Introduction to C

4

CS/COE 0449
Introduction to
Systems Software

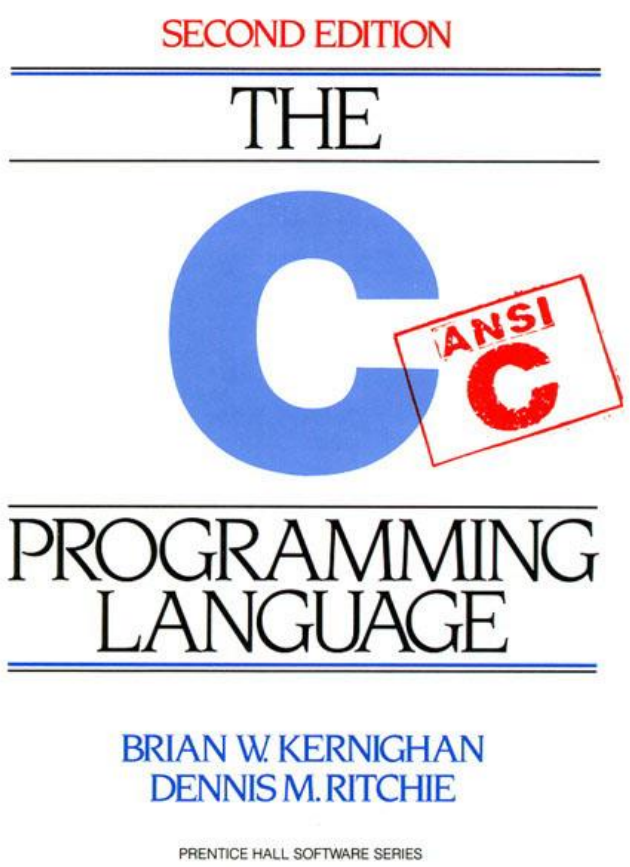
wilkie
(with content borrowed from Vinicius Petrucci
and Jarrett Billingsley)

Spring 2019/2020

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

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

- From IEEE Spectrum:
  - https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019
- Still relatively popular…
  - Lots of legacy code.
  - Lots of embedded devices.
  - Python, Java, R, JS are all written in C.

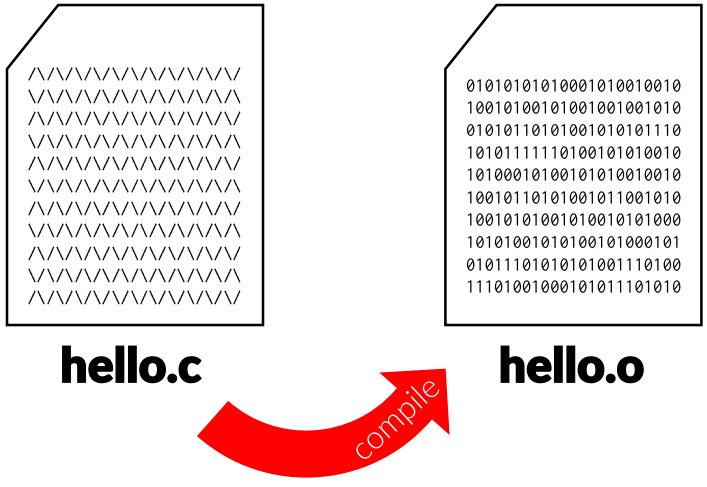
Compilation

- 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.

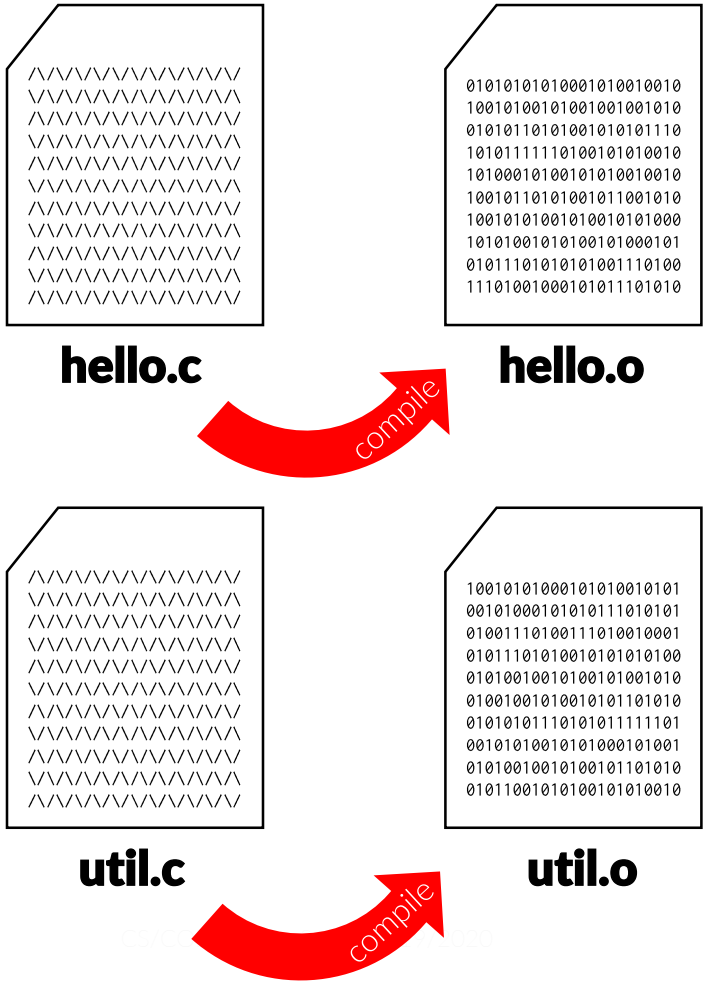
- A general C compiler will typically convert *.c source files into an intermediate *.o object file. Then, it will *link* these together to form an executable.
  - Assembly is also part of this process, but it is done behind the scenes.
  - You can have gcc (a common C compiler) spit out the assembly if you want!

Compilation: Simple Overview – Step 1



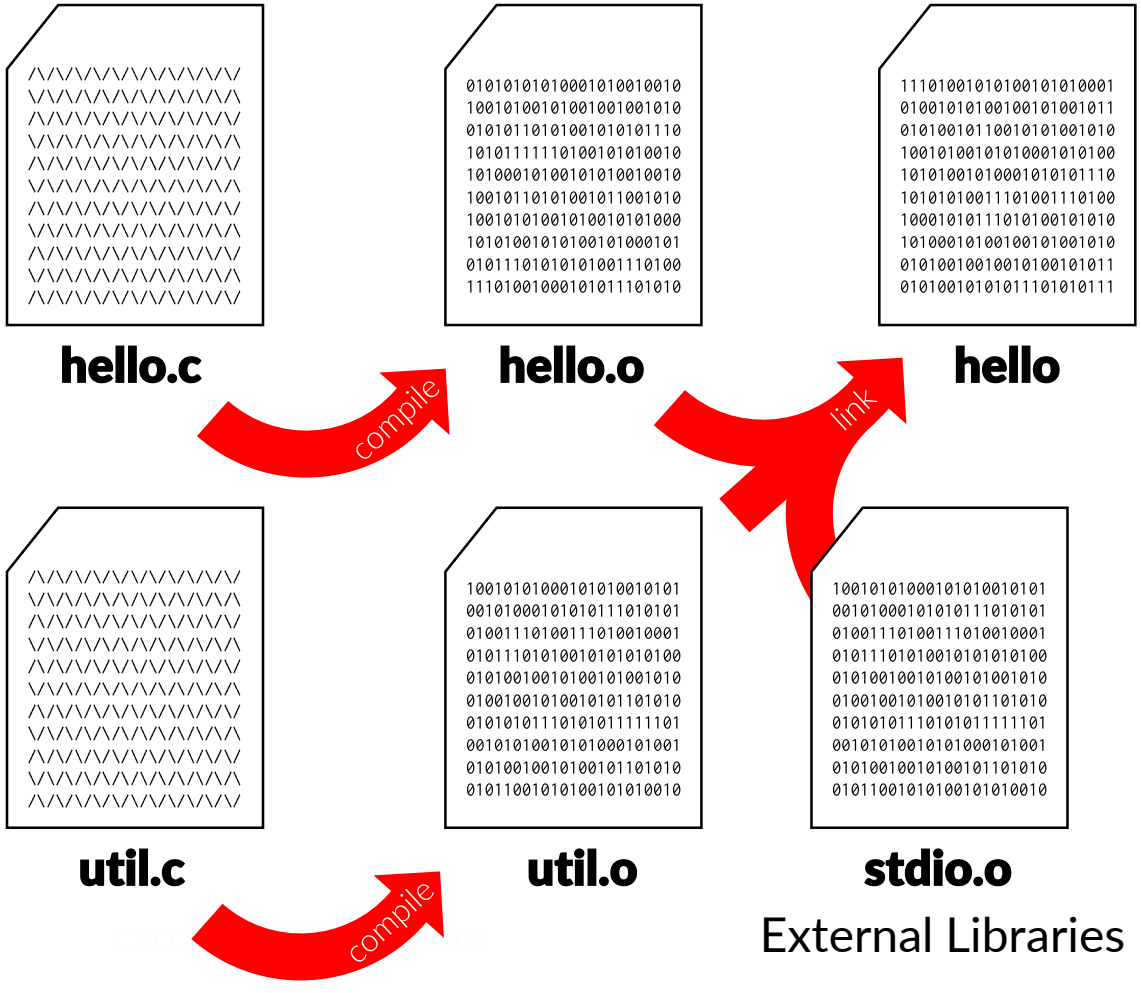
- The compiler takes source code (*.c files) and translates them into machine code.
- This file is called an "***object file***" and is just potentially one part of your overall project.
- The machine code is not quite an executable.
  - This object file is JUST representing the code for that particular source file.
  - You may require extra stuff provided by the system elsewhere.

Compilation: Simple Overview – Step 2



- You may have multiple files.
- They may reference each other.
  - For instance, one file may contain certain common functionality and then this is invoked by your program elsewhere.
- You break your project up into pieces similarly to your Java programs.
- The compiler treats them independently.

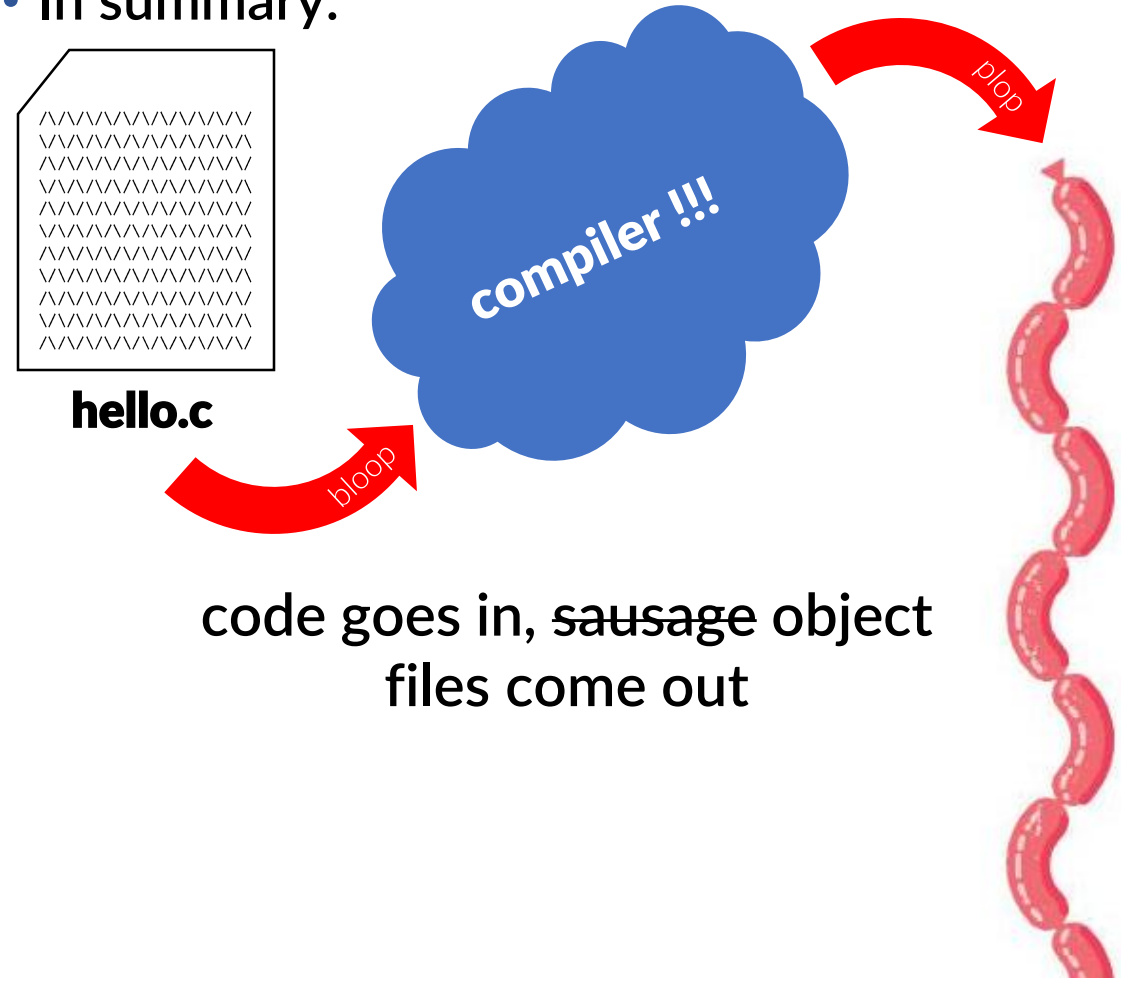
Compilation: Simple Overview – Step 3



- Then, each piece is merged together to form the executable.
- This process is called ***linking***.
  - The name refers to how the references to functions, etc, between files are now filled in.
  - Before this step… it is unclear *where* functions will end up in the final executable.
  - Keep this in mind as we look at memory and pointers later!

It's just a grinder.

- In summary:



code goes in, ~~sausage~~ object files come out

Some compilers output assembly and rely on an assembler to produce machine code

These days, it's common for the compiler itself to produce machine code, or some kind of platform-independent assembly code (typically: a bytecode)

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.

C vs. Java

	C (C99)	Java
Type of Language	Function Oriented	Object Oriented
Programming Unit	Function	Class = Abstract Data Type
Compilation	*gcc hello.c* - creates machine language code	*javac Hello.java* - creates Java virtual machine language bytecode
Execution	*a.out* - loads and executes program	*java Hello* - interprets bytecodes

hello, world	`#include<stdio.h>`
	`int main(void) {`
	`  printf("Hello World\n");`
	`  return 0;`
	`}`
		`public class HelloWorld {`
		`  public static void main(String[] args) {`
		`  System.out.println("Hello World");`
		`  }`
		`}`

Storage	Manual (**malloc**, **free**)	Automatic (garbage collection)

From http://www.cs.princeton.edu/introcs/faq/c2java.html

C vs. Java

	C (C99)	Java
Comments	`/* … */` or `//` … end of line	`/* … */` or `//` … end of line
Constants	`#define, const`	`final`
Preprocessor	Yes	No
Variable declaration	At beginning of a block	Before you use it
Variable naming conventions	`sum_of_squares`	`sumOfSquares`
Accessing a library	`#include <stdio.h>`	`import java.io.File;`

From http://www.cs.princeton.edu/introcs/faq/c2java.html

Hello World

```c
// 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;
}
```

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.
    - `gcc -std=c11 hello.c`
  - C18 – Minor refinement of C11. The current C standard.
    - `gcc -std=c18 hello.c`

- **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.

The C Syntax

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

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 “main” function

```c
// 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

```c
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

```c
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

```c
#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

```c
#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	>=32 bits (plain ints), infinite (long 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

```c
#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

```c
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

```c
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
```

The C Syntax: Control Flow

Once you C the program, you can BE 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 trouble is stacking up on us!

```c
#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 ? rand() : -a;
}

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

Output:

```
x = 0
x = 1804289383
x = -4
x = 846930886
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***