

# INTRODUCTION TO C

3

CS/COE 0449  
Introduction to  
Systems Software

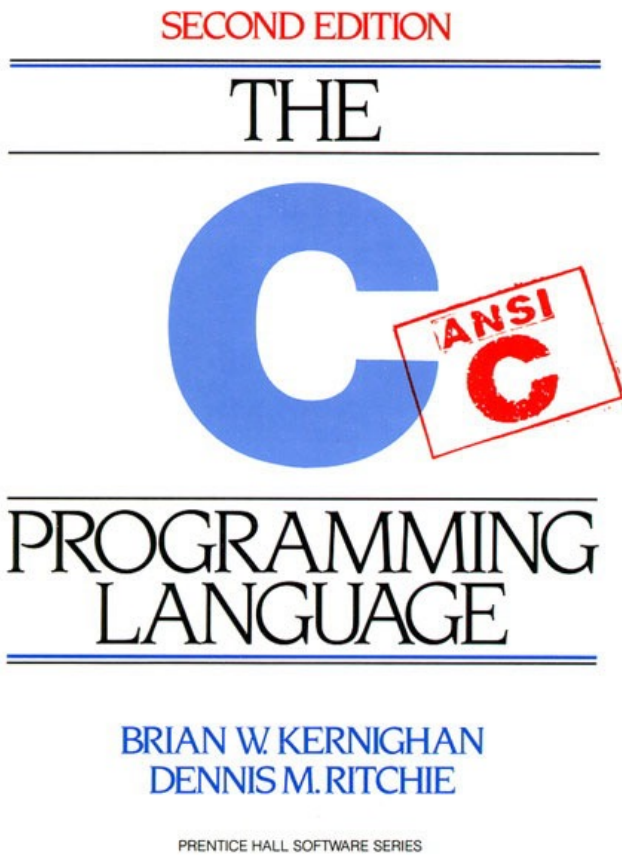
Luis Oliveira

(with content borrowed from wilkie and Vinicius Petrucci)

# OVERVIEW OF C

What You C is What You Get

# C: The Universal Assembly Language


























*C is not a “very high-level” language, nor a “big” one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.*

— Kernighan and Ritchie

- Allows writing programs to exploit underlying features of the architecture
  - memory management, special instructions, parallelism.

# C: Relevance – check the link for updated numbers

Rank	Language	Type	Score
1	Python	  	100.0
2	Java	  	95.4
3	C	  	94.7
4	C++	  	92.4
5	JavaScript		88.1
6	C#	   	82.4
7	R		81.7
8	Go	 	77.7
9	HTML		75.4
10	Swift	 	70.4

- From IEEE Spectrum:
  - <https://spectrum.ieee.org/top-programming-languages>
- Still relatively popular...
  - Lots of legacy code.
  - Lots of embedded devices.
  - Python, JavaRE, R, JS are all written in C.












# TIOBE index

*TIOBE Programming Community index is an indicator of the popularity of programming languages*

Jan 2021	Jan 2020	Change	Programming Language	Ratings	Change
1	2	⬆	C	17.38%	+1.61%
2	1	⬇	Java	11.96%	-4.93%
3	3		Python	11.72%	+2.01%
4	4		C++	7.56%	+1.99%
5	5		C#	3.95%	-1.40%
6	6		Visual Basic	3.84%	-1.44%
7	7		JavaScript	2.20%	-0.25%
8	8		PHP	1.99%	-0.41%
9	18	⬆	R	1.90%	+1.10%
10	23	⬆	Groovy	1.84%	+1.23%
11	15	⬆	Assembly language	1.64%	+0.76%












<https://www.tiobe.com/tiobe-index/>

# TIOBE index

Jan 2022	Jan 2021	Change	Programming Language		Ratings	Change
1	3	▲		Python	13.58%	+1.86%
2	1	▼		C	12.44%	-4.94%
3	2	▼		Java	10.66%	-1.30%
4	4			C++	8.29%	+0.73%
5	5			C#	5.68%	+1.73%
6	6			Visual Basic	4.74%	+0.90%
7	7			JavaScript	2.09%	-0.11%
8	11	▲		Assembly language	1.85%	+0.21%
9	12	▲		SQL	1.80%	+0.19%
10	13	▲		Swift	1.41%	-0.02%
11	8	▼		PHP	1.40%	-0.60%












<https://www.tiobe.com/tiobe-index/>

# TIOBE index

Jan 2023	Jan 2022	Change	Programming Language		Ratings	Change
1	1			Python	16.36%	+2.78%
2	2			C	16.26%	+3.82%
3	4	▲		C++	<b>12.91%</b>	<b>+4.62%</b>
4	3	▼		Java	12.21%	+1.55%
5	5			C#	5.73%	+0.05%
6	6			Visual Basic	4.64%	-0.10%
7	7			JavaScript	2.87%	+0.78%
8	9	▲		SQL	2.50%	+0.70%
9	8	▼		Assembly language	1.60%	-0.25%
10	11	▲		PHP	1.39%	-0.00%
11	10	▼		Swift	1.20%	-0.21%

<https://www.tiobe.com/tiobe-index/>

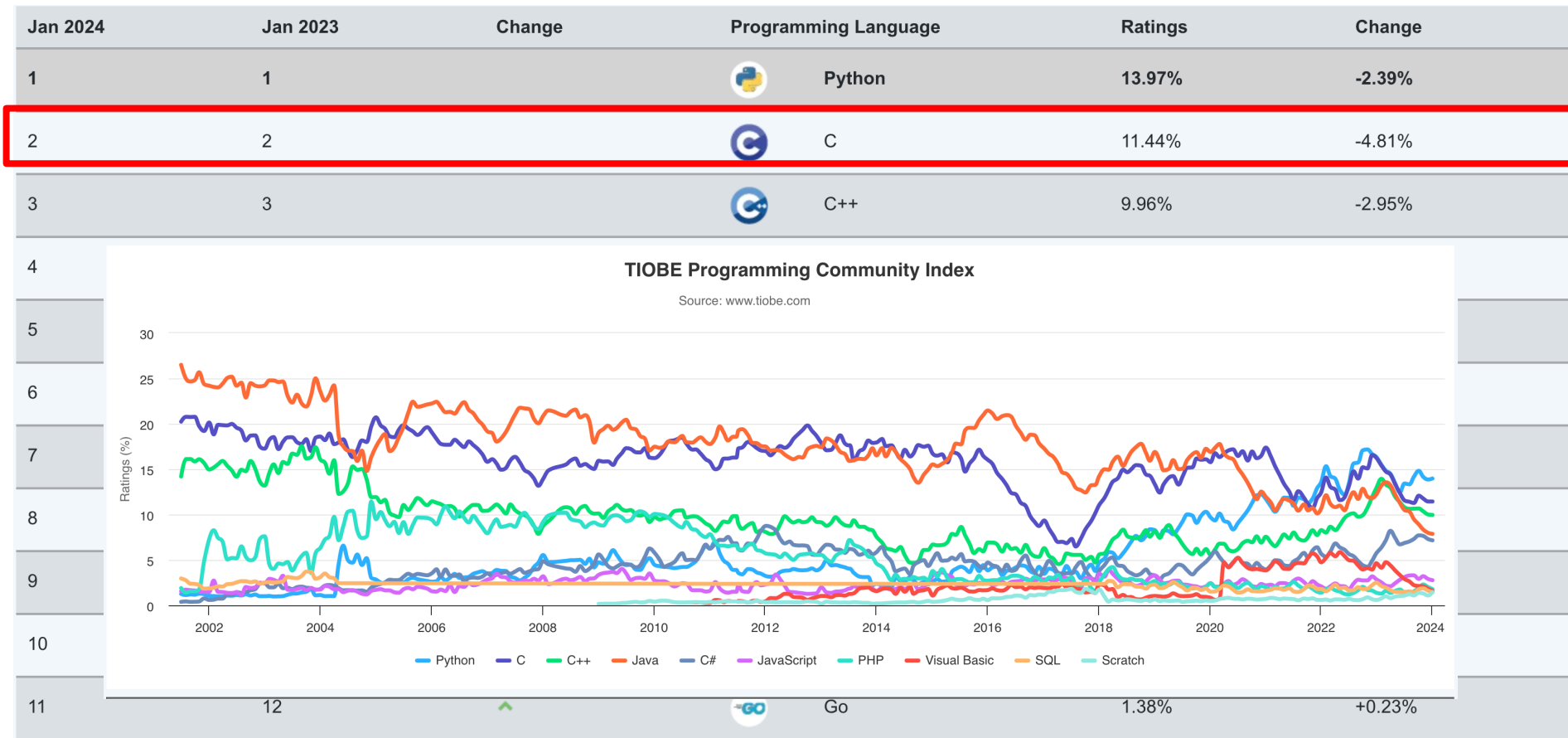
# TIOBE index

Jan 2024	Jan 2023	Change	Programming Language		Ratings	Change
1	1			Python	13.97%	-2.39%
2	2			C	11.44%	-4.81%
3	3			C++	9.96%	-2.95%
4	4			Java	7.87%	-4.34%
5	5			C#	7.16%	+1.43%
6	7	⬆		JavaScript	2.77%	-0.11%
7	10	⬆		PHP	1.79%	+0.40%
8	6	⬇		Visual Basic	1.60%	-3.04%
9	8	⬇		SQL	1.46%	-1.04%
10	20	⬆		Scratch	1.44%	+0.86%
11	12	⬆		Go	1.38%	+0.23%

<https://www.tiobe.com/tiobe-index/>



# TIOBE index



# THE C SYNTAX

Nothing can be said to be certain, except death and C-like syntaxes.

# C Dialects

- You will see a lot of different styles of C in the world at large.
  - The syntax has changed very little.
- There have been a few different standard revisions.
  - C89 – ANSI / ISO C
    - `gcc -ansi -Wpedantic hello.c`
  - C99 – Adds ‘complex’ numbers and single-line comments
    - `gcc -std=c99 hello.c`
  - C11 – Newer than 99 (laughs in Y2K bug) starts to standardize Unicode and threading libraries (better text and parallel execution).
    - `gcc -std=c11 hello.c`
  - C18 – Minor refinement of C11. The current C standard.
    - `gcc -std=c18 hello.c`
  - C23 – Soon... Binary literals `0b100101` 😊. Checked int arithmetic.
- We will more or less focus on the C99 standard in our course.
  - I’ll try to point out some newer things if they are relevant.

# C vs. Java

	C	Java
Type of Language	Procedural language	Object Oriented
Programming Unit	Function	Class = Abstract Data Type
Compilation	<i>gcc hello.c</i> - creates machine language code	<i>javac Hello.java</i> - creates Java virtual machine language bytecode
Execution	<i>./a.out</i> - loads and executes program	<i>java Hello</i> - interprets bytecodes
hello, world	<pre>#include&lt;stdio.h&gt; int main(void) {     printf("Hello World\n");     return 0; }</pre>	<pre>public class HelloWorld {     public static void main(String[] args) {         System.out.println("Hello World");     } }</pre>
Storage	Manual ( <b>malloc</b> , <b>free</b> )	Automatic (garbage collection)

# C vs. Java

	C	Java
Comments	<code>/* ... */</code> or <code>// ...end of line</code>	<code>/* ... */</code> or <code>// ...end of line</code>
Constants	<code>#define</code> , <code>const</code>	<code>final</code>
Preprocessor	Yes	No
Variable declaration	Hmmm, it depends 😊 Old versions are weird!	Before you use it
Variable naming conventions	<code>sum_of_squares</code>	<code>sumOfSquares</code>
Accessing a library	<code>#include &lt;stdio.h&gt;</code>	<code>import java.io.File;</code>

# Hello World

```
// Includes the declaration of the printf function
#include <stdio.h>

// The main function first of your code to be executed
int main(void) {
    // The rules for printing strings are much like Java.
    // For instance, \n denotes a newline.
    printf("Hello World\n");

    // Returning a 0 is usually considered "successful"
    return 0;
}
```

# Hello World

```
// Includes the declaration of the printf function
```

```
#include <stdio.h>
```

Somewhere in  
"stdio.h"

This means it  
returns an int

It accepts the memory address of  
an "array of const chars"

```
int printf(const char * format, ...);
```

Everything must be **declared**  
before being **used**.

And a variable number of other  
arguments (literally stated as "...")

```
// For instance, \n denotes a newline.
```

```
printf("Hello World\n");
```

This includes functions!

```
// Returning a 0 is usually considered "successful"
```

```
return 0;
```

```
}
```

printf is a tricky one :)

# The “main” function

```
// File includes go at the top of the file:
#include <stdio.h>

// The main function first of your code to be executed
// The void is used when there are no arguments.
// We will look at traditional command-line arguments later.
int main(void) {
    // Programs return an int (a word) to reflect errors.

    // Returning a 0 is usually considered "successful"
    return 0;
}
```



# Declaring variables

```
int main(void) {  
    // Variables are declared within functions, generally  
    // at the top. Type followed by name.  
  
    // They are optionally initialized using an '='  
    int n = 5;  
  
    // When they are not initialized, their value is  
    // arbitrary.  
  
    // Returning a 0 is usually considered "successful"  
    return 0;  
}
```

# Casting

```
int main(void) {  
    // When you initialize, the given literal is coerced  
    // to that type.  
    int n = -50000;  
  
    // You can then coerce the value between variables.  
    // No matter how much nonsense it might be:  
    char smaller = n;  
  
    // You can explicitly cast the value, as well:  
    unsigned int just_nonsense = (unsigned int)n;  
  
    return 0;  
}
```

# Integer Sizes – Revisted: sizeof

```
#include <stdio.h>    // Gives us 'printf'
#include <stddef.h>    // Gives us the 'size_t' type

int main(void) {
    // The special 'sizeof' macro gives us the byte size
    // The 'size_t' type is provided by the C standard
    // and is used whenever magnitudes are computed.
    size_t int_byte_size = sizeof(int);
    size_t uint_byte_size = sizeof(unsigned int);

    printf("sizeof(int): %lu\n", int_byte_size);
    printf("sizeof(unsigned int): %lu\n", uint_byte_size);

    return 0;
}
```

# Integer Sizes – Revisted

```
#include <stdio.h>    // Gives us 'printf'

int main(void) {
    printf("sizeof(x):    (bytes)\n");
    printf("char:         %lu\n", sizeof(char));
    printf("short:        %lu\n", sizeof(short));
    printf("int:          %lu\n", sizeof(int));
    printf("unsigned int: %lu\n", sizeof(unsigned int));
    printf("long:          %lu\n", sizeof(long));
    printf("float:         %lu\n", sizeof(float));
    printf("double:        %lu\n", sizeof(double));
    return 0;
}
```

# Integers: Python vs. Java vs. C

Language	sizeof(int)
Python 2	>=32 bits (plain ints), infinite (long ints) (Python 3 only has infinite ints)
Java	32 bits
C	Depends on computer; 16 or 32 or 64

- C: `int`
  - integer type that target processor works with most efficiently
  - For modern C, this is generally a good-enough default choice.
- Only guarantee:
  - `sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)`
  - Also, `short` >= 16 bits, `long` >= 32 bits
  - All could be 64 bits
- Impacts portability between architectures

# Constants

```
const float PI = 3.1415; // not a great approximation :)
```

```
int main(void) {
```

```
    // You can use constants in the place of literals:
```

```
    float angle = PI * 2.0;
```

```
    // But, you cannot implicitly modify them:
```

```
    PI = 3.0; // EVEN WORSE approximation NOT ALLOWED!
```

```
    return 0;
```

```
}
```

example.c: In function 'main':

example.c:8:6: error: assignment of read-only variable 'PI'

# Enumerations

```
#include <stdio.h>
```

```
enum { CS445, CS447, CS449 };
```

```
int main(void) {
```

```
    // You can use enums like constants:
```

```
    int my_class = CS449;
```

```
    // They are assigned an integer starting from 0.
```

```
    printf("%d\n", my_class); // Prints 2
```

```
    return 0;
```

```
}
```

# Operators: Java stole 'em from here

```
int main(void) {  
    int a = 5, b = -3, result; // assignment  
  
    // Note: parentheses help group expressions:  
    result = a + b + (a - b); // add, subtract  
    result = a * b / (a % b); // multiply, divide, modulo  
    result = a & b | ~(a ^ b); // and, or, complement, xor  
    result = a << b;           // left shift  
    result = a >> b;           // right shift  
  
    return 0;  
}
```



# Augmented Operators

```
int main(void) {  
    int a = 5, b = -3;  
  
    a += b;    // +=, -= (same as: a = a + b)  
    a *= b;    // *=, /=, %= (ditto: a = a * b)  
    a &= b;    // &=, |=, ^= (no ~= since it is a unary op)  
    a <<= b;   // <<=, >>=  
    a++;      // increment (same as: a = a + 1)  
    a--;      // decrement (ditto: a = a - 1)  
  
    return 0;  
}
```

# Expressions: an expression of frustration!!

```
char a = 0x76;  
short b = 0x5610;  
_____ c = (a & b) // what type is the result?
```

- C often coerces (implicitly casts) integers when operating on them.
- To remove ambiguity, expressions, such as (a & b), result in a type that most accommodates that operation.
- Specifically, C will coerce all inputs of binary operators to at least an `int` type.
  - You'll find that "this is weird, but consistent" is C's general motto

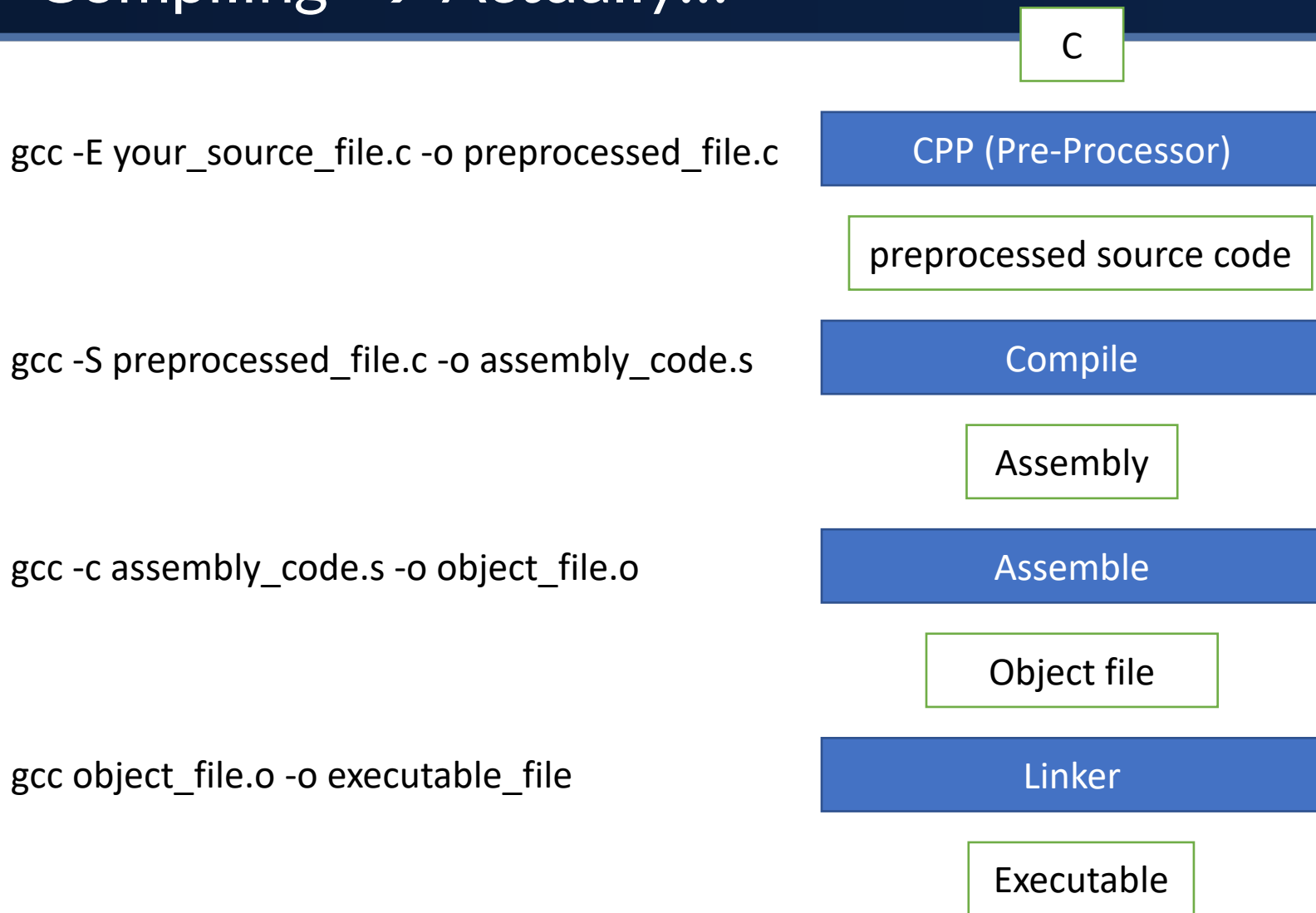
```
printf("%lu\n", sizeof(a & b));    // prints 4  
printf("%lu\n", sizeof('c'));    // prints ?
```

# COMPILING

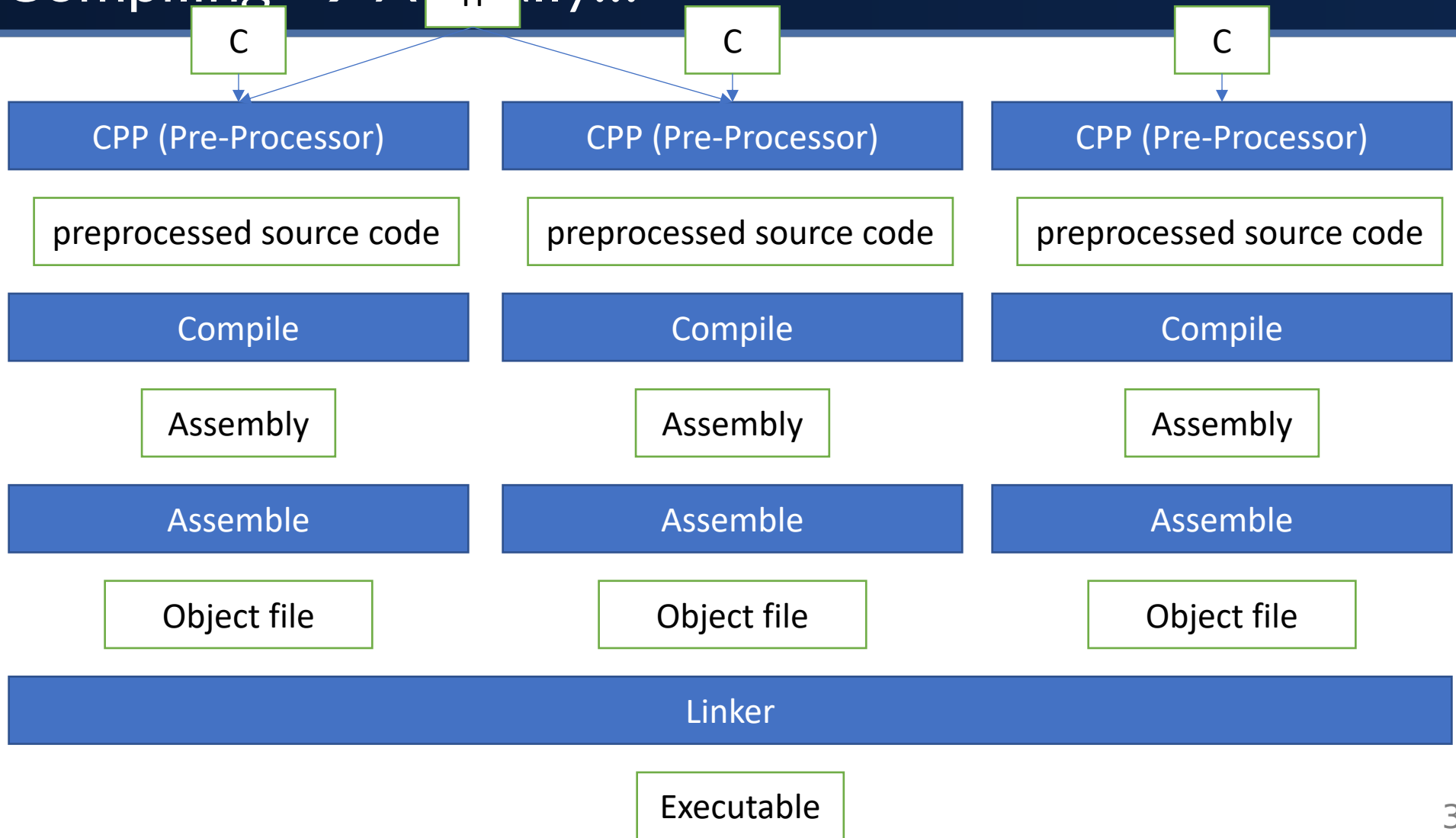
# Compilation

- C is a compiled language.
  - Code is generally converted into machine code.
  - Java, by contrast, indirectly converts to machine code using a byte-code.
  - Python, by contrast to both, interprets the code.
- The difference is in a trade-off about when and how to create a machine-level representation of the source code.
- A general C compiler will typically convert \*.c source files into an intermediate \*.o object file. Then, it will *link* these together to form an executable.
  - Assembly is also part of this process, but it is done behind the scenes.
  - You can have gcc (a common C compiler) spit out the assembly if you want!

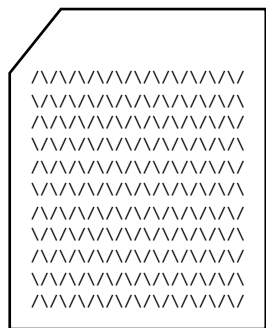
# “Compiling” → Actually...



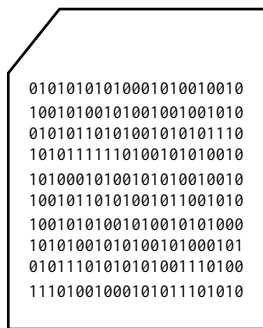
# “Compiling” → Actually...



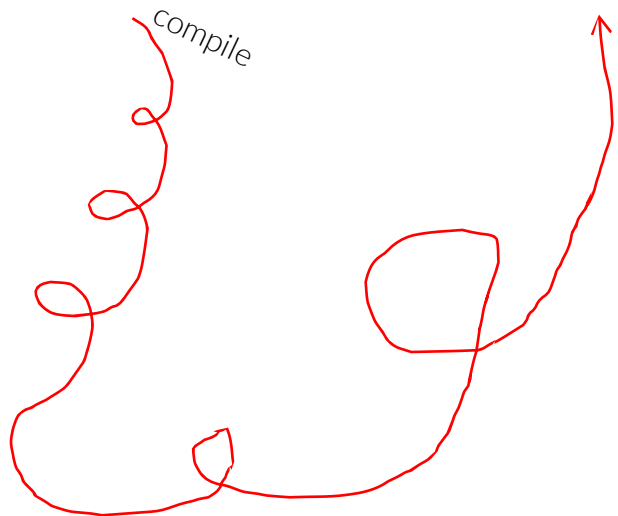
# But usually... Magic!



**hello.c**



**hello**



- The compiler takes source code (\*.c files) and translates them into machine code.
- You can execute this file!  
(cause... magic!)  
(but it's all lies!)
- But what if I want to use multiple files?
  - Magic...
- Where does printf come from?
  - MAGIC!!!

# Compilation vs. Interpretation

## C (compiled)

- Compiler + Linker translates code into machine code.
- Machine code can be directly loaded by the OS and executed by the hardware. Fast!!
- New hardware targets require recompilation in order to execute on those new systems.

## Python (interpreted)

- Interpreter is written in some language (e.g. C) that is itself translated into machine code.
- The Python source code is then executed as it is read by the interpreter. Usually slower.
- Very portable! No reliance on hardware beyond the interpreter.



# Compilation vs. Virtual Targets (bytecode)

- Java translates source to a “byte code” which is a made-up architecture, but it resembles machine code somewhat.
- Technically, architectures *could* execute this byte code directly.
  - But these were never successful or practical.
- Instead, a type of virtual machine simulates that pseudo-architecture. (interpretation)
  - Periodically, the fake byte code is translated into machine code.
  - This is a type of delayed compilation! Just-In-Time (JIT) compilation.
- This is a compromise to either approach.
  - Surprisingly very competitive in speed.
  - I don't think the JVM-style JIT is going away any time soon.
  - Check: Just In Time (JIT) Compilers - Computerphile
    - [https://www.youtube.com/watch?v=d7KHAVaX\\_Rs](https://www.youtube.com/watch?v=d7KHAVaX_Rs)

# The C Pre-Processor

- The C language is incredibly simplistic.
- To add some constrained complexity, there is a macro language.
  - This code does not get translated to machine code, but to more code!

```
#include "hello.h" // Just dumps the local file to this spot.
```

```
#include <stdio.h> // Same thing, but from a system path.
```

```
#define DEBUG 0 // Just a simple text replace
```

```
#if ( DEBUG ) // Conditionally compiles certain code
```

```
...
```

```
#else
```

```
...
```

```
#endif
```

# The C Pre-Processor

- It can also replace simple expressions with complex!

```
#include "hello.h" // Just dumps the local file to this spot.
```

```
#include <stdio.h> // Same thing, but from a system path.
```

```
#define ADD(x,y) x+y // Just a simple text replace  
// Which can be a bit problematic :)
```

```
int a = 2, b = 5;
```

```
printf("%d + %d = %d\n", a, b, ADD(a,b))
```

```
printf("(%d + %d)*2 = %d\n", a, b, ADD(a,b)*2)
```

# The C Pre-Processor

- It can also replace simple expressions with complex!

```
#include "hello.h" // Just dumps the local file to this spot.
```

```
#include <stdio.h> // Same thing, but from a system path.
```

```
#define ADD(x,y) ((x)+(y)) // Just a simple text replace  
                        // What's with the parentheses?
```

```
int a = 2, b = 5;
```

```
printf("%d + %d = %d\n", a, b, ADD(a,b))
```

```
printf("(%d + %d)*2 = %d\n", a, b, ADD(a,b)*2)
```

# THE C SYNTAX: CONTROL FLOW

Once you C the program, you can B the program.

# Controlling the flow: an intro to spaghetti

```
int main(void) {  
    int a = 5, b = -3;  
  
    if (a >= 5) {    // A traditional Boolean expression  
        printf("A\n");  
    }  
    else // No need for { } with a single statement  
        printf("B\n");  
        printf("Always happens!\n") // <-- Why { } are good  
  
    return 0;  
}
```

# Controlling the flow: Boolean Expressions

- C does not have a Boolean type!
  - However, the C99 and newer standard library provides one in `<stdbool.h>`
- The Boolean expressions are actually just an `int` type.
  - It is just the general, default type. Weird but consistent, yet again!

```
int a = 5, b = -3, result;
```

```
result = a <= b; // 0 when false, non-0 when true
```

```
result = a > b; // typical comparisons: >=, <=, >, <
```

```
result = a == b; // like Java, equality is two equals
```

```
result = a != b; // inequality, again, works like Java
```

# Controlling the flow: Putting it Together

- `if` statements therefore take an `int` and not a Boolean, as an expression.
  - If the expression is `0` it is considered false.
  - Otherwise, it is considered true.

```
if (0) {    // Always false
    printf("Never happens.\n");
}
```

```
if (-64) { // Always true
    printf("Always happens.\n");
}
```



# Throwing us all for a loop

- Most loops (while, do) work exactly like Java.
  - Except, of course, the expressions are `int` typed, like `if` statements.
- For loops only come in the traditional variety:
  - `for` (initialization; loop invariant; update statement)
  - C89 does not allow variable declaration within:
    - **ERROR:** `for (int i = 0; i < 10; i++) ...`
  - However, C99 and newer does allow this. Please do it.
- Loops have special statements that alter the flow:
  - `continue` will end the current iteration and start the next.
  - `break` will exit the loop entirely.

# Loop Refresher: While, Do-While, For Loops

```
int main(void) {  
    int i = 0;  
    while (i < 10) {          // Each loop here is equivalent  
        i++;  
    }  
  
    i = 0;  
    do {                      // Do loops guarantee one invocation  
        i++;  
    } while (i < 10);        // Note the semi-colon!  
  
    for (i = 0; i < 10; i++) {  
    }  
  
    return 0;  
}
```

# Taking a break and switching it up

- The `switch` statement requires proper placement of `break` to work properly.
  - Starts at `case` matching expression and follows until it sees a `break` .
  - It will “fall through” other `case` statements if there is no `break` between them.

```
switch (character) {  
    case '+': ... // Falls through (acts as '-' as well)  
    case '-': ... break;  
    case '*': ... break;  
    default: ... break; // When does not match any case  
} // Note: unlike Java, cannot match strings!!
```

- Sometimes fall through is used on purpose... but it's a bug 99% of the time :/

# Control Flow: Summary

Note: a *statement* can be a { block }

- Conditional Blocks:

- `if (expression) statement`
- `if (expression) statement`  
`else statement`

- The if statement can be chained:

```
if (expression) statement
else if (expression) statement
else statement
```

- Conventional Loops:

- `while (expression) statement`
- `do`  
`statement`  
`while (expression);`

# Control Flow: Summary

Note: a *statement* can be a { block }

- For Loops:

- `for` (statement; expression; statement) *statement*
- `continue`; // Skip to end of loop body
- `break`; // Exit loop regardless of state of the loop invariant

- Switch:

- `switch` (expression) {  
    *case* const1: *statements*  
    *case* const2: *statements*  
    *default*: *statements*  
}
- `break`; // Exit switch body (don't fall through)

# What's your function?

```
int number_of_people(void) {  
    return 3;  
}
```

```
void news(void) {  
    printf("no news");  
}
```

```
int sum(int x, int y) {  
    return x + y;  
}
```

- Familiar: Java is, once again, C-like
- You declare the return type before the name.
  - `void` is used when there is nothing returned
  - It is also used to **explicitly** denote there being no arguments.
  - You **SHOULD** specify `void` instead of having an empty list.
- Functions must be declared before they can be used.
  - We will look at how we divide functions up between files soon!

# This is all the structure you get, kid

- C gives us a very simple method of defining *aggregate data types*.
- The struct keyword can combine several data types together:

```
struct Song {  
    int lengthInSeconds;  
    int yearRecorded;  
}; // Note the semi-colon!
```

// You can declare a Song variable like so:

```
struct Song my_song;  
my_song.lengthInSeconds = 512;
```

# I don't like all that typing... So I'll... typedef it

- To avoid typing the full name “struct Song” we can create a Song type instead.
- The typedef keyword defines new types.

```
typedef struct {  
    int lengthInSeconds;  
    int yearRecorded;  
} Song; // Note Song is now written afterward!
```

// You can declare a Song variable like so:

```
Song my_song;  
my_song.lengthInSeconds = 512;
```



# I don't like all that typing... So I'll... typedef it

- You can also do this with integer types, for instance to define bool:

```
typedef int bool;
```

- And `enum` types, although it won't complain if you mix/match them:

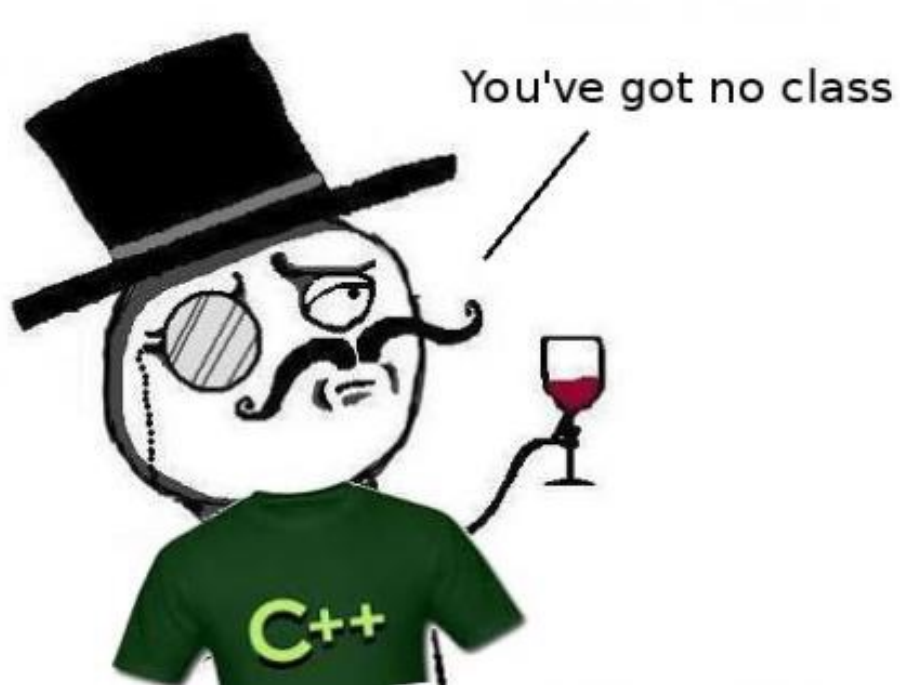
```
typedef enum { CS445, CS447, CS449 } Course;
```

- Now, functions can better illustrate they take an enum value:
  - Though, it accepts any integer and, yikes, any enum value without complaint!

```
void print_course(Course course) {  
    switch (course) {  
        case CS449: printf("The best course: CS449!\n");  
    }  
}
```

# That's seriously all you get...

- Unlike Java, C is not Object-Oriented and has no class instantiation.
- That's C++!



# Garbage in, garbage out: initialization

- As we saw earlier, variables don't require initialization.
- However, unlike Java, the variables do not have a default value.
  - Java will initialize integers to 0 if you do not specify.
  - C, on the other hand...
- The default values for variables are undefined.
  - They could be anything.
  - The Operating System ultimately decides.
    - Generally, whatever memory is left over. Also known as “garbage.”
- **ALWAYS INITIALIZE YOUR VARIABLES**

The slide features a white background with two dark blue diagonal bands. The top band runs from the left edge to the top right, and the bottom band runs from the bottom left to the right edge. Both bands contain a repeating pattern of binary code (0s and 1s) in a lighter blue color.

# SCOPE AND LIFETIME

# Scope and lifetime – what are they?

- Scope: Refers to the visibility of a symbol
  - Symbol: the name of something stored in memory
  - E.g.: variable, function
- When you compile, the compiler matches names with memory locations!

```
int a; // this variable is visible globally.
```

```
void func() {
```

```
    int b; // this variable is visible inside the function
```

```
    b = a % 2; // I can use variables a and b
```

```
}
```

```
void foreshadowing() {
```

```
    int a = 3; // Variable a is now a different thing!
```

```
    int c = a; // I can use variable a but the innermost
```

```
    c += b;    // Using variable b here is erroneous.
```

```
}
```

# Scopes in C

- In C, scopes are defined by files and blocks

main.c

```
#include <stdio.h>
```

```
// anything here is in the scope of the file!
```

```
int func() { // this brace starts a block!
```

```
    for( int i=0 ; i<10 ; i++ )
```

```
    { // this brace starts a block!
```

```
    } // this brace ends a block!
```

```
    // more code
```

```
    { // this brace starts a block!
```

```
    } // this brace ends a block!
```

```
}
```


# Global scopes

- Globals can be different



**External linkage** – global across the program

```
int a; // when does this variable become valid?  
      // when does it stop being valid?
```



**Internal linkage** – within this C file!

```
static int b; // when does this variable become valid?  
             // when does it stop being valid?
```

# Scope and lifetime – what are they?

- Lifetime: Refers to the validity of a variable
  - When is the memory allocated?
  - When does it become invalid?

Global variables are allocated at compile-time!  
They are static (as in unmovable)!  
Can be used at any point of your program

```
int a; // when does this variable become valid?  
      // when does it stop being valid?
```

These are called automatic variables

```
void func() {  
    int b; // when does this variable become valid?  
           // when does it stop being valid?  
}
```

Local variables are allocated every time the function is called!  
They are dynamic  
They are “destroyed” once the function returns

But are they  
really?



# What is the output of this code?



```
#include <stdio.h>
```

```
int a = 42;
```

```
void func() {  
    printf("This function is being called!\n");  
    int i = 0;  
    printf("\tThe value of variable i is %d\n", i);  
    printf("\tThe value of variable a is %d\n", a);  
}
```

```
This function is being called!  
The value of variable i is 0  
The value of variable a is 42
```

[Check the code](#)

# What is the output of this code?



```
#include <stdio.h>

int a = 42;

void func() {
    printf("This function is being called!\n");
    int i = 0;
    printf("The value of variable i is %d\n", i);
    printf("The value of variable a is %d\n", a);
    i++;
    printf("The value of variable i is %d\n", i);
    int a = 12;
    printf("The value of variable a is %d\n", a);
}
```

[Check the code](#)

```
This function is being called!
The value of variable i is 0
The value of variable a is 42
The value of variable i is 1
The value of variable a is 12
```

# What is the output of this code?

```
#include <stdio.h>
```

```
int a = 42;
```

```
void func() {  
    printf("This function is being called!\n");  
    int i = 0;  
    printf("The value of variable i is %d\n", i);  
    printf("The value of variable a is %d\n", a);  
    {  
        i++;  
        int a = 12;  
    }  
    printf("The value of variable i is %d\n", i);  
    printf("The value of variable a is %d\n", a);  
}
```



[Check the code](#)

```
This function is being called!  
The value of variable i is 0  
The value of variable a is 42  
The value of variable i is 1  
The value of variable a is 42
```

# What if local variables existed from the beginning?

- Can't the compiler figure out how much memory it needs?
  - Why aren't all local variables static?
  - Why do they have to be created destroyed??

```
void func() {  
    auto int b; // when does this variable become valid?  
               // when does it stop being valid?  
    func();  
}
```

# But we can make them!

- Can't the compiler figure out how much memory it needs?
  - Why aren't all local variables static?
  - Why do they have to be created destroyed??

```
void func() {  
    static int b; // now it survives across calls!  
  
    func();  
}
```

# What is the output of this code?

```
#include <stdio.h>
```

```
void func() {  
    printf("This function is being called!\n");  
    int i = 0;  
    printf("The value of variable i is %d\n", i);  
  
    i++;  
  
    printf("The value of variable i is %d\n", i);  
}
```



```
This function is being called!  
The value of variable i is 0  
The value of variable i is 1
```

```
This function is being called!  
The value of variable i is 0  
The value of variable i is 1
```

# What is the output of this code?

```
#include <stdio.h>
```

```
void func() {  
    printf("This function is being called!\n");  
    static int i = 0;  
    printf("The value of variable i is %d\n", i);  
  
    i++;  
  
    printf("The value of variable i is %d\n", i);  
}
```

[Check the code](#)



```
This function is being called!  
The value of variable i is 0  
The value of variable i is 1
```

```
This function is being called!  
The value of variable i is 1  
The value of variable i is 2
```

# You hear a whisper... static;

- Remember the `static` keyword?
  - Java uses it to make class variables/functions.
  - This is going to become tricky really fast 😊
- You can declare `static` functions and global variables in C
  - It simply affects the ability of other files to use them
  - Since they are already “static” because their size is calculated at compile-time
- This is useful for avoiding ***name collisions***, when two functions have the same name.
  - This normally would make using multiple files and other people’s code troublesome.
  - Using `static` helps because it will not pollute the name space.



# Controlled the symbols

- We'll investigate the impact of using `static` when we discuss Linking
  - The last step when you "compile" your code.

```
C(gcc -c speak-static.c)
```

```
#include <stdio.h>
```

```
static void only be referenced in this file.
```

```
speak_number(int n) {  
    printf("Number! %d\n", n);  
}
```

```
int main(void) {  
    speak_number(42);  
    return 0;  
}
```

# extern; Importing symbols

- The other side of the coin is the `extern` keyword.
- This tells the linker that it should expect the symbol to be found elsewhere.

**C** (speak.c)

```
#include <stdio.h>
```

↙ **This symbol is... somewhere.**

```
extern int number;
```

```
// number is explicitly extern
```

```
void speak_number(int n) {  
    printf("Number! %d %d\n", n, number);  
}
```

**C** (main.c)

↙ **Here it is!!**

```
int number = 2;
```

```
// Declare the function (implicitly extern)
```

```
void speak_number(int n);
```

```
int main(void) {  
    speak_number(42);  
    return 0;  
}
```

# Final thoughts of global variables

- You should always avoid global variables.
- However, if you are using them, make sure to liberally use `static`
  - This will stop the names of variables from polluting the name space.
  - The use of `extern` is likely indicating a poor design.
- This is also true for functions, too.
  - Generally declare them `static` unless you need them from within another file.
  - Helps make it clear what functions are important and which can be deleted or refactored.
  - (Much like private functions in classes)
- To avoid surprises, always initialize your global variables!



# Scope and lifetime

**OMET:** I WANT MORE TABLES!

**Luis:** Ok!

Scope/Lifetime	Example	Equivalent to
Scope: Function Lifetime: Function	<pre>int f() {     int x; }</pre>	<pre>int f() {     auto int x; }</pre>
Scope: Global Lifetime: Program	<pre>int x; int f() {  }</pre>	
Scope: File Lifetime: Program	<pre>static int x; static int f() {  }</pre>	
Scope: Function Lifetime: Program	<pre>int f() {     static int x; }</pre>	

You'll prob. never see  
"auto" in C code!

# The trouble is stacking up on us!

```
#include <stdio.h>    // Gives us 'printf'
#include <stdlib.h>    // Gives us 'rand' which returns a random-ish int

void undefined_local() {
    int x;
    printf("x = %d\n", x);
}

void some_calc(int a) {
    a = a % 2 ? 1 : -a;
}

int main(void) {
    for (int i = 0; i < 5; i++) {
        some_calc(i * i);
        undefined_local();
    }
    return 0;
}
```

Output:

```
x = 0
x = 1
x = -4
x = 1
x = -16
```



Q: Hmm. Where is the value for 'x' coming from? Why?

# Where's that data coming from??

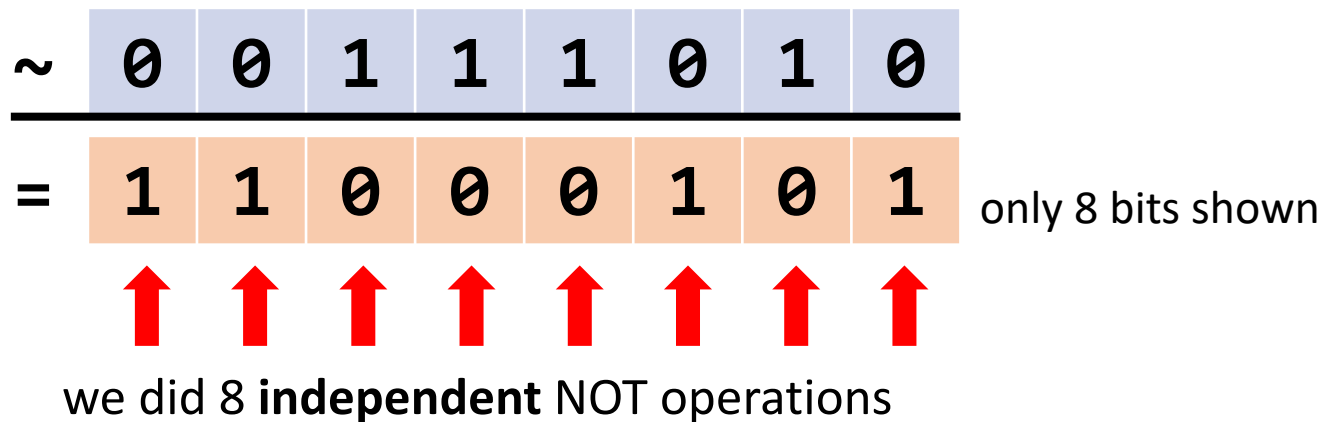
- Every variable and data in your program technically has a location in which it lives.
- In the previous nonsense example, the “x” variable was sharing the same space as the “a” variable from the other function.
  - The section of incremental memory called the stack, in this specific case.
  - This is not defined behavior of the language, but rather the OS.
- C does not impose many rules on how memory is laid out and used.
  - In fact, it gets right out of the way and lets you fall flat on your face.
- Now, we will take a deeper dive into... **MEMORY**

# BITWISE MANIPULATION

Small review – If I have time!

# Applying NOT to a whole bunch of bits

- If we use the **not** instruction ( $\sim$  in C), this is what happens:

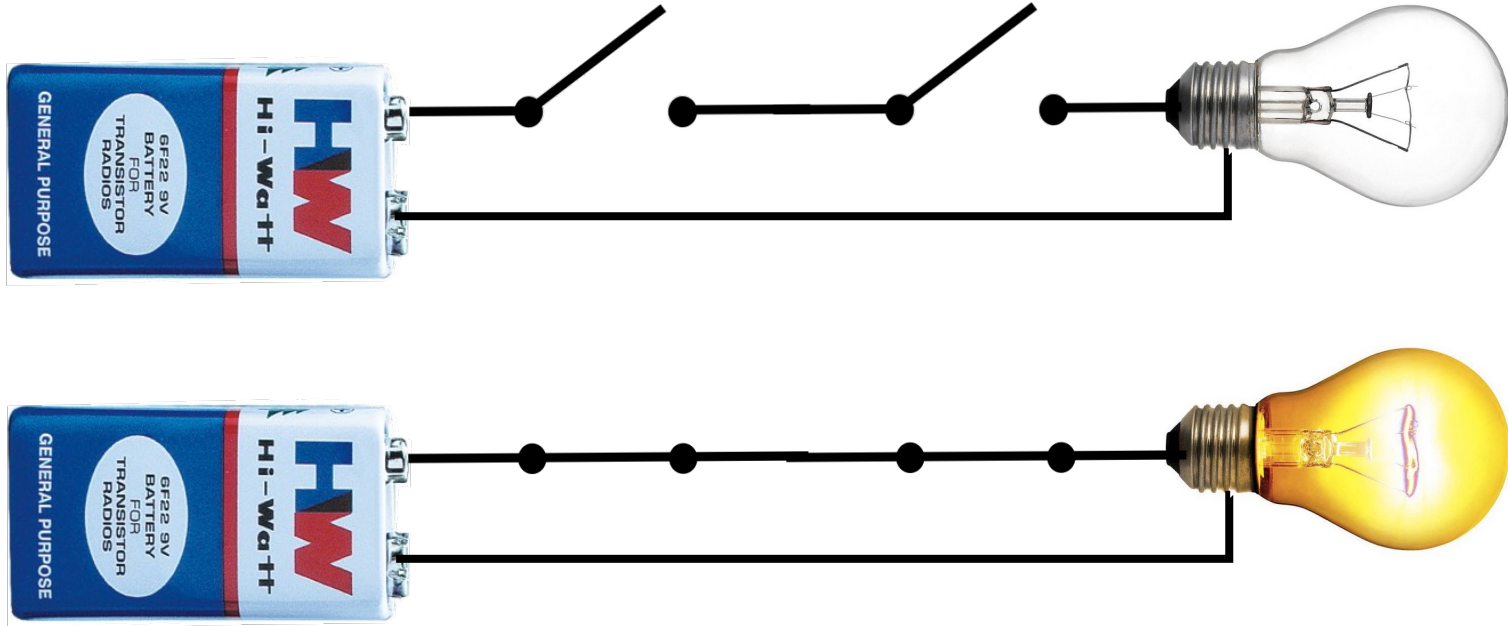


That's it.



# Let's add some switches


- There are two switches in a row connecting the light to the battery.
- **How do we make it light up?**



# AND (Logical product)

- AND is a **binary (two-operand) operation**.
- It can be written a number of ways:  
**A&B    A $\wedge$ B    A·B    AB**
- If we use the **and** instruction (& in C):

	1	1	1	1	0	0	0	0
&	0	0	1	1	1	0	1	0
=	0	0	1	1	0	0	0	0



we did 8 **independent** AND operations

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

# AND (Logical product)

Great to reset bits:  
i.e., make them zero.

	1	1	1	1	0	0	0	0
&	0	0	1	1	1	0		
=	0	0						

↑ ↑

we did 8 independent AND operations

Why?

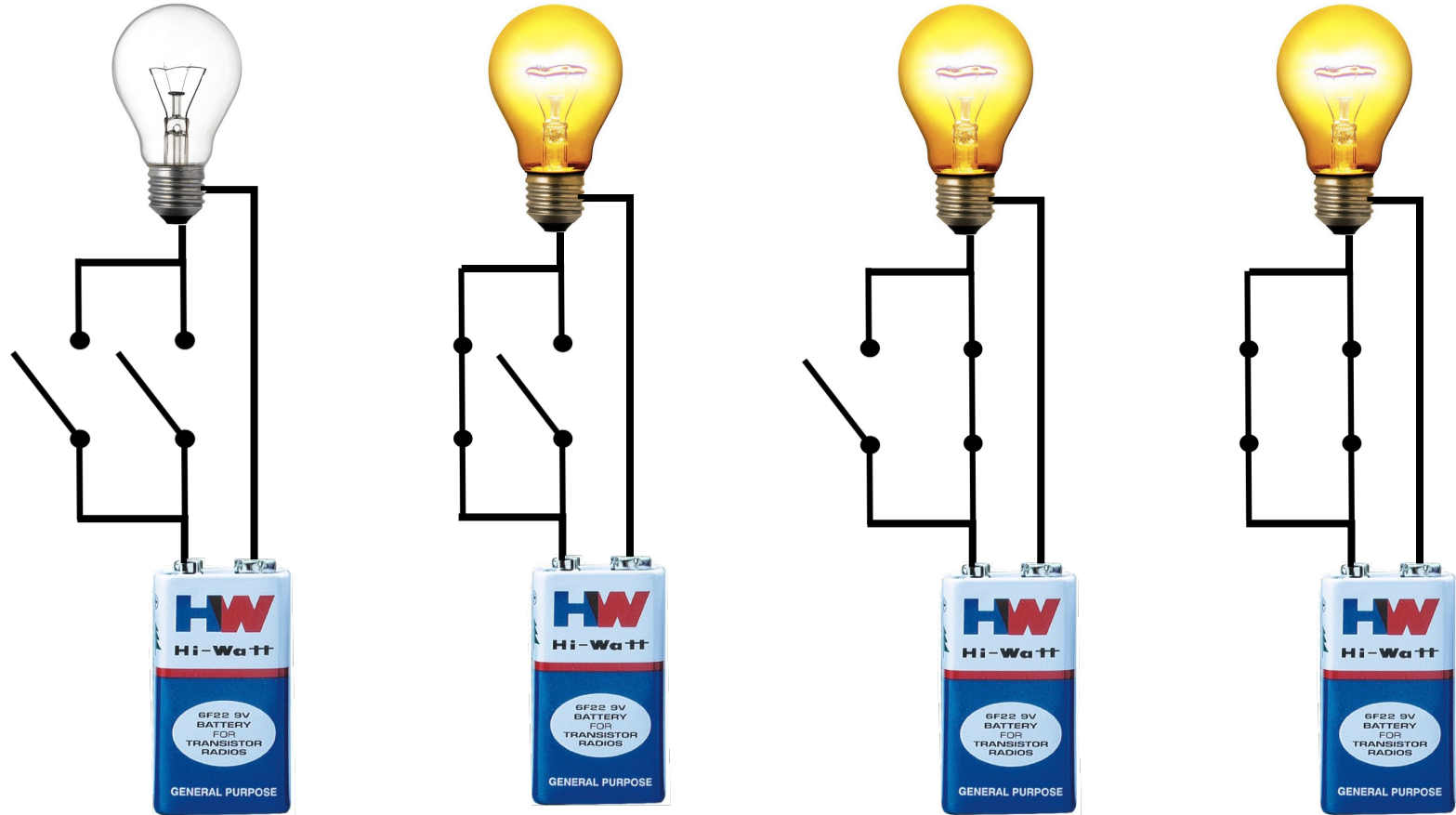
$x \& 0 = 0 \rightarrow$  Forces zero

$x \& 1 = x \rightarrow$  Remains unchanged

A	B	Q
0	0	0
0	1	0
1	0	0

# "Switching" things up


- NOW how can we make it light up?



# OR (Logical sum...?)

- we might say "**and/or**" in English
- it can be written a number of ways:
  - **A|B    A∨B    A+B**
- if we use the **or** instruction (or **|** in C/Java):

	1	1	1	1	0	0	0	0
	0	0	1	1	1	0	1	0
=	1	1	1	1	1	0	1	0



We did 8 **independent** OR operations.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

# OR (Logical sum...?)

- w
  - it
  - if
- Great to set bits:  
i.e., make them one.

	1	1	1	1	0	0	0	0
	0	0	1	1	1	0		
=	1	1						

We did 8 independent OR operations.

Why?

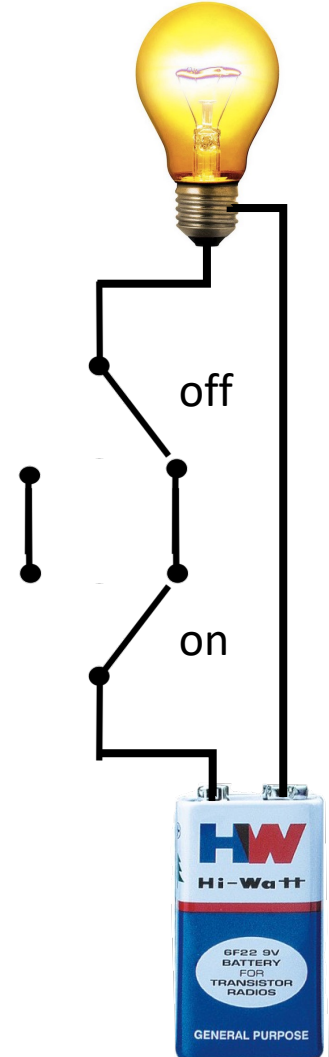
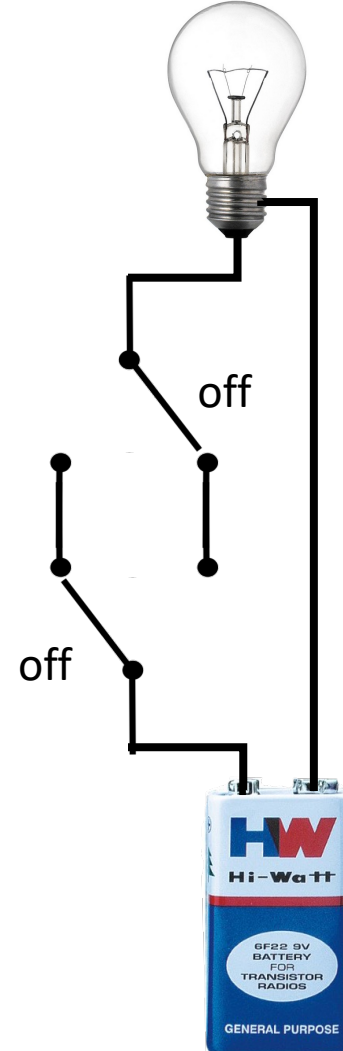
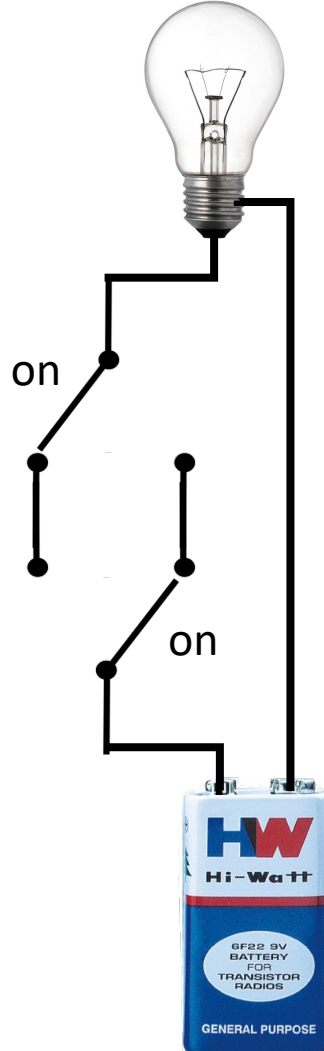
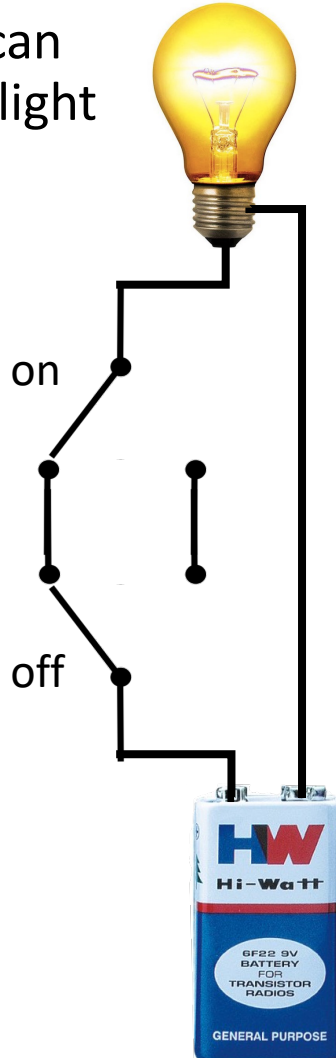
$x | 1 = 1 \rightarrow$  Forces one

$x | 0 = x \rightarrow$  Remains unchanged

A	B	Q
0	0	0
0	1	1
1	0	1

# 3-way switching

- NOW how can we make it light up?




# XOR ("Logical" difference?)

- We might say "**or**" in English.
- It can be written a number of ways:

$$A \wedge B \quad A \oplus B$$

- If we use the **xor** instruction (^ in C):

	1	1	1	1	0	0	0	0
^	0	0	1	1	1	0	1	0
=	1	1	0	0	1	0	1	0



We did 8 **independent** XOR operations.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0



# XOR ("Logical" difference?)

Great to capture the different bits:

	1	1	1	1	0	0	0	0
^	0	0	1	1	1	0		
=	1	1	0	0	1	0		

↑ ↑  
We did 8 in

Why?

$1^0 = 0^1 = 1 \rightarrow$  Is different  
 $1^1 = 0^0 = 0 \rightarrow$  Is not different

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

# XOR ("Logical" difference?)

Great to flip bits:

	1	1	1	1	0	0	0	0
$\wedge$	0	0	1	1	1	0		
$=$	1	1	0	0	1	0		

↑ ↑

We did 8 in

Why?

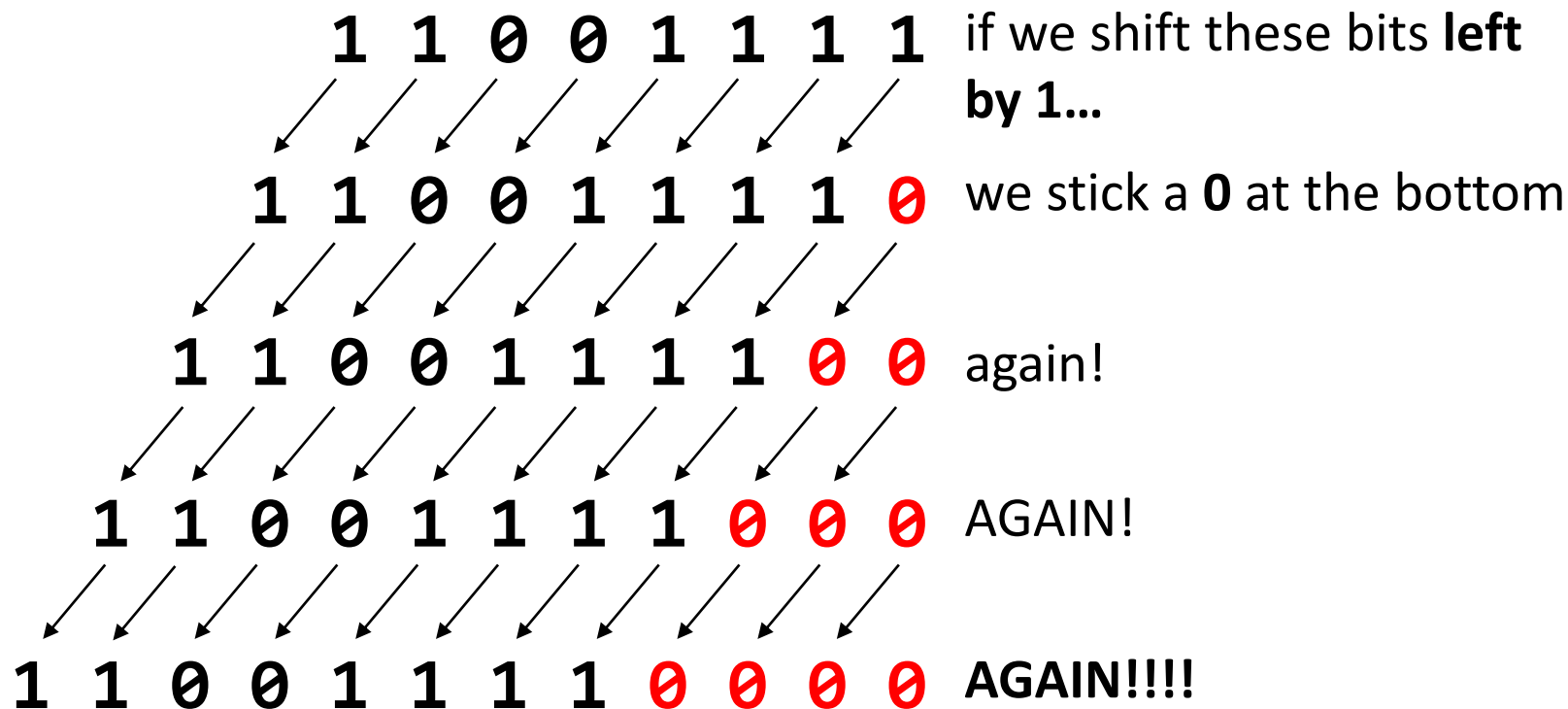
$x \wedge 0 = x \rightarrow$  Doesn't change

$x \wedge 1 = \sim x \rightarrow$  Flip!

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

# Bit shifting

- Moving bits around.



- C (and Java) use the `<<` operator for left shift

**Eg.  $10 = 5 \ll 1 \rightarrow 00001010 = 00000101 \ll 1$**

- **`B = A << 4;`** *// B = A shifted left 4 bits*

If the bottom 4 bits of the result are now 0s...

- ...what happened to the *top* 4 bits?

**0011 0000 0000 1111 1100 1101 1100 1111**

the bit bucket is not a real place

it's a programmer joke ok

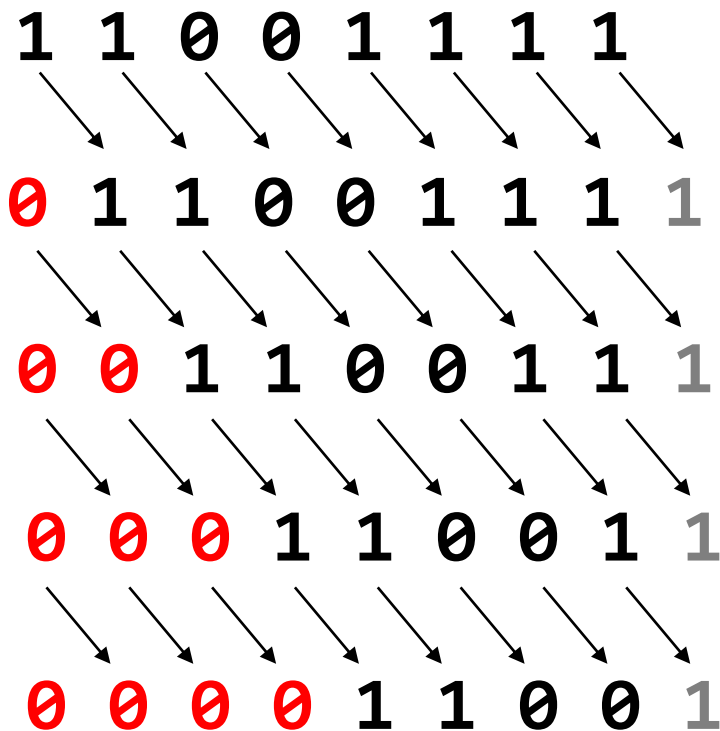
in the UK they might say the “Bit Bin”

bc that’s their word for trash



# Shift Right (Logical)

- We can **shift right**, too (srl in MIPS)



if we shift these bits **right**  
**by 1...**

we stick a **0** at the top

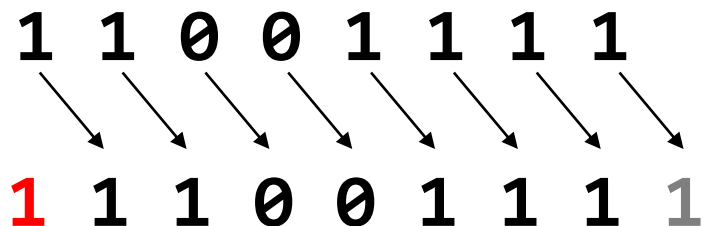
again!

AGAIN!

**Wait... what if this was a  
negative number?**

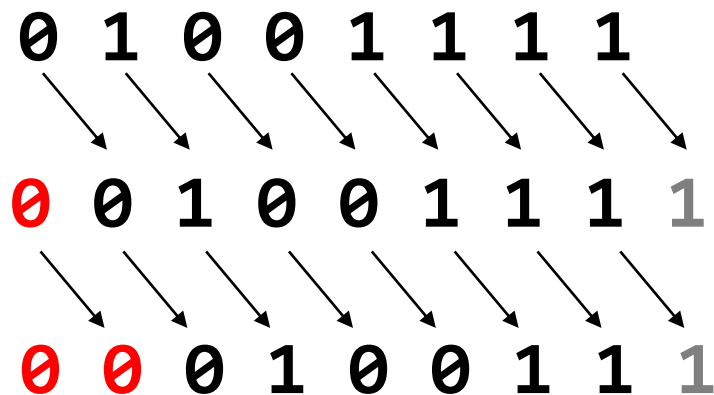
# Shift Right (Arithmetic)

- We can **shift right with sign-extension, too** (MIPS: sra)



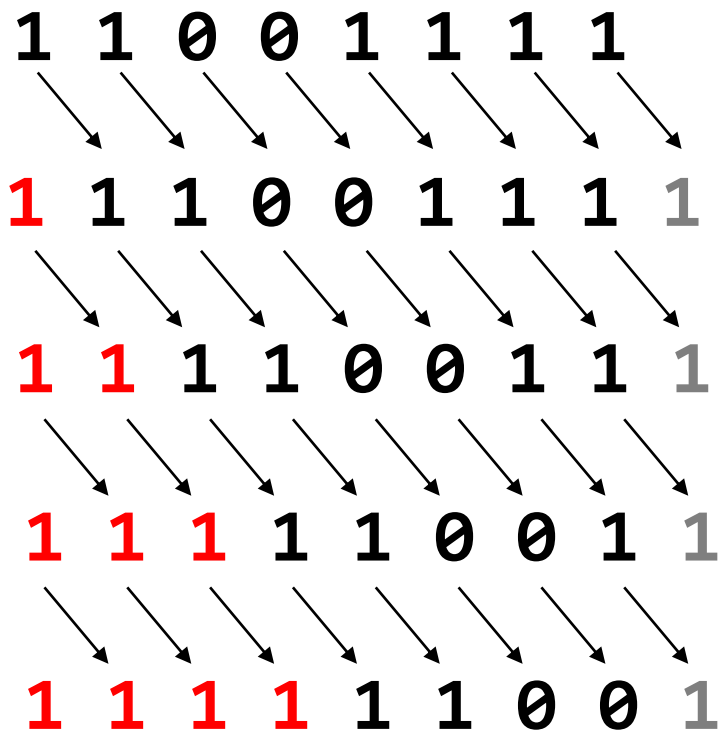
if we shift these bits **right**  
**by 1...**

we copy the **1** at the top (or 0,  
if MSB was a 0)



# Shift Right (Arithmetic)

- We can shift right with sign-extension, too (MIPS: sra)



if we shift these bits **right**  
**by 1...**

we copy the **1** at the top (or 0,  
if MSB was a 0)

again!

AGAIN!

**AGAIN!!!!!!** (It's still  
negative!)

# C Bitwise Operations: Summary

C code	Description	MIPS instruction
<code>x   y</code>	or	<code>or x, x, y</code>
<code>x &amp; y</code>	and	<code>and x, x, y</code>
<code>x ^ y</code>	xor	<code>xor x, x, y</code>
<code>~x</code>	complement (negate)	<code>nor x, x, \$0</code> (“not”)
<code>x &lt;&lt; y</code>	left-shift logical	<code>sll x, x, y</code>
<code>x &gt;&gt; y</code>	right-shift logical	<code>srl x, x, y</code>

When x is signed (most of the time...):

<code>x &gt;&gt; y</code>	right-shift arithmetic	<code>sra x, x, y</code>
---------------------------	------------------------	--------------------------



# FIELDS

# This is all the structure you get, kid

- C gives us a very simple method of defining *aggregate data types*.
- The struct keyword can combine several data types together:

```
struct Color {  
    int red;  
    int green;  
    int blue;  
}; // Note the semi-colon!
```

// You can declare a Color variable like so:

```
struct Color my_color;  
my_color.red = 23;
```

# I don't like all that typing... So I'll... typedef it

- To avoid typing the full name “struct Song” we can create a Song type instead.
- The typedef keyword defines new types.

```
typedef struct {  
    int red;  
    int green;  
    int blue;  
} Color; // Note Color is now written afterward!
```

```
// You can declare a Color variable like so:  
Color my_color;  
my_color.red = 23;
```

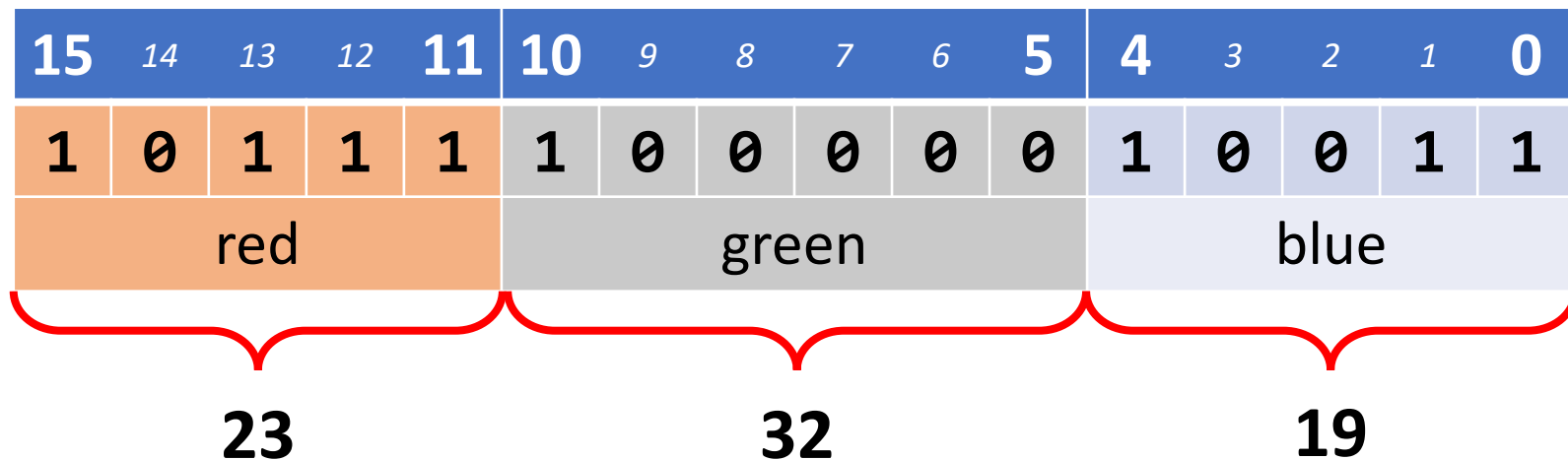
# What's the sizeof?

- Small exercise...
  - Pseudo-write the code to print how much memory struct Color takes in memory!
  - I haven't done it yet :D
    - We can all do it !!

SMALLER IS (SOMETIMES) BETTER

# The masters of meaning

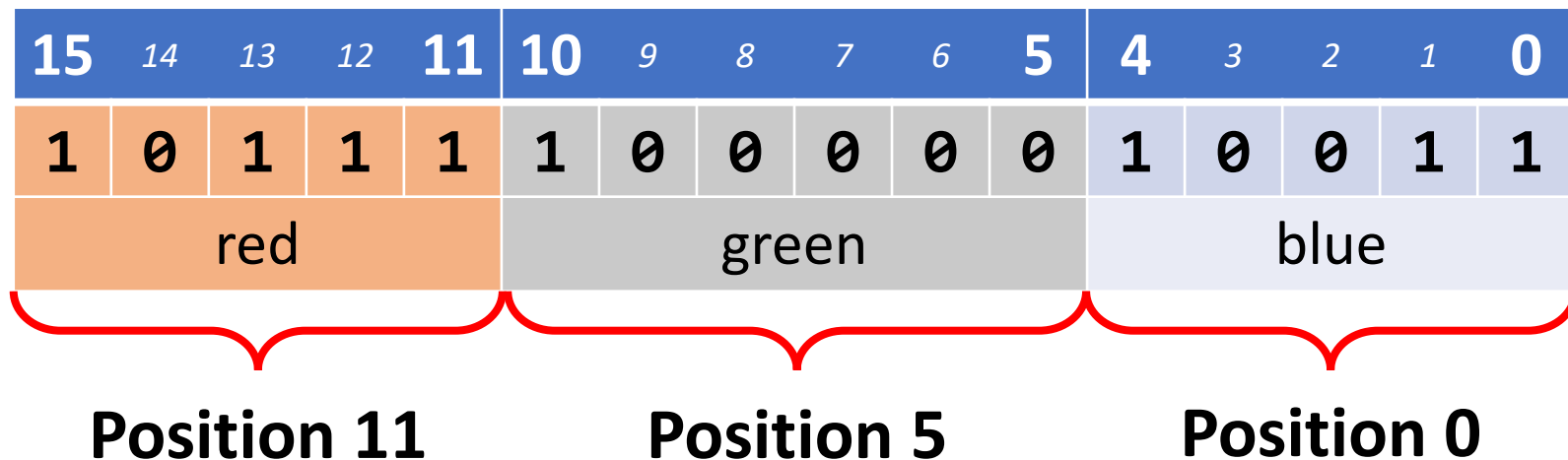
- well what if we wanted to store multiple *integers* in one value?



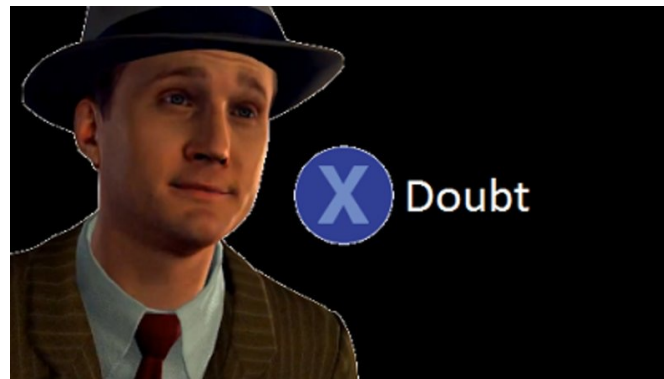
That's this color, in RGB565.

# Field extraction

- This bitfield has 3 fields: red, green, and blue

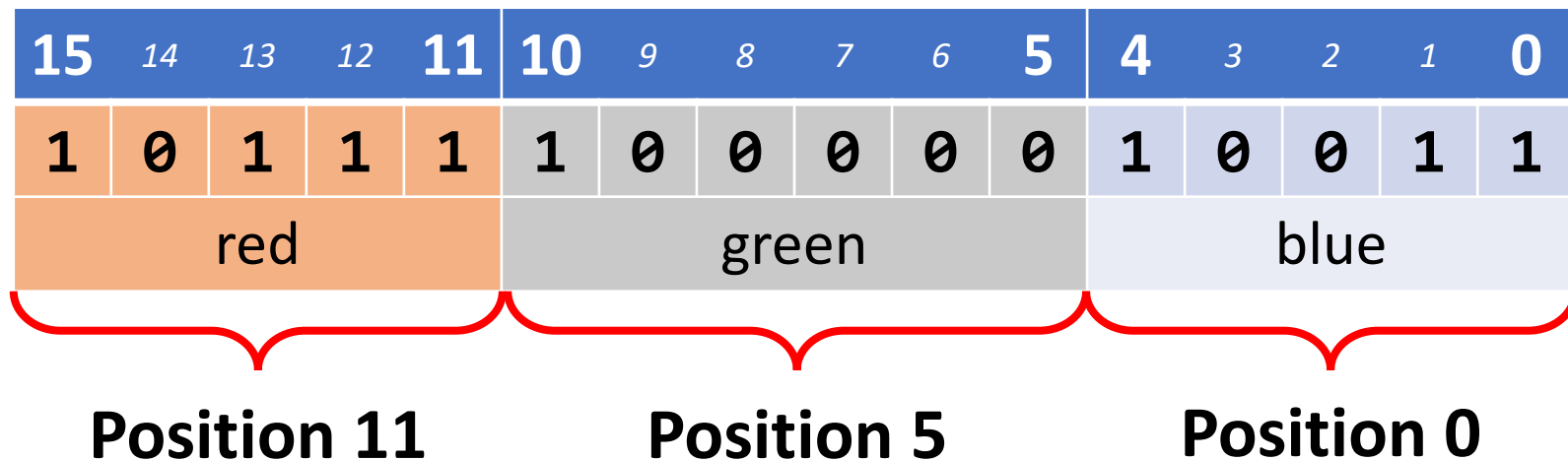


```
red = color >> 11;  
green = color >> 5;  
blue = color >> 0;
```



# Field extraction

- This bitfield has 3 fields: red, green, and blue



**red** = **color** >> 11;    ➔ 000000000000010111

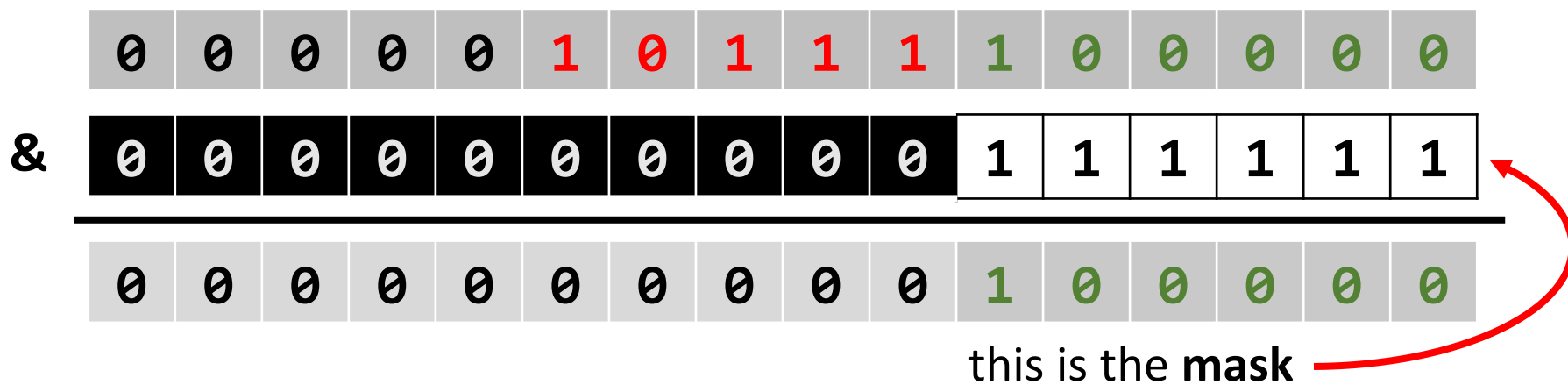
**green** = **color** >> 5;    ➔ 0000010111100000

**blue** = **color** >> 0;    ➔ 1011110000010011



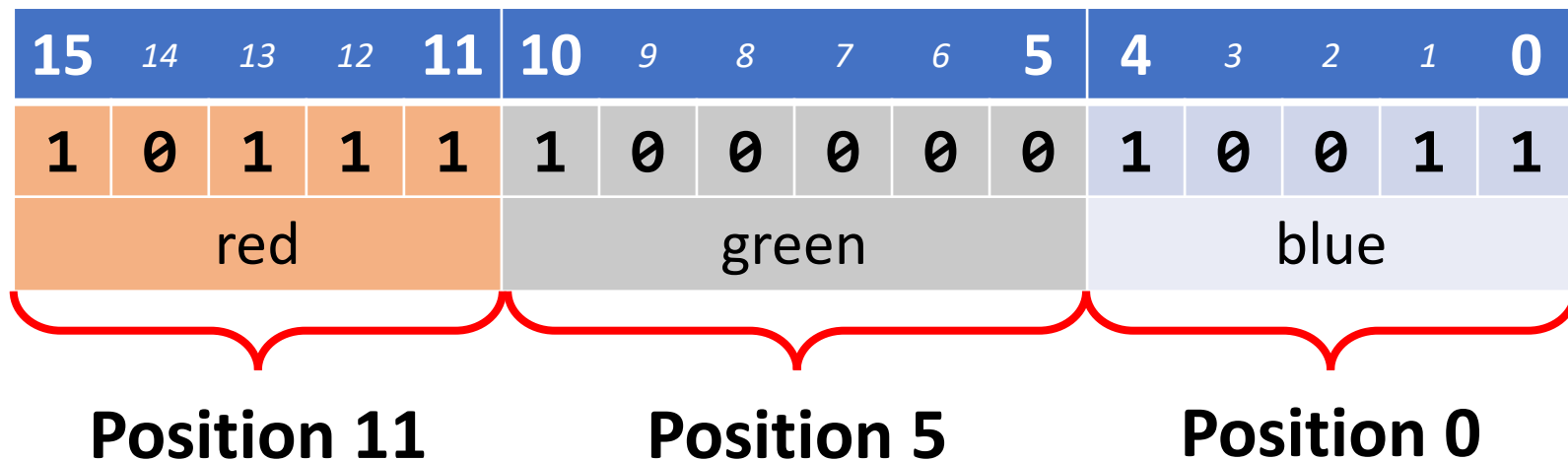
# Masquerade

- we need to **get rid of (zero out)** the bits that we don't care about
- a **mask** is a **specialty-constructed value** that has:
  - 1s in the bits that we want to keep
  - 0s in the bits that we want to discard
- which bits do we want to keep? which do we want to discard?



# Field extraction

- This bitfield has 3 fields: red, green, and blue



**red** = (**color** >> 11) & 0x1F; → ...0010111

**green** = (**color** >> 5) & 0x3F; → ...100000

**blue** = (**color** >> 0) & 0x1F; → ...10011

# Checking notes

- Talk to people around you, and come up with one question

From Vinicius Petrucci's slides

Based on slides originally designed by Drs. Bryant and O'Hallaron, CMU

# OTHER EXAMPLES && APPLICATIONS

Some examples of using shift operators  
in combination with bitmasks  
(and some extra floating-point stuff!)

# Shifting Arithmetic?

- What are the following computing?
  - $x \gg n$ 
    - $0b\ 0100 \gg 1 = 0b\ 0010$
    - $0b\ 0100 \gg 2 = 0b\ 0001$
    - Divide by  $2^n$
  - $x \ll n$ 
    - $0b\ 0001 \ll 1 = 0b\ 0010$
    - $0b\ 0001 \ll 2 = 0b\ 0100$
    - Multiply by  $2^n$
- Shifting is faster than general multiply and divide operations

# Left Shifting Arithmetic 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation:  $x * 2^n$ ?

$x = 25;$	00011001	=	Signed Unsigned	25
$L1=x<<2;$	0001100100	=	100	100
$L2=x<<3;$	00011001000	=	-56	200
$L3=x<<4;$	000110010000	=	-112	144

signed overflow

unsigned overflow

# Right Shifting Arithmetic 8-bit Examples

- **Reminder:** C operator `>>` does *logical* shift on **unsigned** values and *arithmetic* shift on **signed** values
  - **Logical** Shift:  $x/2^n$ ?

`xu = 240u;`    `11110000`    = 240

`R1u=xu>>3;`    `00011110000`    = 30

`R2u=xu>>5;`    `0000011110000`    = 7

rounding (down)

# Right Shifting Arithmetic 8-bit Examples

- **Reminder:** C operator `>>` does *logical* shift on **unsigned** values and *arithmetic* shift on **signed** values
  - **Arithmetic** Shift:  $x/2^n$ ?

`xS = -16;`    11110000    = -16

`R1S=xu>>3;`    11111110000    = -2

`R2S=xu>>5;`    111111110000    = -1

rounding (down)



# Examples

- Extract 2nd most significant byte of an int
- Extract the sign bit of a signed int
- Conditionals as Boolean expressions

# Using Shifts and Masks

- Extract 2<sup>nd</sup> most significant *byte* of an `int`:

- First shift, then mask:  $(x \gg 16) \& 0xFF$

<b>x</b>	00000001 00000010 00000011 00000100
<b>x &gt;&gt; 16</b>	00000000 00000000 00000001 00000010
<b>0xFF</b>	00000000 00000000 00000000 11111111
<b>(x &gt;&gt; 16) &amp; 0xFF</b>	00000000 00000000 00000000 00000010

- Or first mask, then shift:  $(x \& 0xFF0000) \gg 16$

<b>x</b>	00000001 00000010 00000011 00000100
<b>0xFF0000</b>	00000000 11111111 00000000 00000000
<b>x &amp; 0xFF0000</b>	00000000 00000010 00000000 00000000
<b>(x &amp; 0xFF0000) &gt;&gt; 16</b>	00000000 00000000 00000000 00000010

# Using Shifts and Masks

- Extract the *sign bit* of a signed `int`:
  - First shift, then mask:  $(x \gg 31) \ \& \ 0x1$ 
    - Assuming arithmetic shift, but this works in either case
    - Need mask to clear 1s possibly shifted in

<b>x</b>	00000001 00000010 00000011 00000100
<b>x&gt;&gt;31</b>	00000000 00000000 00000000 00000000
<b>0x1</b>	00000000 00000000 00000000 00000001
<b>(x&gt;&gt;31) &amp; 0x1</b>	00000000 00000000 00000000 00000000

<b>x</b>	10000001 00000010 00000011 00000100
<b>x&gt;&gt;31</b>	11111111 11111111 11111111 11111111
<b>0x1</b>	00000000 00000000 00000000 00000001
<b>(x&gt;&gt;31) &amp; 0x1</b>	00000000 00000000 00000000 00000001

# Using Shifts and Masks

- Conditionals as Boolean expressions
  - For `int x`, what does `(x<<31)>>31` do?

<code>x=!!123</code>	00000000 00000000 00000000 000000001
<code>x&lt;&lt;31</code>	10000000 00000000 00000000 00000000
<code>(x&lt;&lt;31)&gt;&gt;31</code>	11111111 11111111 11111111 11111111
<code>!x</code>	00000000 00000000 00000000 00000000
<code>!x&lt;&lt;31</code>	00000000 00000000 00000000 00000000
<code>(!x&lt;&lt;31)&gt;&gt;31</code>	00000000 00000000 00000000 00000000

- Can use in place of conditional?