Dennis Ritchie

# Introduction to Programming in C

Prof. Darrell Long

CSE 13S

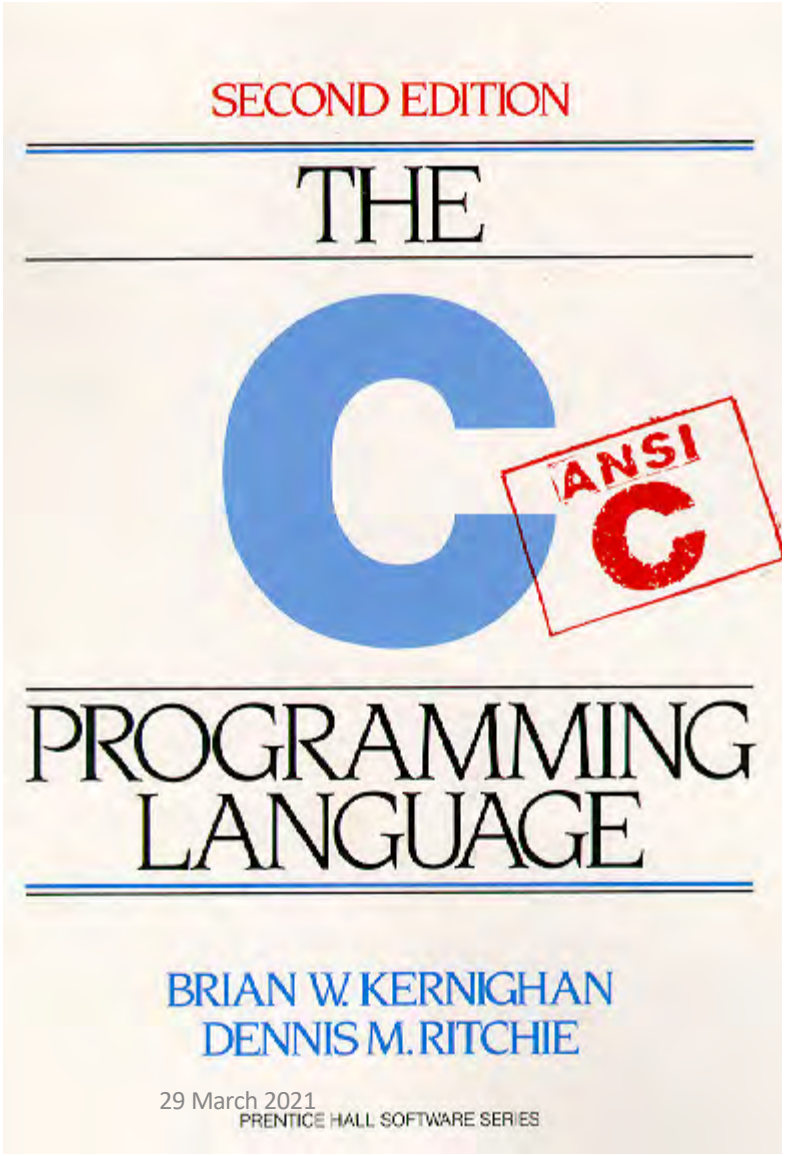
# Where did **C** come from?

- Derived from B, written by Ken Thompson
- Influenced by
  - CPL and BCPL languages
  - PDP-11 processor

| Year | C Standard |
|---|---|
| 1972 | Birth |
| 1978 | K&R **C** |
| 1989 | ANSI **C** |
| 1999 | C99 |
| 2011 | C11 |
| 2018 | C18 |

# Hello, World!

# Compile and run!




# You *must* read this book!

- Yes, it presents a dated version of C:
  - ANSI C versus C11
  - Most people still do not use the features of C11 daily.
- It is simple,
- Clear,
- Short, and
- A classic.

} All good things!

It is how all of your professors learned **C**.

| Scale | 0 | 58 | 100 |
|---|---|---|---|
| K | Dead | Dead | Dead |
| ℃ | Snow | Hottest on Earth | Water Boils |
| ℉ | Minnesota | Santa Cruz | Tucson |

- Kelvin is the scale for science:
  - 0 K ➠ All nuclear motion ceases (absolute zero)
- ℃ = K – 273.15
- ℉ = ℃ × 9 ÷ 5 + 32

# Temperatures

# First Attempt

darrell — vi fahr.c — 41×16

```c
#include <stdio.h>

// Print a table of ℉ to ℃, for 0 to 300

int main(void) {
  int fahr, celsius;
  int lower = 0, upper = 300, step = 20;
  fahr = lower;
  while (fahr <= upper) {
    celsius = 5 * (fahr - 32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

Should we be concerned?

darrell — -bash — 41×16

```
pascal:~ darrell$ cc -o fahr fahr.c
pascal:~ darrell$ ./fahr | head -10
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
160     71
180     82
pascal:~ darrell$
```

# Second Attempt

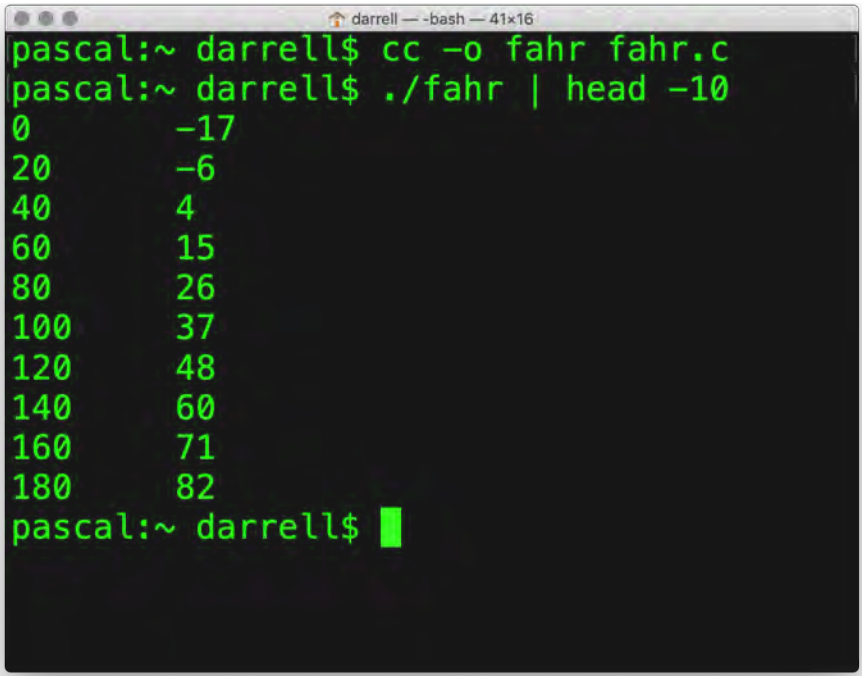
darrell — vi cvtF.c — 42×16

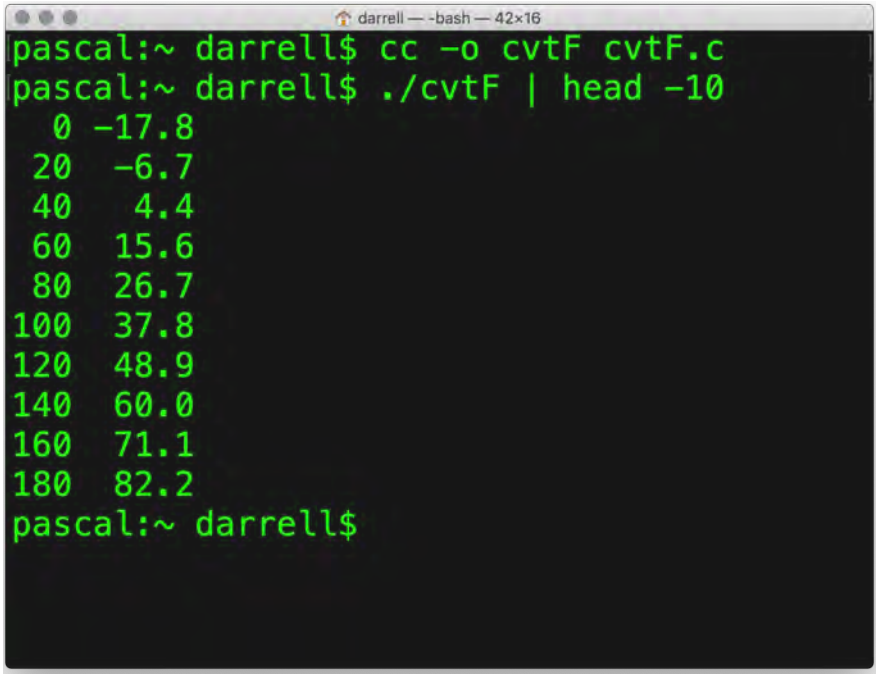
```
#include <stdio.h>

// Print a table of ℉ to ℃, for 0 to 300

int main(void) {
  float fahr, celsius;
  int lower = 0, upper = 300, step = 20;
  fahr = lower;
  while (fahr <= upper) {
    celsius = (5.0 / 9.0) * (fahr - 32);
    printf("%3.0f%6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

darrell — -bash — 42×16

```
pascal:~ darrell$ cc -o cvtF cvtF.c
pascal:~ darrell$ ./cvtF | head -10
  0 -17.8
 20  -6.7
 40   4.4
 60  15.6
 80  26.7
100  37.8
120  48.9
140  60.0
160  71.1
180  82.2
pascal:~ darrell$
```

# Comparing Results




Integer arithmetic truncates!

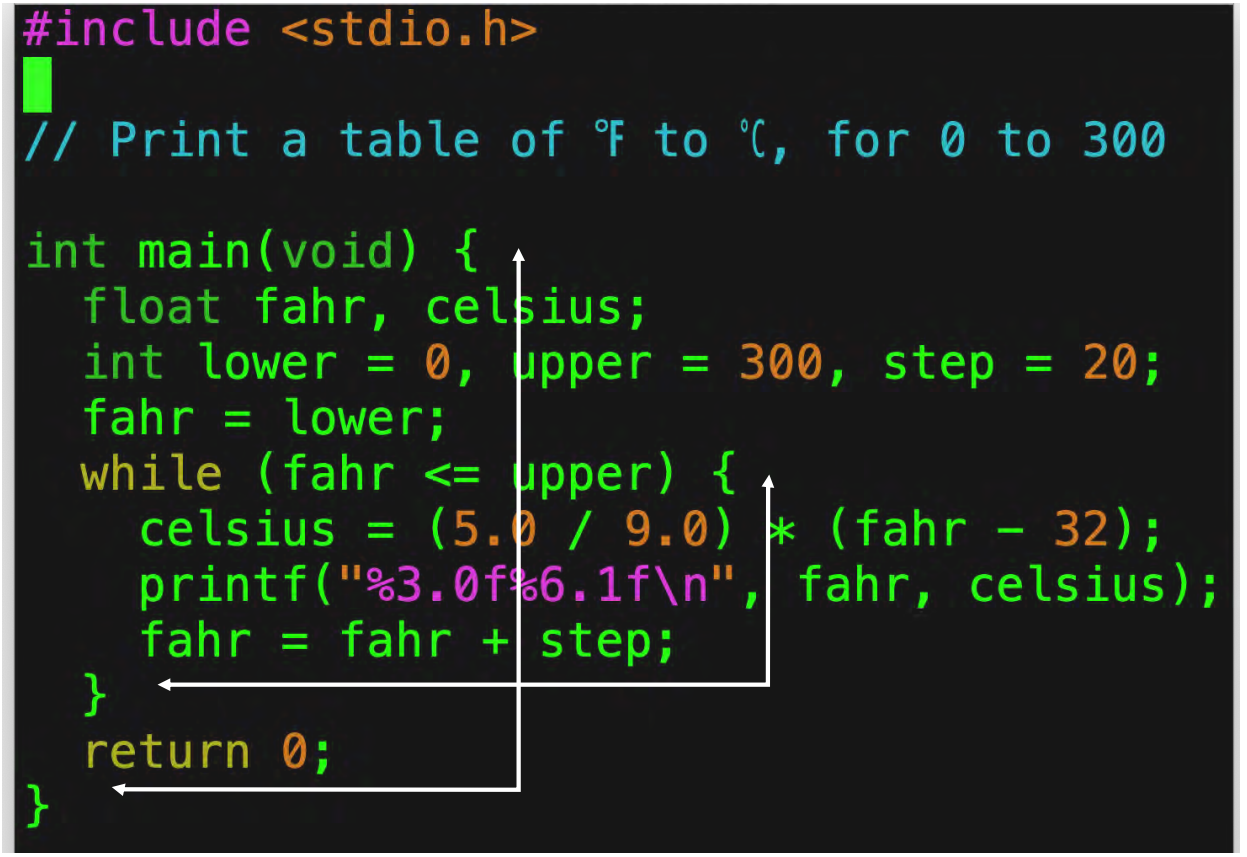
## `int main(void)`

- There is exactly one main program.
- It's really just a function with a special name.
- It returns an `int` as its status value.
- In this case, it takes no arguments.

```c
#include <stdio.h>

// Print a table of ℉ to ℃, for 0 to 300

int main(void) {
  float fahr, celsius;
  int lower = 0, upper = 300, step = 20;
  fahr = lower;
  while (fahr <= upper) {
    celsius = (5.0 / 9.0) * (fahr - 32);
    printf("%3.0f%6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

# { … }

- **C** is a “curly brace” language
  - Many were influenced by **C**.
- { } are used to group statements.
- Use them even for single statements.
- { } is called a block.
- A block introduces a local scope.
  - We will discuss this later.



# while ()

- `while ()` is the simplest loop.
- It is called a top-test loop.
  - More on that later.
- It executes *while* the Boolean statement inside the `()` is true.
- Always use `{}` with `while`, even for a single statement.

```
#include <stdio.h>

// Print a table of ℉ to ℃, for 0 to 300

int main(void) {
  float fahr, celsius;
  int lower = 0, upper = 300, step = 20;
  fahr = lower;
  while (fahr <= upper) {
    celsius = (5.0 / 9.0) * (fahr - 32);
    printf("%3.0f%6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

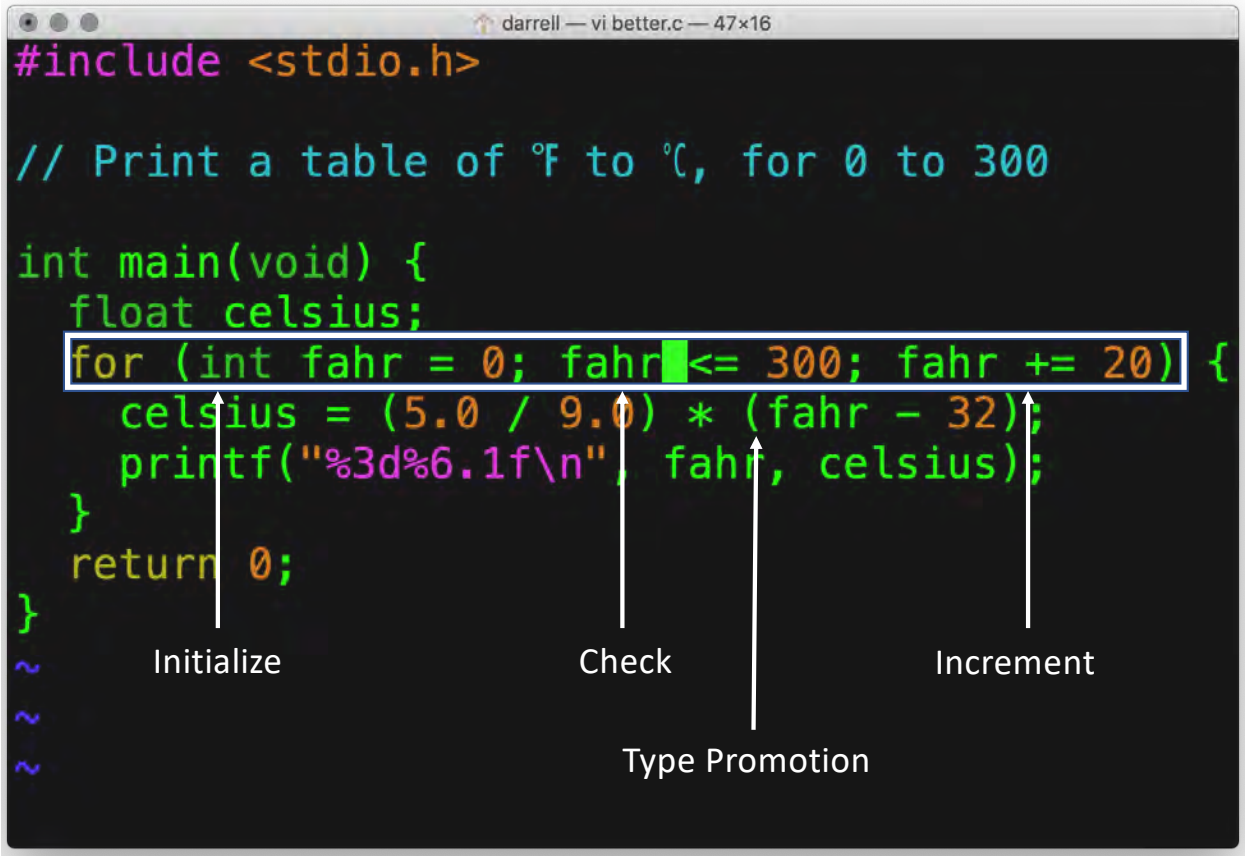
# `printf()`

- `printf()` is *just a function* in the standard I/O library.
  - It's not magic, you could write it too!
- `"%6.1f"` is called a format string.
- It says: there is a floating point number, 6 characters wide, with one digit after the decimal place.

```c
#include <stdio.h>

// Print a table of ℉ to ℃, for 0 to 300

int main(void) {
  float fahr, celsius;
  int lower = 0, upper = 300, step = 20;
  fahr = lower;
  while (fahr <= upper) {
    celsius = (5.0 / 9.0) * (fahr - 32);
    printf("%3.0f%6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

# for ()

- Why is this better?
  - Many would say it is easier to understand.
- The initialization is explicit.
- The test and increment are all visible at the top.
- Clarity above all.
  - Do not try to be clever.



# Declaring Variables

```c
int main(void) {
  char *s, c;
  int i;
  float f;
  double d;

  s = "This is a string";
  c = 47;                          // This is a small integer
  i = 12345678;                    // This is a bigger integer
  f = 3.1415;                      // This is a floating point number
  d = 2.718281828459045235354; // This is more precise
}
```

- In **C**, you must declare a variable before you can use it.
- Declaring it means to specify its *type*.
- For now, we will be concerned with the scalar types:
  - `char`, `int`, and
  - `float`, and `double`.

```
{
  float x = 1.61803; // Golden ratio
  {
    float y = 1.0 - x; // y is only here, but x is out there
  }
  {
    int x = 1962; // Both x and y exist only here
    int y = 1962 - 1967;
  }
}
```

## Scope

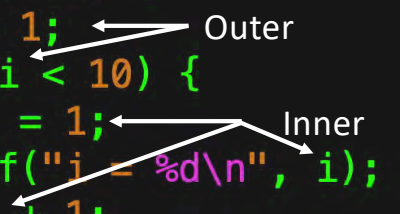
- Each pair of "curly braces" { … } introduce what is called a *scope*.
- The scope of a variable tells us where that variable exists, or is defined.

# Scoping Rules

- When you have a set of { … } you create a *new scope*.
- You can create new *local variables* in that scope.
- Those local variables can have any type.
- They can have any legal name.
- If they have the same name as a variable in an outer scope, then they *hide* that variable.

```c
#include <stdio.h>
int main(void) {
  int i = 1;
  while (i < 10) {
    int i = 1;
    printf("i = %d\n", i);
    i = i + 1;
  }
}
```

```c
#include <stdio.h>
int main(void) {
  int i = 1;
  while (i < 10) {
    int i = 1;
    printf("i = %d\n", i);
    i = i + 1;
  }
}
```

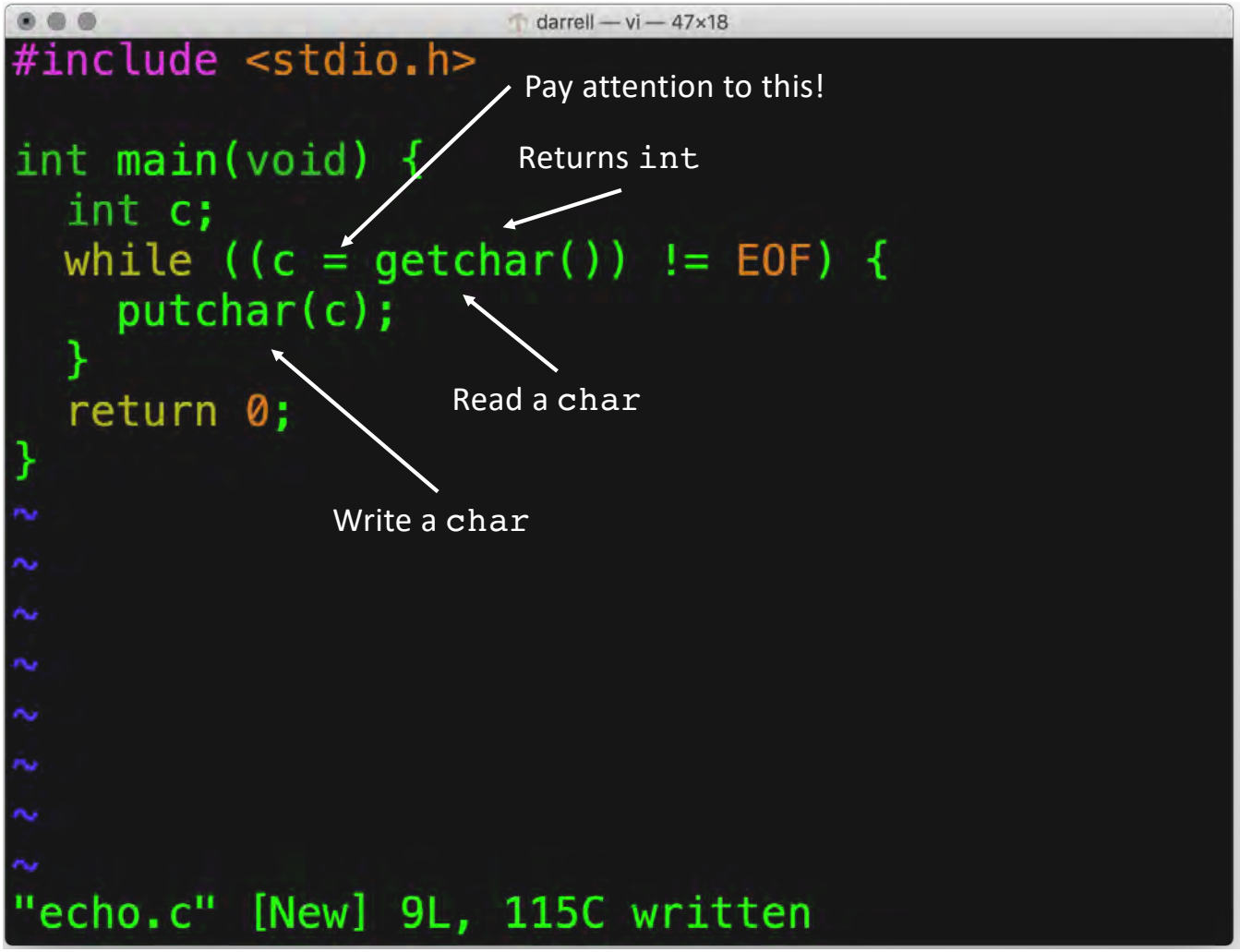



```
pascal $ cc nest.c -o nest.c
pascal $ ./nest.c | head -10
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
i = 1
```

# So what happens?

- The outer `i` gets *hidden* by the inner `i`!
  - The outer `i` is *always* 1.
  - The inner `i` is 1, then 2, then back to 1 again.
- Why didn't **C** warn me?
  - **C** always does exactly what you tell it, and what you did was legal.

Copy `stdin` to `stdout`



# Count Lines

darrell — vi — 47×18

```
#include <stdio.h>

int main(void) {
  int c, lines = 0;
  while ((c = getchar()) != EOF) {
    if (c == '\n') {
      lines += 1;
    }
    putchar(c);
  }
  printf("%d lines copied.\n", lines);
  return 0;
}
~
~
~
~
```

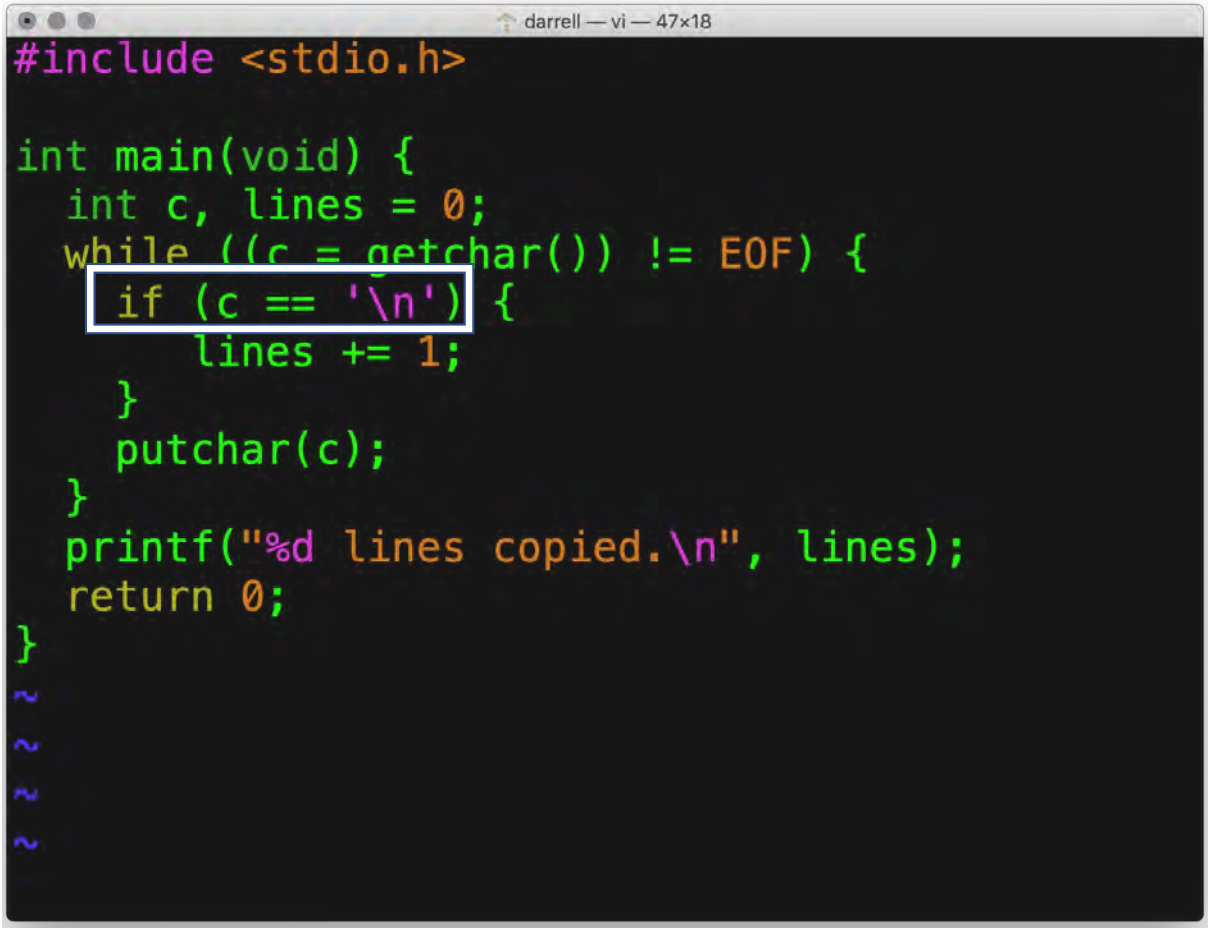
lines = lines + 1

darrell — -bash — 47×18

```
pascal:~ darrell$ cc -o lc lc.c
pascal:~ darrell$ ./lc < lc.c
#include <stdio.h>

int main(void) {
  int c, lines = 0;
  while ((c = getchar()) != EOF) {
    if (c == '\n') {
      lines += 1;
    }
    putchar(c);
  }
  printf("%d lines copied.\n", lines);
  return 0;
}
13 lines copied.
pascal:~ darrell$
```

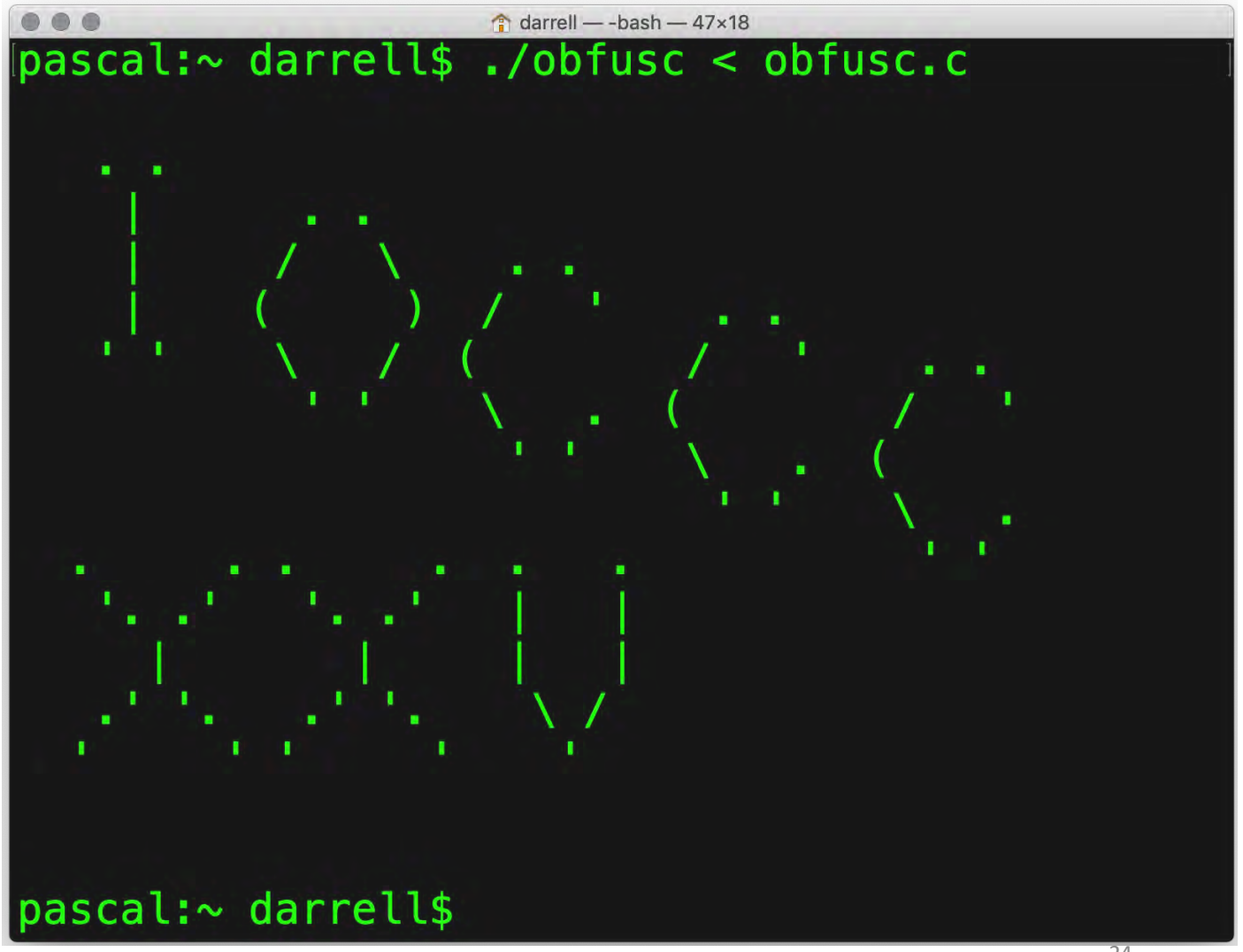
# `if ()`

- `if` executes the next statement if the Boolean expression is true.
- Even through `{}` are not required for a single statement, *always* use them.
- Why?
  - It avoids errors when adding statements.



Do not try to be clever…

What does it do?



# Vade Mecum

```c
#include <stdint.h>

#define MAXIMUS 1000000
#define SYMBOLA 7

int v[] = { 1000, 500, 100,  50,  10,   5,   1 };
int c[] = {  'M', 'D', 'C', 'L', 'X', 'V', 'I' };

char *itor(int n) {
  static char b[2 * MAXIMUS / 1000]; // pertinax sacculo secreti
  int s = 0;
  n = n < 0 ? 0 : n; // nihil esse maior
  n %= MAXIMUS; // potest esse maior quam maximus
  for (int i = 0; i < SYMBOLA; i += 1) {
    while (n >= v[i]) {
      b[s++] = c[i];
      n -= v[i];
    }
  }
  b[s] = '\0';
  return b;
}
```

Walk with me through this code…

- In it, you will see many of the things that you will learn this quarter.

# Summary

**C** provides the basic set of statements and operators that you expect from an *imperative* programming language.

It is relatively low level — close to the machine (in contrast to a language like Python).

You can write unreadable programs — Don't try to be clever by being obscure.

Clarity is paramount!

**C** is powerful, but
Programming in **C** can be dangerous!