1 C Programming Language

> If you want to teach systems, don't drum up the programmers, sort the issues, and make PRs. Instead, teach them to yearn for the vast and endless C.
>
> Antoine de Saint-Exupéry (With edits from Bhuvy)

C is the de-facto programming language to do serious system serious programming. Why? Most kernels are written in largely in C. The Linux Kernel [6] and the XNU kernel Inc. [3] of which Mac OS X is based off. The Windows Kernel uses C++, but doing system programming on that is much harder on windows that UNIX for beginner system programmers. Most of you have some experience with C++, but C is a different beast entirely. You don't have nice abstractions like classes and RAII to clean up memory. You are going to have to do that yourself. C gives you much more of an opportunity to shoot yourself in the foot but lets you do things at a much finer grain level.

History of C

C was developed by Dennis Ritchie and Ken Thompson at Bell Labs back in 1973 [7]. Back then, we had gems of programming languages like Fortran, ALGOL, and LISP. The goal of C was two fold. One, to target the most popular computers at the time like the PDP-7. Two, try and remove some of the lower level constructs like managing registers, programming assembly for jumps and instead create a language that had the power to express programs procedurally (as opposed to mathematically like lisp) with more readable code all while still having the ability to interface with the operating system. It sounded like a tough feat. At first, it was only used internally at Bell Labs along with the UNIX operating system.

The first "real" standardization is with Brian Kernighan and Dennis Ritchie's book [5]. It is still widely regarded today as the only Portable set of C instructions. The K&R book is known as the de-facto standard for learning C. There were different standards of C from ANSI to ISO after the Unix guides. The one that we will be mainly focusing on is the POSIX C library. Now to get the elephant out of the room, the Linux kernel is not entirely POSIX compliant. Mostly, it is because they didn't want to pay the fee for compliance but also it doesn't want to be completely compliant with a bunch of different standards because then it has to ensue increasing development costs to maintain compliance.

Fast forward however many years, and we are at the current C standard put forth by ISO: C11. Not all the code that we us in this class will be in this format. We will aim to using C99 as the standard that most computers recognize. We will talk about some off-hand features like `getline` because they are so widely used with the GNU-C library. We'll begin by providing a decently comprehensive overview of the language with pairing facilities.

Features

- Fast. There is ver little separating you and the system.
- Simple. C and its standard library pose a simple set of portable functions.
- Memory Management. C let's you manage your memory. This can also bite you if you have memory errors.
- It's Everywhere. Pretty much every computer that is not embedded has some way of interfacing with C. The standard library is also everywhere. C has stood the test of time as a popular language, and it doesn't look like it is going anywhere.

Crash course intro to C

The only way to start learning C is by starting with hello world. As per the original example that Kernighan and Ritchie proposed way back when, the hello world hasn't changed that much.

```c
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

1. The `#include` directive takes the file `stdio.h` (which stands for **st**an**d**ard **i**nput and **o**utput) located somewhere in your operating system, copies the text, and substitutes it where the `#include` was.
2. The `int main(void)` is a function declaration. The first word `int` tells the compiler what the return type of the function is. The part before the parens (`main`) is the function name. In C, no two functions can have the same name in a single compiled program, shared libraries are a different touchy subject. Then, what comes after is the parameter list. When we give the parameter list for regular functions `(void)` that means that the compiler should error if the function is called with any arguments. For regular functions having a declaration like `void func()` means that you are allowed to call the function like `func(1, 2, 3)` because there is no delimiter. In the case of `main`, it is a special function. There are many ways of declaring `main` but the ones that you will be familiar with are `int main(void)`, `int main()`, and `int main(int argc, char *argv[])`.
3. `printf("Hello World");` is what we call a function call. `printf` is defined as a part of `stdio.h`. The function has been compiled and lives somewhere else on our machine. All we need to do is include the header and call the function with the appropriate parameters (a string literal `"Hello World"`). If you don't have the newline, the buffer will not be flushed. It is by convention that buffered IO is not flushed until a newline.
4. `return 0;`. `main` has to return an integer. By convention, `return 0` means success and anything else means failure.

```
$ gcc main.c -o main
$ ./main
Hello World
$
```

1. `gcc` is short for the GNU-Compiler-Collection which has a host of compilers ready for use. The compiler infers from the extension that you are trying to compile a .c file
2. `./main` tells your shell to execute the program in the current directory called main. The program then prints out hello world

Preprocessor

What is the preprocessor? Preprocessing is an operation that the compiler performs **before** actually compiling the program. It is a copy and paste command. Meaning the following substitution is performed.

```c
#define MAX_LENGTH 10
char buffer[MAX_LENGTH]
// After
char buffer[10]
```

There are side effects to the preprocessor though. One problem is that the preprocessor needs to be able to tokenize properly, meaning trying to redefine the internals of the C language with a preprocessor may be impossible. Another

problem is that they can't be nested infinitely – there is an unbounded depth where they need to stop. Macros are also just simple text substitutions. For example, look at what can happen if we have a macro that performs an inline modification.

```
#define min(a,b) ((a)<(b) ? (a) : (b))
int main() {
  int x = 4;
  if(min(x++, 5)) printf("%d is six", x);
  return 0;
}
```

Macros are simple text substitution so the above example expands to `x++ < 100 ? x++ : 100` (parenthesis omitted for clarity). Now for this case, it is opaque what gets printed out but it will be 6. Also consider the edge case when operator precedence comes into play.

```
#define min(a,b) a < b ? a : b
int x = 99;
int r = 10 + min(99, 100); // r is 100!
// This is what it is expanded to
int r = 10 + 99 < 100 ? 99 : 100
// Which means
int r = (10 + 99) < 100 ? 99 : 100
```

You can also have logical problems with the flexibility of certain parameters. One common source of confusion is with static arrays and the `sizeof` operator.

```
#define ARRAY_LENGTH(A) (sizeof((A)) / sizeof((A)[0]))
int static_array[10]; // ARRAY_LENGTH(static_array) = 10
int* dynamic_array = malloc(10); // ARRAY_LENGTH(dynamic_array) = 2 or 1
```

What is wrong with the macro? Well, it works if we have a static array like the first array because `sizeof` a static array returns the number of bytes that array takes up, and dividing it by the `sizeof(an_element)` would give you the number of entries. But if we use a pointer to a piece of memory, taking the sizeof the pointer and dividing it by the size of the first entry won't always give us the size of the array.

Extra: Includes and conditionals

The other preprocessor include is the `#include` directive and conditionals. The include directive is explained by example.

```
// foo.h
int bar();
```

```
// foo.c unpreprocessed
#include "foo.h"
int bar() {
}
```

After preprocessing, the compiler sees this

```
// foo.c unpreprocessed
int bar();

int bar() {

}
```

The other thing we have is conditionals. If a macro is defined or equal to zeros, that branch is not taken

```
int main() {
#ifdef __GNUC__
    return 1;
#else
    return 0;
#endif
}
```

Using `gcc` your compiler would see this

```
int main() {
    return 1;
}
```

Using `clang` your compiler would see this

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

Language Facilities

Keywords

C has an assortment of keywords. Here are some constructs that you should know briefly as of C99.

1. `break` is a keyword that is used in case statements or looping statements. When used in a case statement, the program jumps to the end of the block.

```
switch(1) {
  case 1: /* Goes to this switch */
    puts("1");
    break; /* Jumps to the end of the block */
  case 2: /* Ignores this program */
```

```
    puts("2");
    break;
} /* Continues here */
```

In the context of a loop, it breaks out of the inner-most loop. The loop can be either a `for`, `while`, or `do-while` construct

```
while(1) {
  while(2) {
    break; /* Breaks out of while(2) */
  } /* Jumps here */
  break; /* Breaks out of while(1) */
} /* Continues here */
```

2. `const` is a language level construct that tells the compiler that this data should not be modified. If one tries to change a const variable, the program will not even compile. `const` works a little differently when put before the type, the compiler flips the first type and const. Then the compiler uses a left associativity rule. Meaning that whatever is left of the pointer is constant. This is known as const-correctedness.

```
const int i = 0; // Same as "int const i = 0"
char *str = ...; // Mutable pointer to a mutable string
const char *const_str = ...; // Mutable pointer to a constant string
char const *const_str2 = ...; // Same as above
const char *const const_ptr_str = ...;
// Constant pointer to a constant string
```

But, it is important to know that this is a compiler imposed restriction only. There are ways of getting around this and the program will run fine with defined behavior. In systems programming, the only type of memory that you can't write to is system write-protected memory.

```
const int i = 0; // Same as "int const i = 0"
(*((int *)&i)) = 1; // i == 1 now
const char *ptr = "hi";
*ptr = '\0'; // Will cause a Segmentation Violation
```

3. `continue` is a control flow statement that exists only in loop constructions. Continue will skip the rest of the loop body and set the program counter back to the start of the loop before.

```
int i = 10;
while(i--) {
  if(1) continue; /* This gets triggered */
  *((int *)NULL) = 0;
} /* Then reaches the end of the while loop */
```

4. `do {} while();` is another loop constructs. These loops execute the body and then check the condition at the bottom of the loop. If the condition is zero, the loop body is not executed and the rest of the program is executed. Otherwise, the loop body is executed.

```
int i = 1;
do {
  printf("%d\n", i--);
} while (i > 10) /* Only executed once */
```

5. `enum` is to declare an enumeration. An enumeration is a type that can take on many, finite values. If you have an enum and don't specify any numerics, the c compiler when generate a unique number for that enum (within the context of the current enum) and use that for comparisons. To declare an instance of an enum, you must say `enum <type> varname`. The added benefit to this is that C can type check these expressions to make sure that you are only comparing alike types.

```
enum day{ monday, tuesday, wednesday,
  thursday, friday, saturday, sunday};

void process_day(enum day foo) {
  switch(foo) {
    case monday:
      printf("Go home!\n"); break;
    // ...
  }
}
```

It is completely possible to assign enum values to either be different or the same. Just don't rely on the compiler for consistent numbering. If you are going to use this abstraction, try not to break it.

```
enum day{
  monday = 0,
  tuesday = 0,
  wednesday = 0,
  thursday = 1,
  friday = 10,
  saturday = 10,
  sunday = 0};

void process_day(enum day foo) {
  switch(foo) {
    case monday:
      printf("Go home!\n"); break;
    // ...
  }
}
```

6. `extern` is a special keyword that tells the compiler that the variable may be defined in another object file or a library, so the compiler doesn't throw an error when either the variable is not defined or if the variable is defined twice because the first file will really be referencing the variable in the other file.

```c
// file1.c
extern int panic;

void foo() {
  if (panic) {
    printf("NONONONONO");
  } else {
    printf("This is fine");
  }
}

//file2.c

int panic = 1;
```

7. `for` is a keyword that allows you to iterate with an initialization condition, a loop invariant, and an update condition. This is meant to be a replacement for the while loop

```c
for (initialization; check; update) {
  //...
}

// Typically
int i;
for (i = 0; i < 10; i++) {
  //...
}
```

As of the C89 standard, you cannot declare variables inside the `for` loop. This is because there was a disagreement in the standard for how the scoping rules of a variable defined in the loop would work. It has since been resolved with more recent standards, so people can use the for loop that they know and love today

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

}
```

The order of evaluation for a `for` loop is as follows

(a) Perform the initialization condition.

(b) Check the invariant. If false, terminate the loop and execute the next statement. If true, continue to the body of the loop.

(c) Perform the body of the loop.

(d) Perform the update condition.

(e) Jump to checking the invariant step.

8. `goto` is a keyword that allows you to do conditional jumps. Do not use `goto` in your programs. The reason being is that it makes your code infinitely more hard to understand when strung together with multiple chains. It is fine to use in some contexts though. The keyword is usually used in kernel contexts when adding another stack frame for cleanup isn't a good idea. The canonical example of kernel cleanup is as below.

```
void setup(void) {
  Doe *deer;
  Ray *drop;
  Mi *myself;

  if (!setupdoe(deer)) {
    goto finish;
  }

  if (!setupray(drop)) {
    goto cleanupdoe;
  }

  if (!setupmi(myself)) {
    goto cleanupray;
  }

  perform_action(deer, drop, myself);

cleanupray:
  cleanup(drop);
cleanupdoe:
  cleanup(deer);
finish:
  return;
}
```

9. `if else else-if` are control flow keywords. There are a few ways to use these (1) A bare if (2) An if with an else (3) an if with an else-if (4) an if with an else if and else. The statements are always executed from the if to the else. If any of the intermediate conditions are true, the if block performs that action and goes to the end of that block.

```
// (1)

if (connect(...))
  return -1;

// (2)
if (connect(...)) {
  exit(-1);
} else {
  printf("Connected!");
}

// (3)
if (connect(...)) {
  exit(-1);
```

```
} else if (bind(..)) {
  exit(-2);
}

// (1)
if (connect(...)) {
  exit(-1);
} else if (bind(..)) {
  exit(-2);
} else {
  printf("Successfully bound!");
}
```

10. `inline` is a compiler keyword that tells the compiler it's okay not to create a new function in the assembly. Instead, the compile is hinted at substituting the function body directly into the calling function. This is not always recommended explicitly as the compiler is usually smart enough to know when to `inline` a function for you.

```
inline int max(int a, int b) {
  return a < b ? a : b;
}

int main() {
  printf("Max %d", max(a, b));
  // printf("Max %d", a < b ? a : b);
}
```

11. `restrict` is a keyword that tells the compiler that this particular memory region shouldn't overlap with all other memory regions. The use case for this is to tell users of the program that it is undefined behavior if the memory regions overlap.

```
memcpy(void * restrict dest, const void* restrict src, size_t bytes);

void add_array(int *a, int * restrict c) {
  *a += *c;
}
int *a = malloc(3*sizeof(*a));
*a = 1; *a = 2; *a = 3;
add_array(a + 1, a) // Well defined
add_array(a, a) // Undefined
```

12. `return` is a control flow operator that exits the current function. If the function is `void` then it simply exits the functions. Otherwise another parameter follows as the return value.

```
void process() {
  if (connect(...)) {
    return -1;
  } else if (bind(...)) {
    return -2
```

```
  }
  return 0;
}
```

13. `signed` is a modifier which is rarely used, but it forces an type to be signed instead of unsigned. The reason that this is so rarely used is because types are signed by default and need to have the `unsigned` modifier to make them unsigned but it may be useful in cases where you want the compiler to default a signed type like.

```
int count_bits_and_sign(signed representation) {
  //...
}
```

14. `sizeof` is an operator that is evaluated at compile time, which evaluates to the number of bytes that the expression contains. Meaning that when the compiler infers the type the following code changes.

```
char a = 0;
printf("%zu", sizeof(a++));
```

```
char a = 0;
printf("%zu", 1);
```

Which then the compiler is allowed to operate on further. A note that you must have a complete definition of the type at compile time or else you may get an odd error. Consider the following

```
// file.c
struct person;

printf("%zu", sizeof(person));

// file2.c

struct person {
  // Declarations
}
```

This code will not compile because sizeof is not able to compile `file.c` without knowing the full declaration of the `person` struct. That is typically why we either put the full declaration in a header file or we abstract the creation and the interaction away so that users cannot access the internals of our struct. Also, if the compiler knows the full length of an array object, it will use that in the expression instead of decaying it to a pointer.

```
char str1[] = "will be 11";
char* str2 = "will be 8";
```

```
sizeof(str1) //11 because it is an array
sizeof(str2) //8 because it is a pointer
```

Be careful, using sizeof for the length of a string!

15. `static` is a type specifier with three meanings.

    (a) When used with a global variable or function declaration it means that the scope of the variable or the function is only limited to the file.

    (b) When used with a function variable, that declares that the variable has static allocation – meaning that the variable is allocated once at program start up not every time the program is run.

```
static int i = 0;

static int _perform_calculation(void) {
  // ...
}

char *print_time(void) {
  static char buffer[200]; // Shared every time a function is called
  // ...
}
```

16. `struct` is a keyword that allows you to pair multiple types together into a new structure. Structs are contiguous regions of memory that one can access specific elements of each memory as if they were separate variables.

```
struct hostname {
  const char *port;
  const char *name;
  const char *resource;
}; // You need the semicolon at the end
// Assign each individually
struct hostname facebook;
facebook.port = "80";
facebook.name = "www.google.com";
facebook.resource = "/";

// You can use static initialization in later versions of c
struct hostname google = {"80", "www.google.com", "/"};
```

17. `switch case default` Switches are essentially glorified jump statements. Meaning that you take either a byte or an integer and the control flow of the program jumps to that location.

```
switch(/* char or int */) {
  case INT1: puts("1");
  case INT2: puts("2");
  case INT3: puts("3");
```

```
}
```

If we give a value of 2 then

```
switch(2) {
  case 1: puts("1"); /* Doesn't run this */
  case 2: puts("2"); /* Runs this */
  case 3: puts("3"); /* Also runs this */
}
```

One of the more famous examples of this is Duff's device which allows for loop unrolling. You don't need to understand this code for the purposes of this class, but it is fun to look at [2].

```
send(to, from, count)
register short *to, *from;
register count;
{
  register n=(count+7)/8;
  switch(count%8){
  case 0:  do{ *to = *from++;
  case 7:      *to = *from++;
  case 6:      *to = *from++;
  case 5:      *to = *from++;
  case 4:      *to = *from++;
  case 3:      *to = *from++;
  case 2:      *to = *from++;
  case 1:      *to = *from++;
    }while(--n>0);
  }
}
```

This piece of code highlights that switch statements are just goto statements, and you can put whatever other valid piece of code on the other end of a switch case. Most of the time it doesn't make sense, some of the time it just makes too much sense.

18. `typedef` declares an alias for a type. Often used with structs to reduce the visual clutter of having to write 'struct' as part of the type.

```
typedef float real;
real gravity = 10;
// Also typedef gives us an abstraction over the underlying type used.
// In the future, we only need to change this typedef if we
// wanted our physics library to use doubles instead of floats.

typedef struct link link_t;
//With structs, include the keyword 'struct' as part of the original
    types
```

In this class, we regularly typedef functions. A typedef for a function can be this for example

```
typedef int (*comparator)(void*,void*);

int greater_than(void* a, void* b){
    return a > b;
}
comparator gt = greater_than;
```

This declares a function type comparator that accepts two `void*` params and returns an integer.

19. `union` is a new type specifier. A union is one piece of memory that a bunch of variables occupy. It is used to maintain consistency while having the flexibility to switch between types without maintaining functions to keep track of the bits. Consider an example where we have different pixel values.

```
union pixel {
  struct values {
    char red;
    char blue;
    char green;
    char alpha;
  } values;
  uint32_t encoded;
}; // Ending semicolon needed
union pixel a;
// When modifying or reading
a.values.red;
a.values.blue = 0x0;

// When writing to a file
fprintf(picture, "%d", a.encoded);
```

20. `unsigned` is a type modifier that forces `unsigned` behavior in the variables they modify. Unsigned can only be on primitive int types (like `int` and `long`). There is a lot of behavior associated with unsigned arithmetic, just know for the most part unless you need to do bit shifting you probably won't need it.

21. `void` is a two folded keyword. When used in terms of function or parameter definition then it means that it returns no value or accepts no parameter specifically. The following declares a function that accepts no parameters and returns nothing.

```
void foo(void);
```

The other use of `void` is when you are defining. A `void *` pointer is just a memory address. It is specified as an incomplete type meaning that you cannot dereference it but it can be promoted to any time to any other type. Pointer arithmetic with these pointer is undefined behavior.

```
int *array = void_ptr; // No cast needed
```

22. `volatile` is a compiler keyword. This means that the compiler should not optimize its value out. Consider the following simple function.

```
int flag = 1;
pass_flag(&flag);
while(flag) {
    // Do things unrelated to flag
}
```

The compiler may, since the internals of the while loop have nothing to do with the flag, optimize it to the following even though a function may alter the data.

```
while(1) {
    // Do things unrelated to flag
}
```

If you put the volatile keyword then it forces the compiler to keep the variable in and perform that check. This is particularly useful for cases where you are doing multi-process or multi-threaded programs so that we can

23. `while` represents the traditional `while` loop. There is a condition at the top of the loop. While that condition evaluates to a non-zero value, the loop body will be run.

C data types

There are many data types in c. All of them as you may realize are either an integer or a floating point number and different types are variations on that.

1. `char` Represents exactly one byte of data. The number of bits in a byte might vary. `unsigned char` and `signed char` mean the exact same thing. This must be aligned on a boundary (meaning you cannot use bits in between two addresses). The rest of the types will assume 8 bits in a byte.
2. `short (short int)` must be at least two bytes. This is aligned on a two byte boundary, meaning that the address must be divisible by two.
3. `int` must be at least two bytes. Again aligned to a two byte boundary [4, P. 34]. On most machines this will be 4 bytes.
4. `long (long int)` must be at least four bytes, which are aligned to a four byte boundary. On some machines this can be 8 bytes.
5. `long long` must be at least eight bytes, aligned to an eight byte boundary.
6. `float` represents an IEEE-754 single precision floating point number tightly specified by IEEE [1]. This will be four bytes aligned to a four byte boundary on most machines.
7. `double` represents an IEEE-754 double precision floating point number specified by the same standard, which is aligned to the nearest eight byte boundary.

Operators

Operators are language constructs in C that are defined as part of the grammar of the language.

1. `[]` is the subscript operator. `a[n] == (a + n)*` where `n` is a number type and `a` is a pointer type.

2. `->` is the structure dereference operator. If you have a pointer to a struct `*p`, you can use this to access one of its elements. `p->element`.
3. `.` is the structure reference operator. If you have an object on the stack `a` then you can access an element `a.element`.
4. `+/-a` is the unary plus and minus operator. They either keep or negate the sign, respectively, of the integer or float type underneath.
5. `*a` is the dereference operator. If you have a pointer `*p`, you can use this to access the element located at this memory address. If you are reading, the return value will be the size of the underlying type. If you are writing, the value will be written with an offset.
6. `&a` is the addressof operator. This takes the an element and returns its address.
7. `++` is the increment operator. You can either take it prefix or postfix, meaning that the variable that is being incremented can either be before or after the operator. `a = 0; ++a === 1` and `a = 1; a++ === 0`.
8. `-` is the decrement operator. Same semantics as the increment operator except with decreasing the value by one.
9. `sizeof` is the sizeof operator. This is also mentioned in the keywords section.
10. `a <mop> b` where `<mop>=+, -, *, %, /` are the mathematical binary operators. If the operands are both number types, then the operations are plus, minus, times, modulo, and division respectively. If the left operand is a pointer and the right operand is an integer type, then only plus or minus may be used and the rules for pointer arithmetic are invoked.
11. `»/«` are the bit shift operators. The operand on the right has to be an integer type whose signedness is ignored unless it is signed negative in which case the behavior is undefined. The operator on the left decides a lot of semantics. If we are left shifting, there will always be zeros introduced on the right. If we are right shifting there are a few different cases
    - If the operand on the left is signed, then the integer is sign extended. This means that if the number has the sign bit set, then any shift right will introduce ones on the left. If the number does not have the sign bit set, any shift right will introduce zeros on the left.
    - If the operand is unsigned, zeros will be introduced on the left either way.

```
unsigned short uns = -127; // 1111111110000001
short sig = 1; // 0000000000000001
uns << 2; // 1111111000000100
sig << 2; // 0000000000000100
uns >> 2; // 1111111111100000
sig >> 2; // 0000000000000000
```

12. `<=/>=` are the greater than equal to/less than equal to operators. They do as the name implies.
13. `</>` are the greater than/less than operators. They again do as the name implies.
14. `==/=` are the equal/not equal to operators. They once again do as the name implies.
15. `&&` is the logical and operator. If the first operand is zero, the second won't be evaluated and the expression will evaluate to 0. Otherwise, it yields a 1-0 value of the second operand.
16. `||` is the logical or operator. If the first operand is not zero, then second won't be evaluated and the expression will evaluate to 1. Otherwise, it yields a 1-0 value of the second operand.
17. `!` is the logical not operator. If the operand is zero, then this will return 1. Otherwise, it will return 0.
18. `&` If a bit is set in both operands, it is set in the output. Otherwise, it is not.

19. `|` If a bit is set in either operand, it is set in the output. Otherwise, it is not.
20. If a bit is set in the input, it will not be set in the output and vice versa.
21. `?:` is the ternary operator. You put a boolean condition before the and if it evaluates to non-zero the element before the colon is returned otherwise the element after is. `1 ? a : b === a` and `0 ? a : b === b`.
22. `a, b` is the comma operator. `a` is evaluated and then `b` is evaluated and `b` is returned.

The C and Linux

So up until this point we've covered language fundamentals with C. What starts diverging from all forms of C is the functions we use and how we interact with the operating system. We'll now be focusing our attention to C and the POSIX variety of functions available to us. We will talk about portable functions, for example `fwrite printf` etc etc, but we will be evaluating the internals and scrutinizing them under the POSIX models and more specifically LINUX. There are a number of things to that philosophy that makes the rest of this easier to know, so we'll put those things here.

Everything is a File

One POSIX mantra is that everything is a File. Although that has become recently outdated, and moreover wrong, it is the convention we still use today. What POSIX means is everything is a file is that everything is a file descriptor or an integer. For example, here is a file object, a network socket, and a kernel object

```c
int file_fd = open(...);
int network_fd = socket(...);
int kernel_fd = epoll_create1(...);
```

And operations on those objects are done through system calls. One last thing to note before we move on is that the file descriptors are merely *pointers*. Imagine that each of the file descriptors in the example actually refer to an entry in a table of objects that the operating system picks and chooses from. Objects can be allocated and deallocated, closed and opened, etc. The program interacts with these objects by using the API or system calls specified

System Calls

Before we dive into common C functions, we need to know what a system call is. If you are a student and completed HW0, feel free to gloss over this section.

A system call is an operation that the kernel carries out instead of the program. The operating system prepares a system call then the kernel executes the system call to the best of its ability in kernel space. In the previous example we opened up a file descriptor object. We can now also write some bytes to the file descriptor object that represents a file and the oeprating system will do its best to get the bytes written to the disk.

```c
write(file_fd, "Hello!", 6);
```

When we say the kernel tries its best, the operation could fail for a number of reasons. The file is no longer valid, the hard drive failed, the system was interrupted etc etc. The way that you communicate with the outside system is with system calls though. The other thing to note is that system calls are expensive. Their cost in terms of time and CPU cycles has recently been decreased, but try to use them as sparingly as possible.

C Calls

Many C calls that we will discuss in the next sections will call some of the calls above in their linux implementation. Their Windows implementation may be entirely different. But we will be looking at one operating system in general.

Common C Functions

To find more information about any functions, use the man pages. Note the man pages are organized into sections. Section 2 are System calls. Section 3 are C libraries. On the web, Google `man 7 open`. In the shell, `man -S2 open` or `man -S3 printf`

Input/Output

In this section we will cover all the basic input and output functions in the standard library with references to system calls. As a little bit of terminology, when your program is running on a terminal your Standard Output is your terminal and Standard Input is your terminal waiting for input. There are other cases as you'll see later in this course of redirecting those to point to other things. They are designated by the filedescriptors 0 and 1 respectively. 2 is reserved for standard error which by library convention is unbuffered.

stdout oriented streams

Standard output or stdout oriented streams are streams whose only options are to write to stdout. `printf` is the function with which most people are familiar in this category. The first parameter is a format string that includes placeholders for the data to be printed. Common format specifiers are the following

1. `%s` treat the argument as a c string pointer, keep printing all characters until the NULL-character is reached
2. `%d` prints the argument as an integer
3. `%p` print the argument as a memory address.

By default, for performance, `printf` does not actually write anything out until its buffer is full or a newline is printed. Here is an example of printing things out.

```
char *name = ... ; int score = ...;
printf("Hello %s, your result is %d\n", name, score);
printf("Debug: The string and int are stored at: %p and %p\n", name,
    &score );
// name already is a char pointer and points to the start of the array.
// We need "&" to get the address of the int variable
```

From the previous section, `printf` calls the system call `write`. `printf` includes an internal buffer so, to increase performance `printf` may not call `write` every time you call `printf`. Another quirk of printf it is forced to call `write` on a newline character. If one wants to `printf` to call write without a newline `fflush( FILE* inp )`. Lastly, `printf` is a C library function. `write` is a system call and as we know system calls are expensive. On the other hand, `printf` uses a buffer which suits our needs better at that point. Meaning, don't over-optimize about using printf a lot even if system calls are expensive – programs do it all the time.

To print strings and single characters use `puts( name )` and `putchar( c )` where name is a pointer to a C string and c is just a `char`

```
puts("Current selection: ");
putchar('1');
```

Other streams

To print to other file streams use `fprintf( _file_ , "Hello %s, score: %d", name, score);` Where _file_ is either predefined 'stdout' 'stderr' or a FILE pointer that was returned by `fopen` or `fdopen`. You can also use file descriptors in the printf family of functions! Just use `dprintf(int fd, char* format_string, ...);`. Just remember the stream may be buffered through internal buffering, so you will need to assure that the data is written to the file descriptor.

To print data into a C string, use `sprintf` or better `snprintf`. `snprintf` returns the number of characters written excluding the terminating byte. In the following, this would be a maximum of 199. We would use `sprintf` in cases where

we know that the size of the string will not be anything more than a certain fixed amount – think about printing an integer, it will never be more than 11 characters with the null byte.

```
// Fixed
char int_string[20];
sprintf(int_string, "%d", integer);

// Variable length
char result[200];
int len = snprintf(result, sizeof(result), "%s:%d", name, score);
```

stdin oriented functions

Standard input or stdin oriented functions read from stdin directly. Most of these functions have been deprecated due to them being poorly designed, as such we treat stdin as a file that we can read bytes from. One of the most notorious offenders is `gets`. `gets` is deprecated in C99 standard and has been removed from the latest C standard (C11). The reason that it was deprecated was that there is no way to control for the length, so strings could get overrun very easily.

Programs should use `fgets` or `getline` instead. Here is a quick example of reading at most 10 characters from stdin.

```
char *fgets (char *str, int num, FILE *stream);

ssize_t getline(char **lineptr, size_t *n, FILE *stream);

// Example, the following will not read more than 9 chars
char buffer[10];
char *result = fgets(buffer, sizeof(buffer), stdin);
```

The result is NULL if there was an error or the end of the file is reached. Note, unlike `gets`, `fgets` copies the newline into the buffer, which you may want to discard. On the other hand, one of the advantages of `getline` is that will automatically allocate and reallocate a buffer on the heap of sufficient size.

```
// ssize_t getline(char **lineptr, size_t *n, FILE *stream);

 /* set buffer and size to 0; they will be changed by getline */
char *buffer = NULL;
size_t size = 0;

ssize_t chars = getline(&buffer, &size, stdin);

// Discard newline character if it is present,
if (chars > 0 && buffer[chars-1] == '\n')
    buffer[chars-1] = '\0';

// Read another line.
// The existing buffer will be re-used, or, if necessary,
// It will be 'free''d and a new larger buffer will 'malloc''d
chars = getline(&buffer, &size, stdin);

// Later... don't forget to free the buffer!
free(buffer);
```

In addition to those functions, we have `perror` that has a two-fold meaning. Let's say that you have a function call that just failed because you checked the man page, and it is a failing return code. `perror(const char* message)` will print the English version of the error to stderr.

```
int main(){
    int ret = open("IDoNotExist.txt", O_RDONLY);
    if(ret < 0){
        perror("Opening IDoNotExist:");
    }
    //...
    return 0;
}
```

To have a library function parse input in addition to reading it, use `scanf` (or `fscanf` or `sscanf`) to get input from the default input stream, an arbitrary file stream or a C string respectively. All of those functions will return how many items were parsed; it is a good idea to check if the number is equal to the amount expected. Also naturally like `printf`, `scanf` functions require valid pointers. Instead of just pointing to valid memory, they need to also be writable. It's a common source of error to pass in an incorrect pointer value. For example,

```
int *data = malloc(sizeof(int));
char *line = "v 10";
char type;
// Good practice: Check scanf parsed the line and read two values:
int ok = 2 == sscanf(line, "%c %d", &type, &data); // pointer error
```

We wanted to write the character value into c and the integer value into the malloc'd memory. However, we passed the address of the data pointer, not what the pointer is pointing to! So `sscanf` will change the pointer itself. The pointer will now point to address 10 so this code will later fail when free(data) is called.

Now, scanf will just keep reading characters until the string ends. To stop scanf from causing a buffer overflow, use a format specifier. Make sure to pass one less than the size of the buffer.

```
char buffer[10];
scanf("%9s", buffer); // reads up to 9 characters from input (leave room
    for the 10th byte to be the terminating byte)
```

One last thing to note is if you thought system calls were expensive, the `scanf` family is much more expensive due to compatibility reasons. Imagine needing to take all of the printf specifiers correctly, the code isn't going to be very efficient. Most of the time, if you are writing a standalone program, write the parser yourself. If it is a one of program, feel free to use scanf.

string.h

String.h functions are a series of functions that deal with how to manipulate and check pieces of memory. Most of them deal with C-Strings A C-String is a series of bytes delimited by a NUL character which is eaul to the byte 0x00. More information about all of these functions. Any behavior not in the docs like passing `strlen(NULL)` is considered undefined behavior.

- `int strlen(const char *s)` returns the length of the string not including the null byte
- `int strcmp(const char *s1, const char *s2)` returns an integer determining the lexicographic order of the strings. If s1 where to come before s2 in a dictionary, then a -1 is returned. If the two strings are equal, then 0. Else, 1.

- `char *strcpy(char *dest, const char *src)` Copies the string at `src` to `dest`. **assumes dest has enough space for src**
- `char *strcat(char *dest, const char *src)` Concatenates the string at `src` to the end of destination. **This function assumes that there is enough space for `src` at the end of destination including the NULL byte**
- `char *strdup(const char *dest)` Returns a `malloc`'d copy of the string.
- `char *strchr(const char *haystack, int needle)` Returns a pointer to the first occurrence of `needle` in the `haystack`. If none found, `NULL` is returned.
- `char *strstr(const char *haystack, const char *needle)` Same as above but this time a string!
- `char *strtok(const char *str, const char *delims)`

  A dangerous but useful function strtok takes a string and tokenizes it. Meaning that it will transform the strings into separate strings. This function has a lot of specs so please read the man pages a contrived example is below.

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

int main(){
    char* upped = strdup("strtok,is,tricky,!!");
    char* start = strtok(upped, ",");
    do{
        printf("%s\n", start);
    }while((start = strtok(NULL, ",")));
    return 0;
}
```

Output

```
strtok
is
tricky
!!
```

Why is it tricky? Well what happens when I change `upped` like this?

```c
char* upped = strdup("strtok,is,tricky,,,!!");
```

- For integer parsing use `long int strtol(const char *nptr, char **endptr, int base);` or `long long int strtoll(const char *nptr, char **endptr, int base);`.

  What these functions do is take the pointer to your string `*nptr` and a `base` (ie binary, octal, decimal, hexadecimal etc) and an optional pointer `endptr` and returns a parsed value.

```c
int main(){
    const char *nptr = "1A2436";
    char* endptr;
```

```
    long int result = strtol(nptr, &endptr, 16);
    return 0;
}
```

Be careful though! Error handling is tricky because the function won't return an error code. If you give it a string that is not a number it will return 0. This means you can't differentiate between a valid "0" and an invalid string. See the man page for more details on strol behavior with invalid and out of bounds values. A safer alternative is use to `sscanf` (and check the return value).

```
int main(){
    const char *input = "0"; // or "!##@" or ""
    char* endptr;
    long int parsed = strtol(input, &endptr, 10);
    if(parsed == 0){
        // Either the input string was not a valid base-10 number or
            it really was zero!

    }
    return 0;
}
```

- `void *memcpy(void *dest, const void *src, size_t n)` moves `n` bytes starting at `src` to `dest`. **Be careful**, there is undefined behavior when the memory regions overlap. This is one of the classic works on my machine examples because many times valgrind won't be able to pick it up because it will look like it works on your machine. When the autograder hits, fail. Consider the safer version below.
- `void *memmove(void *dest, const void *src, size_t n)` does the same thing as above, but if the memory regions overlap then it is guaranteed that all the bytes will get copied over correctly. `memcpy` and `memmove` both in `string.h`? Because strings are essentially raw memory with a null byte at the end of them!

C Memory Model

The C memory model is probably unlike most that you've seen before. Instead of allocating an object with type safety, we either use an automatic variable or request a sequence of bytes with `malloc` or another family member and later we `free` it.

Structs

In low-level terms, a struct is just a piece of contiguous memory, nothing more. Just like an array, a struct has enough space to keep all of its members. But unlike an array, it can store different types. Consider the contact struct declared above

```
struct contact {
    char firstname[20];
    char lastname[20];
    unsigned int phone;
};

struct contact bhuvan;
```

```c
/* a lot of times we will do the following typdef
 so we can just write contact contact1 */

typedef struct contact contact;
contact bhuvan;

/* You can also declare the struct like this to get
 it done in one statement */
typedef struct optional_name {
    ...
} contact;
```

If you compile the code without any optimizations and reordering, you can expect the addresses of each of the variables to look like this.

```c
&bhuvan          // 0x100
&bhuvan.firstname // 0x100 = 0x100+0x00
&bhuvan.lastname // 0x114 = 0x100+0x14
&bhuvan.phone    // 0x128 = 0x100+0x28
```

Because all your compiler does is say 'reserve this much space, and I will go and calculate the offsets of whatever variables you want to write to'. The offsets are where the variable starts at. The phone variables starts at the 0x128th bytes and continues for sizeof(int) bytes, but not always. **Offsets don't determine where the variable ends though**. Consider the following hack that you see in a lot of kernel code.

```c
typedef struct {
    int length;
    char c_str[0];
} string;

const char* to_convert = "bhuvan";
int length = strlen(to_convert);

// Let's convert to a c string
string* bhuvan_name;
bhuvan_name = malloc(sizeof(string) + length+1);
/*
Currently, our memory looks like this with junk in those black spaces
                ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
 bhuvan_name = |___|___|___|___|___|___|___|___|___|___|___|

*/


bhuvan_name->length = length;
/*
This writes the following values to the first four bytes
The rest is still garbage
                ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
 bhuvan_name = | 0 | 0 | 0 | 6 |___|___|___|___|___|___|___|
```

```
*/


strcpy(bhuvan_name->c_str, to_convert);
/*
Now our string is filled in correctly at the end of the struct


                 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ____
 bhuvan_name = | 0 | 0 | 0 | 6 | b | h | u | v | a | n | \0 |
                                                       âĂ¿
*/

strcmp(bhuvan_name->c_str, "bhuvan") == 0 //The strings are equal!
```

What that zero length array does is point to the **end of the struct** this means that the compiler will leave room for all of the elements calculated with respect to their size on the operating system (ints, chars, etc). The zero length array will take up no bytes of space. Since structs are just continuous pieces of memory, we can allocate **more** space than required and use the extra space as a place to store extra bytes. Although this seems like a parlour trick, it is an important optimization because to have a variable length string any other way, one would need to have two different memory allocation calls. This is highly inefficient for doing something as common in programming as string manipulation.

Extra: Struct packing

Structs may require something called padding (tutorial). **We do not expect you to pack structs in this course, just know that it is there** This is because in the early days (and even now) when you have to an address from memory you have to do it in 32bit or 64bit blocks. This also meant that you could only request addresses that were multiples of that. Meaning that

```
struct picture{
    int height;
    pixel** data;
    int width;
    char* encoding;
}
```

You think picture looks like this. One box is four bytes

```
| h |  data | w | encod |
|---+-------+---+-------|
|___|___|___|___|___|___|
```

With struct packing, would conceptually look like this

```
struct picture{
    int height;
    char slop1[4];
    pixel** data;
    int width;
    char slop2[4];
    char* encoding;
}
```

Here is some more ascii art for you

```
| h | ? | data  | w | ? | encod |
|---+---+-------+---+---+-------|
|___|___|___ ___|___|___|___ ___|
```

This is on a 64-bit system. This is not always the case because sometimes your processor supports unaligned accesses. What does this mean? Well there are two options you can set an attribute

```
struct __attribute__((packed, aligned(4))) picture{
    int height;
    pixel** data;
    int width;
    char* encoding;
}
```

```
| h |  data | w | encod |
|---+-------+---+-------|
|___|___|___|___|___|___|
```

But now, every time I want to access `data` or `encoding`, I have to do two memory accesses. The other thing you can do is reorder the struct, although this is not always possible

```
struct picture{
    int height;
    int width;
    pixel** data;
    char* enconding;
}
```

```
| h | w | data  | encod |
|---+---+---+---+-------|
|___|___|___|___|___|___|
```

Strings in C

In C we have Null Terminated strings rather than Length Prefixed for historical reasons. What that means for your average everyday programming is that you need to remember the null character! A string in C is defined as a bunch of bytes until you reach ” or the Nul Byte.

Places for strings

Whenever you define a constant string – one in the form `char* str = "constant"` – that string is stored in the *data* depending on your architecture which is **read-only** meaning that any attempt to modify the string will cause a segfault. One can also declare strings to be either in the writable data segment or the stack. To do so, just specify a length for the string or put brackets instead of a pointer `char str[] = "mutable"` and put in the global scope or the function scope for the data segment or the stack respectively. If one, however, `malloc`'s space, one can change that string to be whatever they want. Forgetting to NUL terminate a string is a big effect on the strings! Bounds checking is important. The heartbleed bug mentioned earlier in the book is partially because of this.

Strings in C are represented as characters in memory. The end of the string includes a NUL (0) byte. So "ABC" requires four(4) bytes. The only way to find out the length of a C string is to keep reading memory until you find the NULL byte. C characters are always exactly one byte each.

String constants are constant

A string constant is naturally constant. Any write will cause the operating system to produce a SEGFAULT.

```
char array[] = "Hi!"; // array contains a mutable copy
strcpy(array, "OK");

char *ptr = "Can't change me"; // ptr points to some immutable memory
strcpy(ptr, "Will not work");
```

String literals are character arrays stored in the code segment of the program, which is immutable. Two string literals may share the same space in memory. An example follows.

```
char *str1 = "Bhuvy likes books";
char *str2 = "Bhuvy likes books";
```

The strings pointed to by `str1` and `str2` may actually reside in the same location in memory.

Char arrays, however, contain the literal value which has been copied from the code segment into either the stack or static memory. These following char arrays do not reside in the same place in memory.

```
char arr1[] = "Bhuvy also likes to write";
char arr2[] = "Bhuvy also likes to write";
```

Here are some common ways to initialize a string include. Where do they reside in memory?

```
char *str = "ABC";
char str[] = "ABC";
char str[]={'A','B','C','\0'};
```

```
char ary[] = "Hello";
char *ptr = "Hello";
```

We can also print out the pointer and the contents of a c string very easily. Here is some boilerplate code to do the printout.

```
char ary[] = "Hello";
char *ptr = "Hello";
// Print out address and contents
printf("%p : %s\n", ary, ary);
printf("%p : %s\n", ptr, ptr);
```

As mentioned before, the char array is mutable, so we can change its contents. Be careful not to write bytes beyond the end of the array though. Also again be careful not to get the two mixed up.

```
strcpy(ary, "World"); // OK
strcpy(ptr, "World"); // NOT OK - Segmentation fault (crashes)
```

We can, however, unlike the array, we change `ptr` to point to another piece of memory,

```
ptr = "World"; // OK!
ptr = ary; // OK!
ary = "World"; // NO won't compile
// ary is doomed to always refer to the original array.
printf("%p : %s\n", ptr, ptr);
strcpy(ptr, "World"); // OK because now ptr is pointing to mutable memory
    (the array)
```

What to take away from this is that pointers * can point to any type of memory while the char arrays mentioned earlier will always refer to the same piece of memory. In a more common case, pointers will point to heap memory in which case the memory referred to by the pointer **can** be modified.

Pointers

Up until now, you may have not had a lot of work to do with pointers. Pointers are variables that hold addresses. These addresses have numeric value, but usually we care about the contents underneath. In this section, we will try to take your through a very basic introduction of pointers.

Pointer Basics

Declaring a Pointer

A pointer refers to a memory address. The type of the pointer is useful – it tells the compiler how many bytes need to be read/written and what the semantics for addition are.

```
int *ptr1;
char *ptr2;
```

Due to C's grammar, an `int*` or any pointer is not actually its own type. You have to precede each pointer variable with an asterisk. As a common gotcha, the following

```
int* ptr3, ptr4;
```

Will only declare `*ptr3` as a pointer. `ptr4` will actually be a regular int variable. To fix this declaration, keep the `*` preceding to the pointer

```
int *ptr3, *ptr4;
```

Keep this in mind for structs as well. If one does not typedef them, then the pointer goes after the type.

```
struct person *ptr3;
```

Reading/Writing with pointers

Let's say that we declare a pointer `int *ptr`. For the sake of discussion, let's say that `ptr` contains the memory address `0x1000`. If we want to write to a pointer, we can dereference and assign `*ptr`.

```
*ptr = 0; // Writes some memory.
```

What C will do is take the type of the pointer which is an `int` and write `sizeof(int)` bytes from the start of the pointer, meaning that bytes `0x1000`, `0x1001`, `0x1002`, `0x1003` will all be zero. The number of bytes written depends on the pointer type. It is the same for all primitive types but structs are a little different.

Reading works roughly the same way, except you put the variable in the spot that it needs the value.

```
  int double = *ptr * 2
```

Reading and writing to non-primitive types gets a little tricky. You can't assign a pointer to an integer, you need to have the same type. In addition to that, the compilation unit – usually the file or a header – needs to have the size of the data structure readily available that means that opaque data structures can't be copied. Here is an example of assigning a struct pointer

```
#include <stdio.h>

typedef struct {
    int a1;
    int a2;
} pair;

int main() {
    pair obj;
    pair zeros;
    zeros.a1 = 0;
    zeros.a2 = 0;
    pair *ptr = &obj;
    obj.a1 = 1;
    obj.a2 = 2;
    *ptr = zeros;
    printf("a1: %d, a2: %d\n", ptr->a1, ptr->a2);
    return 0;
}
```

As for reading structure pointers, don't do it directly.

Pointer Arithmetic

In addition to adding to an integer, you can add an integer to a pointer. However, the pointer type is used to determine how much to increment the pointer. For char pointers, this is trivial because characters are always one byte.

```
char *ptr = "Hello"; // ptr holds the memory location of 'H'
ptr += 2; //ptr now points to the first'l'
```

If an int is 4 bytes then ptr+1 points to 4 bytes after whatever ptr is pointing at.

```
char *ptr = "ABCDEFGH";
int *bna = (int *) ptr;
bna +=1; // Would cause iterate by one integer space (i.e 4 bytes on some
    systems)
ptr = (char *) bna;
printf("%s", ptr);
/* Notice how only 'EFGH' is printed. Why is that? Well as mentioned
    above, when performing 'bna+=1' we are increasing the **integer**
    pointer by 1, (translates to 4 bytes on most systems) which is
    equivalent to 4 characters (each character is only 1 byte)*/
return 0;
```

Because pointer arithmetic in C is always automatically scaled by the size of the type that is pointed to, you can't perform pointer arithmetic on void pointers by POSIX standards though compilers will often treat the underlying type as `char`. You can think of pointer arithmetic in C as essentially doing the following

```
int *ptr1 = ...;
int *offset = ptr1 + 4;
```

Think

```
int *ptr1 = ...;
char *temp_ptr1 = (char*) ptr1;
int *offset = (int*)(temp_ptr1 + sizeof(int)*4);
```

To get the value. **Every time you do pointer arithmetic, take a deep breath and make sure that you are shifting over the number of bytes you think you are shifting over.**

So what is a void pointer?

A void pointer is a pointer without a type. Void pointers are used when either a datatype you're dealing with is unknown or when you're interfacing C code with other programming languages. You can think of this as a raw pointer, or just a memory address. You cannot directly read or write to it because the void type does not have a size. malloc by default returns a void pointer that can be safely promoted to any other type.

```
void *give_me_space = malloc(10);
char *string = give_me_space;
```

This does not require a cast because C automatically promotes `void*` to its appropriate type. **Note:**
`gcc` and `clang` are not total ISO-C compliant, meaning that they will let you do arithmetic on a void pointer. They will treat it as a `char *` pointer. Do not do this because it may not work with all compilers!

Common Bugs

Null Bytes

What's wrong with this code?

```
void mystrcpy(char*dest, char* src) {
  // void means no return value
  while( *src ) {dest = src; src ++; dest++; }
}
```

In the above code it simply changes the dest pointer to point to source string. Also the nuls bytes are not copied. Here's a better version -

```
  while( *src ) {*dest = *src; src ++; dest++; }
  *dest = *src;
```

Note it's also usual to see the following kind of implementation, which does everything inside the expression test, including copying the NUL byte.

```
  while( (*dest++ = *src++ )) {};
```

Double Frees

A double free error is when you accidentally attempt to free the same allocation twice.

```
int *p = malloc(sizeof(int));
free(p);

*p = 123; // Oops! - Dangling pointer! Writing to memory we don't own
    anymore

free(p); // Oops! - Double free!
```

The fix is first to write correct programs! Secondly, it's good programming hygiene to reset pointers once the memory has been freed. This ensures the pointer can't be used incorrectly without the program crashing.

```
p = NULL; // Now you can't use this pointer by mistake
```

Returning pointers to automatic variables

```
int *f() {
    int result = 42;
    static int imok;
    return &imok; // OK - static variables are not on the stack
    return &result; // Not OK
}
```

Automatic variables are bound to stack memory only for the lifetime of the function. After the function returns it is an error to continue to use the memory.

Insufficient memory allocation

```
struct User {
   char name[100];
};
typedef struct User user_t;

user_t *user = (user_t *) malloc(sizeof(user));
```

In the above example, we needed to allocate enough bytes for the struct. Instead, we allocated enough bytes to hold a pointer. Once we start using the user pointer we will corrupt memory. The correct code is shown below.

```
struct User {
   char name[100];
};
typedef struct User user_t;

user_t * user = (user_t *) malloc(sizeof(user_t));
```

Buffer overflow/ underflow

Famous example: Heart Bleed performed a memcpy into a buffer that was of insufficient size. Simple example: implement a strcpy and forget to add one to strlen, when determining the size of the memory required.

```
#define N (10)
int i = N, array[N];
for( ; i >= 0; i--) array[i] = i;
```

C does not check that pointers are valid. The above example writes into `array[10]` which is outside the array bounds. This can cause memory corruption because that memory location is probably being used for something else. In practice, this can be harder to spot because the overflow/underflow may occur in a library call. Here is our old friend gets.

```
gets(array); // Let's hope the input is shorter than my array!
```

Strings require strlen(s)+1 bytes

Every string must have a null byte after the last characters. To store the string "Hi" it takes 3 bytes: [H] [i] [\0].

```
  char *strdup(const char *input) {/* return a copy of 'input' */
    char *copy;
    copy = malloc(sizeof(char*));  /* nope! this allocates space for a
       pointer, not a string */
    copy = malloc(strlen(input));  /* Almost...but what about the null
       terminator? */
    copy = malloc(strlen(input) + 1); /* That's right. */
    strcpy(copy, input); /* strcpy will provide the null terminator */
    return copy;
}
```

Using uninitialized variables

```
int myfunction() {
  int x;
  int y = x + 2;
...
```

Automatic variables hold garbage or bit pattern happened to be in memory or register. It is an error to assume that it will always be initialized to zero.

Assuming Uninitialized memory will be zeroed

```
void myfunct() {
   char array[10];
   char *p = malloc(10);
```

Automatic (temporary variables) are not automatically initialized to zero. Heap allocations using malloc are not automatically initialized to zero.

Logic and Program flow mistakes

These are a set of mistakes that may or may not make sense within the context of the program.

Equal vs. Equality

Confusingly in C the assignment operator also returns the assigned value. Most of the time it is ignored. We can use it to initialize multiple things on the same line.

```
  int p1, p2;
  p1 = p2 = 0;
```

More confusingly, if we forget an equals sign in the equality operator we will end up assigning that variable. Most of the time this isn't what we want to do.

```
int answer = 3; // Will print out the answer.
if (answer = 42) {printf("I've solved the answer! It's %d", answer);}
```

The quick way to fix that is to get in the habit of putting constants first.

```
if (42 = answer) {printf("I've solved the answer! It's %d", answer);}
```

There are cases where we want to do it. A common example is getline.

```
while ((nread = getline(&line, &len, stream)) != -1)
```

This piece of code calls getline, and assigns the return value or the number of bytes read to nread. It also in the same line checks if that value is -1 and if so terminates the loop. It is always good practice to put parens around any assignment condition.

Undeclared or incorrectly prototyped functions

You may see code do something like this.

```
time_t start = time();
```

The system function 'time' actually takes a parameter a pointer to some memory that can receive the time_t structure or NULL. The compiler did not catch this error because the programmer did not provide a valid function prototype by including `time.h`.

More confusingly this could compile, work for decades and then crash. The reason for that is that time would be found at link time, not compile time in the c standard library which almost surely is already in memory. Since we aren't passing a parameter in, we are hoping the arguments on the stack (any garbage) is zeroed out because if it isn't, time will try to write the result of the function to that garbage which will cause the program to segfault.

Extra Semicolons

This is a pretty simple one, don't put semicolons when not needed.

```
for(int i = 0; i < 5; i++) ; printf("I'm printed once");
while(x < 10); x++ ; // X is never incremented
```

However, the following code is perfectly OK.

```
for(int i = 0; i < 5; i++){
    printf("%d\n", i);;;;;;;;;;;;;
}
```

It is OK to have this kind of code, because the C language uses semicolons (;) to separate statements. If there is no statement in between semicolons, then there is nothing to do and the compiler moves on to the next statement To save a lot

of confusion, just stick to always using braces. It increases the number of lines of code, which is a great productivity metric.

Topics

- C Strings representation
- C Strings as pointers
- char p[]vs char* p
- Simple C string functions (strcmp, strcat, strcpy)
- sizeof char
- sizeof x vs x*
- Heap memory lifetime
- Calls to heap allocation
- Dereferencing pointers
- Address-of operator
- Pointer arithmetic
- String duplication
- String truncation
- double-free error
- String literals
- Print formatting.
- memory out of bounds errors
- static memory
- fileio POSIX vs. C library
- C io fprintf and printf
- POSIX file IO (read, write, open)
- Buffering of stdout

Questions/Exercises

- What does the following print out?

```
int main(){
fprintf(stderr, "Hello ");
fprintf(stdout, "It's a small ");
fprintf(stderr, "World\n");
fprintf(stdout, "place\n");
return 0;
}
```

- What are the differences between the following two declarations? What does `sizeof` return for one of them?

```
char str1[] = "bhuvan";
char *str2 = "another one";
```

- What is a string in c?
- Code up a simple `my_strcmp`. How about `my_strcat`, `my_strcpy`, or `my_strdup`? Bonus: Code the functions while only going through the strings *once*.
- What should the following usually return?

```
int *ptr;
sizeof(ptr);
sizeof(*ptr);
```

- What is `malloc`? How is it different than `calloc`. Once memory is `malloc`ed how can I use `realloc`?
- What is the & operator? How about *?
- Pointer Arithmetic. Assume the following addresses. What are the following shifts?

```
char** ptr = malloc(10); //0x100
ptr[0] = malloc(20); //0x200
ptr[1] = malloc(20); //0x300
```

  - `ptr + 2`
  - `ptr + 4`
  - `ptr[0] + 4`
  - `ptr[1] + 2000`
  - `*((int)(ptr + 1)) + 3`
- How do we prevent double free errors?
- What is the printf specifier to print a string, `int`, or `char`?
- Is the following code valid? If so, why? Where is `output` located?

```
char *foo(int var){
static char output[20];
snprintf(output, 20, "%d", var);
return output;
}
```

- Write a function that accepts a string and opens that file prints out the file 40 bytes at a time but every other print reverses the string (try using POSIX API for this).
- What are some differences between the POSIX file descriptor model and C's `FILE*` (ie what function calls are used and which is buffered)? Does POSIX use C's `FILE*` internally or vice versa?

Rapid Fire: Pointer Arithmetic

Pointer arithmetic is really important! You need to know how many bytes each pointer is moved by with each addition. The following is a rapid fire section. We'll use the following definitions

```
int *int_; // sizeof(int) == 4;
long *long_; // sizeof(long) == 8;
char *char_;
int *short_; //sizeof(short) == 2;
int **int_ptr; // sizeof(int*) == 8;
```

How many bytes are moved over from the following additions?

1. `int_ + 1`
2. `long_ + 7`
3. `short_ - 6`
4. `short_ - sizeof(long)`
5. `long_ - sizeof(long) + sizeof(int_)`
6. `long_ - sizeof(long) / sizeof(int)`
7. `(char*)(int_ptr + sizeof(long)) + sizeof(int_)`

Rapid Fire Solutions

1. 4
2. 56
3. -12
4. -16
5. 0
6. -16
7. 72

Bibliography

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935.

[2] Tom Duff. Tom duff on duff's device. URL `https://www.lysator.liu.se/c/duffs-device.html`.

[3] Apple Inc. Xnu kernel. `https://github.com/apple/darwin-xnu`, 2017.

[4] ISO 1124:2005. ISO C Standard. Standard, International Organization for Standardization, Geneva, CH, March 2005. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[5] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN 9780131103627. URL `https://books.google.com/books?id=161QAAAAMAAJ`.

[6] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

[7] Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28(3):201–208, March 1993. ISSN 0362-1340. doi: 10.1145/155360.155580. URL `http://doi.acm.org/10.1145/155360.155580`.

Glossary

C A multipurpose programming language. 32

Linux Kernel A widely used operating system kernel. 32

Portable Works on multiple operating systems or machines. 32

POSIX Portable Operating System Interface, a set of standard defined by IEEE for an operating system. 32