‘`while`’ etc.
  - C is case sensitive!
    - `Total` `total` and `TOTAL` are all different variables
  - By convention do use camelCase, and avoid using all CAPS as these convey a different meaning

Examples:

```
Distance
milesPerHour
_voltage
goodChoice
high_level
MIN_RATE
```

Programs are made up of statements

- Examples:
  - Variable definition: `<type> <name>;`
  - Assignments and calculations: `variable = expression;` i.e. evaluate the expression (right hand side) and store the result in variable (left hand side)
  - Comments to the reader `// /**/`
- Expressions:
  - Basic arithmetic, `a + 2, a / b, a * a,` etc.
  - Results of function calls – *don't worry for now!*
  - Control flow: `if`, `for`, `while`, … (*more in a moment…*)

Program flow in our second C program

```c
#include <stdio.h>

int main()
{
  // I'm declaring 'counter' a variable
  // and setting it to 0

  int counter = 0;

  while (counter < 10) {
    // while counter is less than 10, add one
    counter++;
  }

  printf("The variable counter is now %d\n", counter);
}
```

Counter controls the flow round a loop linked to the keyword ‘while’

This is a logical test < or less than which is either true or false

The {} denote the body of the loop inside the body of the function ‘main’

More formally, while is

```
while <expression>
    <statement>
```

- Where `<statement>` may actually be multiple statements in `{ }`

Loops are controlled by logical operators

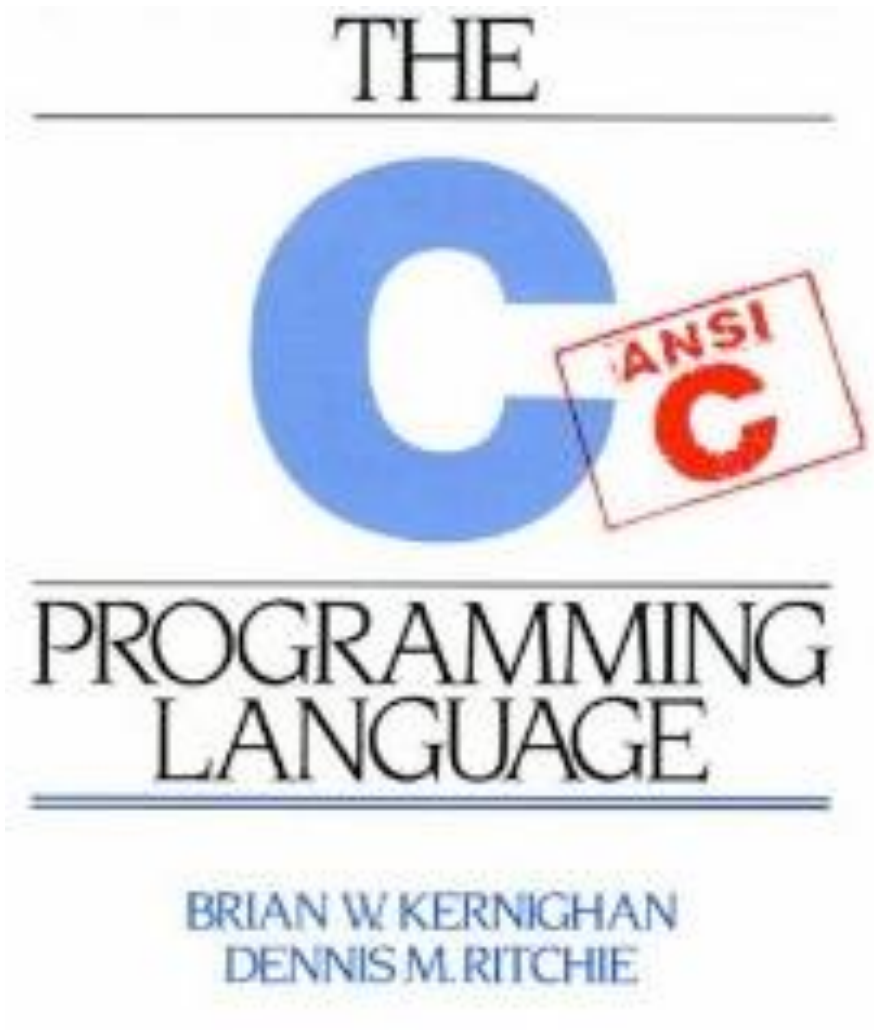
- < less than
- > greater than
- <= and >=
- == and !=
- ! Boolean NOT (unary, logically negates)
- && Boolean AND
- || Boolean OR

```
while (counter < 10) {
  // ..
}

while (guess != -1) {
  // ..
}

while (input < 0 || input > 10) {
  // ..
}
```

And we'll meet more examples and more of the language next week!

Total vocabulary = 32 words

auto	extern	sizeof
break	**float**	static
case	**for**	struct
char	goto	switch
const	if	typedef
continue	**int**	union
default	long	**unsigned**
do	register	void
double	return	volatile
else	**short**	**while**
enum	signed	



Reading: See K&R ch3, pg 35; ch3 pg. 55

Summary

- Introduced core C constructs, operators, and types - *a small language of just 32 words!*
- Programming is about decomposing problems into steps the computer can understand
- We introduced loops, logical and arithmetic binary and unary operators
- Next week: many examples of how this goes together with *data types and problem solving in C*