3 The 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)

Note: This chapter is long and goes into a lot of detail. Feel free to gloss over parts with which you have experience in.

C is the de-facto programming language to do serious system serious programming. Why? Most kernels have their API accessible through C. The Linux kernel [7] and the XNU kernel [4] of which MacOS is based on are written in C and have C API - Application Programming Interface. The Windows Kernel uses C++, but doing system programming on that is much harder on windows that UNIX for novice system programmers. C doesn't have abstractions like classes and Resource Acquisition Is Initialization (RAII) to clean up memory. C also gives you much more of an opportunity to shoot yourself in the foot, but it lets you do things at a much more fine-grained level.

3.1 History of C

C was developed by Dennis Ritchie and Ken Thompson at Bell Labs back in 1973 [8]. Back then, we had gems of programming languages like Fortran, ALGOL, and LISP. The goal of C was two-fold. Firstly, it was made to target the most popular computers at the time, such as the PDP-7. Secondly, it tried to remove some of the lower-level constructs (managing registers, and programming assembly for jumps), and create a language that had the power to express programs procedurally (as opposed to mathematically like LISP) with readable code. All this 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 was with Brian Kernighan and Dennis Ritchie's book [6]. 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, though ISO largely won out as a language specification. We will be mainly focusing on is the POSIX C library which extends ISO. Now to get the elephant out of the room, the Linux kernel is fails to be POSIX compliant. Mostly, this is so because the Linux developers didn't want to pay the fee for compliance. It is also because they did not want to be fully compliant with a multitude of different standards because that meant increased development costs to maintain compliance.

We will aim to use C99, as it is the standard that most computers recognize, but sometimes use some of the

newer C11 features. We will also 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 fairly comprehensive overview of the language with language facilities. Feel free to gloss over if you have already worked with a C based language.

3.1.1 Features

- Speed. There is little separating a program and the system.
- Simplicity. C and its standard library comprise a simple set of portable functions.
- Manual Memory Management. C gives a program the ability to manage its memory. However, this can be a downside if a program has memory errors.
- Ubiquity. Through foreign function interfaces (FFI) and language bindings of various types, most other languages can call C functions and vice versa. 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.

3.2 Crash course introduction to C

The canonical way to start learning C is by starting with the hello world program. The original example that Kernighan and Ritchie proposed way back when hasn't changed.

```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 the return type of the function. The part before the parenthesis (`main`) is the function name. In C, no two functions can have the same name in a single compiled program, although shared libraries may be able. Then, the parameter list comes after. When we provide the parameter list for regular functions `(void)` that means that the compiler should produce an error if the function is called with a non-zero number of arguments. For regular functions having a declaration like `void func()` means that the function can be called like `func(1, 2, 3)`, because there is no delimiter. `main` is a special function. There are many ways of declaring `main` but the standard ones are `int main(void)`, `int main()`, and `int main(int argc, char *argv[])`.
3. `printf("Hello World");` is what a function call. `printf` is defined as a part of `stdio.h`. The function has been compiled and lives somewhere else on our machine - the location of the C standard library. Just remember to include the header and call the function with the appropriate parameters (a string literal `"Hello World"`). If the newline isn't included, the buffer will not be flushed (i.e. the write will not complete immediately).

4. `return 0`. `main` has to return an integer. By convention, `return 0` means success and anything else means failure. Here are some exit codes / statuses with special meaning: `http://tldp.org/LDP/abs/html/exitcodes.html`. In general, assume 0 means success.

```
$ 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".

If systems programming was as easy as writing hello world though, our jobs would be much easier.

3.2.1 Preprocessor

What is the preprocessor? Preprocessing is a copy and paste operation that the compiler performs **before** actually compiling the program. The following is an example of substitution

```
// Before preprocessing
#define MAX_LENGTH 10
char buffer[MAX_LENGTH]

// After preprocessing
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 a bounded depth where they need to stop. Macros are also simple text substitutions, without semantics. For example, look at what can happen if a macro tries to perform 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++ < 5 ? x++ : 5
```

In this case, it is opaque what gets printed out, but it will be 6. Can you try to figure out why? Also, consider the edge case when operator precedence comes into play.

```
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
```

There are also 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 consistently
```

What is wrong with the macro? Well, it works if a static array is passed in because **sizeof** a static array returns the number of bytes that array takes up and dividing it by the **sizeof(an_element)** would give the number of entries. But if passed 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.

3.3 Language Facilities

3.3.1 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, using 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 remain constant. If one tries to change a const variable, the program will fail to compile. `const` works a little differently when put before the type, the compiler re-orders 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-correctness.

```
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 construct. These loops execute the body and then check the condition at the bottom of the loop. If the condition is zero, the next statement is executed – the program counter is set to the first instruction after the loop. 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 will generate a unique number for that enum (within the context of the current enum) and use that for comparisons. The syntax to declare an instance of an enum is `enum <type> varname`. The added benefit to this is that the compiler 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. It is not advisable to rely on the compiler for consistent numbering, if you assign numbers. 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 program compiles on missing variable because the program will reference a variable in the system or another file.

```
// 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 equivalent to a while loop, but with differing syntax.

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

// Typically
```

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

As of the C89 standard, one cannot declare variables inside the `for` loop initialization block. 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

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

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

(a) Perform the initialization statement.

(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 statement.

(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, which is called spaghetti code. It is acceptable to use in some contexts though, for example, error checking code in the Linux kernel. 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. Note that an else is matched with the most recent if. A subtle bug related to a mismatched if and else statement, is the dangling else problem. 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 to moit the C function call procedure and "paste" the code in the callee. Instead, the compiler 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. Note that memcpy has undefined behavior when memory regions overlap. If this might be the case in your program, consider using memmove.

```
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 a 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 to a signed type such as below.

```c
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. When the compiler infers the type the following code changes as follows.

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

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

Which then the compiler is allowed to operate on further. The compiler must have a complete definition of the type at compile-time - not link time - or else you may get an odd error. Consider the following

```c
// 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 programmers 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. Additionally, if the compiler knows the full length of an array object, it will use that in the expression instead of having it decay into 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 startup not every time the program is run, and its lifetime is extended to that of the program.

```
// visible to this file only
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. C-structs are contiguous regions of memory that one can access specific elements of each memory as if they were separate variables. Note that there might be padding between elements, such that each variable is memory-aligned (starts at a memory address that is a multiple of its size).

```
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. Note that, the various cases of a switch statement fall through. It means that if execution starts in one case, the flow of control will continue to all subsequent cases, until a break statement.

```
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 goto statements, and you can put any 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 many 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 used with primitive int types (like `int` and `long`). There is a lot of behavior associated with unsigned arithmetic. For the most part, unless your code involves bit shifting, it isn't essential to know the difference in behavior with regards to unsigned and signed arithmetic.

21. `void` is a double meaning keyword. When used in terms of function or parameter definition, it means that the function explicitly returns no value or accepts no parameter, respectively. 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 an `lvalue`. 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 this 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 use the volatile keyword, the compiler is forced to keep the variable in and perform that check. This is useful for cases where you are doing multi-process or multi-threaded programs so that we can affect the running of one sequence of execution with another.

23. `while` represents the traditional `while` loop. There is a condition at the top of the loop, which is checked before every execution of the loop body. If the condition evaluates to a non-zero value, the loop body will be run.

3.3.2 C data types

There are many data types in C. As you may realize, all of them are either integers or floating point numbers and other types are variations of these.

1. `char` Represents exactly one byte of data. The number of bits in a byte might vary. `unsigned char` and `signed char` means 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 [5, 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.

If you want a fixed width integer type, for more portable code, you may use the types defined in stdint.h, which are of the form [u]int*width*_t, where u (which is optional) represents the signedness, and width is any of 8, 16, 32, and 64.

3.3.3 Operators

Operators are language constructs in C that are defined as part of the grammar of the language. These operators are listed in order of precedence.

- `[]` is the subscript operator. `a[n] == (a + n)*` where `n` is a number type and `a` is a pointer type.
- `->` is the structure dereference (or arrow) operator. If you have a pointer to a struct `*p`, you can use this to access one of its elements. `p->element`.
- `.` is the structure reference operator. If you have an object `a` then you can access an element `a.element`.
- `+/-a` is the unary plus and minus operator. They either keep or negate the sign, respectively, of the integer or float type underneath.
- `*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.
- `&a` is the address-of operator. This takes an element and returns its address.
- `++` is the increment operator. You can use it as a 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`.
- `--` is the decrement operator. This has the same semantics as the increment operator except that it decreases the value of the variable by one.
- `sizeof` is the sizeof operator, that is evaluated at the time of compilation. This is also mentioned in the keywords section.
- `a <mop> b` where `<mop> in {+, -, *, %, /}` are the arithmetic 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.
- `»/«` 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
```

Note that shifting by the word size (e.g. by 64 in a 64-bit architecture) results in undefined behavior.

- `<=/>=` are the greater than equal to/less than equal to, relational operators. They work as their name implies.
- `</>` are the greater than/less than relational operators. They again do as the name implies.
- `==/=` are the equal/not equal to relational operators. They once again do as the name implies.
- `&&` 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.
- `||` 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.
- `!` is the logical NOT operator. If the operand is zero, then this will return 1. Otherwise, it will return 0.
- `&` is the bitwise AND operator. If a bit is set in both operands, it is set in the output. Otherwise, it is not.
- `|` is the bitwise OR operator. If a bit is set in either operand, it is set in the output. Otherwise, it is not.
- `__` is the bitwise NOT operator. If a bit is set in the input, it will not be set in the output and vice versa.
- `?:` is the ternary / conditional 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`.
- `a, b` is the comma operator. `a` is evaluated and then `b` is evaluated and `b` is returned. In a sequence of multiple statements delimited by commas, all statements are evaluated from left to right, and the right-most expression is returned.

3.4 The C and Linux

Up until this point, we've covered C's language fundamentals. We'll now be focusing our attention to C and the POSIX variety of functions available to us to interact with the operating systems. We will talk about portable functions, for example `fwrite` `printf`. We will be evaluating the internals and scrutinizing them under the POSIX models and more specifically GNU/Linux. There are several things to that philosophy that makes the rest of this easier to know, so we'll put those things here.

3.4.1 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 this statement means is that everything is a file descriptor, which is an integer. For example, here is a file object, a network socket, and a kernel object. These are all references to records in the kernel's file descriptor table.

```
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 refers to an entry in a table of objects that the operating system picks and chooses from (that is, the file descriptor table). Objects can be allocated and deallocated, closed and opened, etc. The program interacts with these objects by using the API specified through system calls, and library functions.

3.4.2 System Calls

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

A system call is an operation that the kernel carries out. First, the operating system prepares a system call. Next, the kernel executes the system call to the best of its ability in kernel space and is a privileged operation. In the previous example, we got access to a file descriptor object. We can now also write some bytes to the file descriptor object that represents a file, and the operating system will do its best to get the bytes written to the disk.

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

When we say the kernel tries its best, this includes the possibility that the operation could fail for several reasons. Some of them are: the file is no longer valid, the hard drive failed, the system was interrupted etc. The way that a programmer communicates with the outside system is with system calls. An important 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.

3.4.3 C System Calls

Many C functions that will be discussed in the next sections are abstractions that call the correct underlying system call, based on the current platform. Their Windows implementation, for example, may be entirely different from that of other operating systems. Nevertheless, we will be studying these in the context of their Linux implementation.

3.5 Common C Functions

To find more information about any functions, please 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`

3.5.1 Handling Errors

Before we get into the nitty gritty of all the functions, know that most functions in C handle errors return oriented. This is at odds with programming languages like C++ or Java where the errors are handled with exceptions. There are a number of arguments against exceptions.

1. Exceptions make control flow harder to understand.
2. Exception oriented languages need to keep stack traces and maintain jump tables.
3. Exceptions may be complex objects.

There are a few arguments for exceptions as well

1. Exceptions can come from several layers deep.
2. Exceptions help reduce global state.
3. Exceptions differentiate business logic and normal flow.

Whatever the pros/cons are, we use the former because of backwards compatibility with languages like FORTRAN [3, P. 84]. Each thread will get a copy of **errno** because it is stored at the top of each thread's stack – more on threads later. One makes a call to a function that could return an error and if that function returns an error according to the man pages, it is up to the programmer to check errno.

```
#include <errno.h>

FILE *f = fopen("/does/not/exist.txt", "r");
if (NULL == f) {
    fprintf(stderr, "Errno is %d\n", errno);
    fprintf(stderr, "Description is %s\n", strerror(errno));
}
```

There is a shortcut function **perror** that prints the english description of errno. Also, a function may return the error code in the return value itself.

```
int s = getnameinfo(...);
if (0 != s) {
     fprintf(stderr, "getnameinfo: %s\n", gai_strerror(s));
}
```

Be sure to check the man page for return code characteristics.

3.5.2 Input / Output

In this section we will cover all the basic input and output functions in the standard library with references to system calls. Every process has three streams of data when it starts execution: standard input (for program input), standard output (for program output), and standard error (for error and debug messages). Usually, standard input is sourced from the terminal in which the program is being run in, and standard out is the same terminal. However, a programmer can use redirection such that their program can send output and/or receive input, to and from a file, or other programs.

They are designated by the file descriptors 0 and 1 respectively. 2 is reserved for standard error which by library convention is unbuffered (i.e. IO operations are performed immediately).

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.

For performance, `printf` buffers data until its cache 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` is a C library function, while `write` is a system call system.

The buffering semantics of printf is a little complicated. ISO defines three types of streams [5, P. 278]

- Unbuffered, where the contents of the stream reach their destination as soon as possible.
- Line Buffered, where the contents of the stream reach their destination as soon as a newline is provided.
- Fully Buffered, where the contents of the stream reach their destination as soon as the buffer is full.

Standard Error is defined as “not fully buffered” [5, P. 279]. Standard Output and Input are merely defined to be fully buffered if and only if the stream destination is not an interactive device. Usually, standard error will be unbuffered, standard input and output will be line buffered if the output is a terminal otherwise fully buffered. This relates to printf because printf merely uses the abstraction provided by the FILE interface and uses the above semantics to determine when to write. One can force a write by calling fflush() on the stream.

To print strings and single characters, use `puts(char *name )` and `putchar(char c )`

```
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’ or ‘stderr’) or a FILE pointer that was returned by `fopen` or `fdopen`. There is a printf equivalent that works with file descriptors, called dprintf. Just use `dprintf(int fd, char* format_string, ...);`.

To print data into a C string, use **sprintf** or better **snprintf**. **snprintf** returns the number of characters written excluding the terminating byte. We would use **sprintf** the size of the printed string is less than the provided buffer – think about printing an integer, it will never be more than 11 characters with the NUL byte. If printf is dealing with variadic input, it is safer to use the former function as shown in the following snippet.

```
// 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);
```

3.5.3 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. These functions treat stdin as a file from which we can read bytes. 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 the length being read, therefore buffers could get overrun easily. When this is done maliciously to hijack program control flow, this is known as a buffer overflow.

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

```
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);
```

Note that, unlike **gets**, **fgets** copies the newline into the buffer. 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 a function call failed using the errno convention. **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 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 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 system calls are expensive, the `scanf` family is much more expensive due to compatibility reasons. Since it needs to be able to process all of the printf specifiers correctly, the code isn't efficient TODO: citation needed. For highly performant programs, one should write the parsing themselves. If it is a one-off program or script, feel free to use scanf.

3.5.4 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 equal to the byte 0x00. More information about all of these functions. Any behavior missing from the documentation, such as the result of `strlen(NULL)` is considered undefined behavior.

- `int strlen(const char *s)` returns the length of the string.
- `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`. **This function assumes dest has enough space for src otherwise undefined behavior**
- `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 NUL 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.

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

```c
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 upped is changed to the following?

```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 strt`

  What these functions do is take the pointer to your string `*nptr` and a `base` (i.e. 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 passed an invalid number string, it will return 0. The caller has to be careful from a valid 0 and an error. This often involves an errno trampoline as shown below.

```c
int main(){
  const char *input = "0"; // or "!##@" or ""
  char* endptr;
  int saved_errno = errno;
```

```
    errno = 0
    long int parsed = strtol(input, &endptr, 10);
    if(parsed == 0 && errno != 0){
      // Definitely an error
    }
    errno = saved_errno;
    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 "This 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. Consider the safer version `memmove`.
- `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`?

3.6 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.

3.6.1 Structs

In low-level terms, a struct is 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 person;
```

We will often use the following typedef, so we can write use the struct name as the full type.

```
typedef struct contact contact;
```

```
contact person;

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.

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

All your compiler does is say "reserve this much space". Whenever a read or write occurs in the code, the compiler will calculate the offsets of the variable. The offsets are where the variable starts at. The phone variables starts at the 0x128th bytes and continues for sizeof(int) bytes with this compiler. **Offsets don't determine where the variable ends though**. Consider the following hack seen in a lot of kernel code.

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

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

// Let's convert to a c string
string* person;
person = malloc(sizeof(string) + length+1);
```

Currently, our memory looks like the following image. There is nothing in those boxes

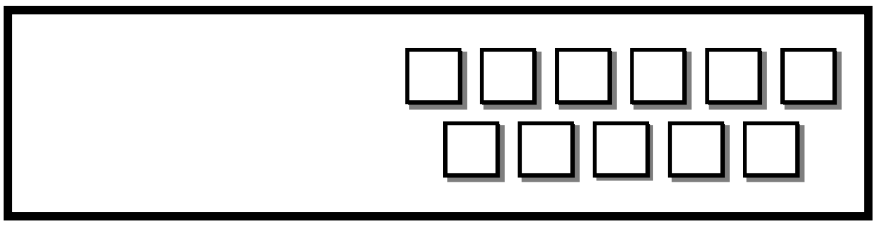

Figure 3.1: Struct pointing to 11 empty boxes

So what happens when we assign length? The first four boxes are filled with the value of the variable at length. The rest of the space is left untouched. We will assume that our machine is big endian. This means that the least significant byte is last.

```
person->length = length;
```

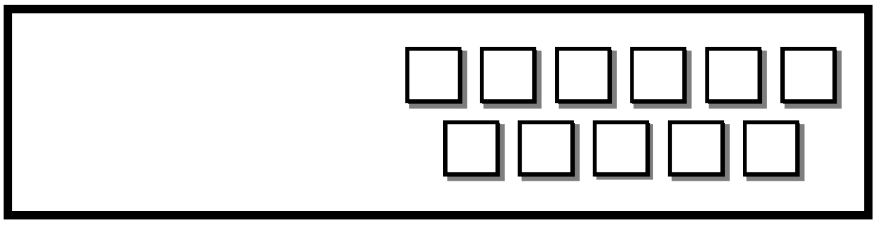

Figure 3.2: Struct pointing to 11 boxes, 4 filled with 0006, 7 junk

Now, we can write a string to the end of our struct with the following call.
m

```
strcpy(person->c_str, to_convert);
```

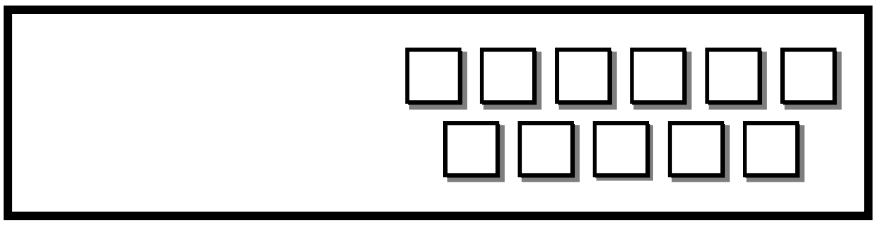

Figure 3.3: Struct pointing to 11 boxes, 4 filled with 0006, 7 the stirng “person”

We can even do a sanity check to make sure that the strings are equal.
m

```
strcmp(person->c_str, "person") == 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 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 parlor 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 is string manipulation.

3.6.2 Strings in C

In C, we have Null Terminated strings rather than Length Prefixed for historical reasons. For everyday programmers, remember to NUL terminate your string! A string in C is defined as a bunch of bytes ended by ‘’ or the NUL Byte.

3.6.3 Places for strings

Whenever you define a string literal - one in the form `char* str = "constant"` – that string is stored in the *data* section. Depending on your architecture, it 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, 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 has 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 NUL byte. C characters are always exactly one byte each.

String literals are constant

A string literal 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 = "Mark Twain likes books";
char *str2 = "Mark Twain 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 reside in different memory locations.

```
char arr1[] = "Mark Twain also likes to write";
char arr2[] = "Mark Twain 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 easily. Here is some boilerplate code to illustrate this.

```
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 to write within the bounds of the array. C does *not* do bounds checking at compile-time, but invalid reads/writes can get your program to crash.

```
strcpy(ary, "World"); // OK
strcpy(ptr, "World"); // NOT OK - Segmentation fault (crashes by
    default; unless SIGSEGV is blocked)
```

Unlike the array, however, we can 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)
```

Unlike pointers, that hold addresses to variables on the heap, or stack, char arrays (string literals) point to read-only memory located in the data section of the program. This means that pointers are more flexible than arrays, even though the name of an array is a pointer to its starting address.

In a more common case, pointers will point to heap memory in which case the memory referred to by the pointer **can** be modified.

3.7 Pointers

Pointers are variables that hold addresses. These addresses have a numeric value, but usually, programmers are interested in the value of the contents at that memory address. In this section, we will try to take you through a basic introduction to pointers.

3.7.1 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 delineates the semantics for pointer arithmetic (addition and subtraction).

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

Due to C's syntax, 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, ensure the `*` precedes the pointer.

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

Keep this in mind for structs as well. If one declares without a typedef, then the pointer goes after the type.

```
struct person *ptr3;
```

Reading / Writing with pointers

Let's say that `int *ptr` was declared. For the sake of discussion, let us assume that `ptr` contains the memory address `0x1000`. To write to the pointer, it must be dereferenced and assigned a value.

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

What C does 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 doubled = *ptr * 2;
```

Reading and writing to non-primitive types gets tricky. The compilation unit - usually the file or a header - needs to have the size of the data structure readily available. This 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. Instead, programmers create abstractions for creating, copying, and destroying structs. If this sounds familiar, it is what C++ originally intended to do before the standards committee went off the deep end.

3.7.2 Pointer Arithmetic

In addition to adding to an integer, pointers can be added to. However, the pointer type is used to determine how much to increment the pointer. A pointer is moved over by the value added times the size of the underlying type. 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) Because pointer arithmetic in C is always automatically scaled by the size of the type that is pointed to, POSIX standards forbid arithmetic on void pointers. Having said that, compilers will often treat the underlying type as `char`. Here is a machine translation. The following two pointer arithmetic operations are equal

```
int *ptr1 = ...;

// 1
int *offset = ptr1 + 4;

// 2
char *temp_ptr1 = (char*) ptr1;
int *offset = (int*)(temp_ptr1 + sizeof(int)*4);
```

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.

3.7.3 So what is a void pointer?

A void pointer is a pointer without a type. Void pointers are used when either the datatype is unknown or when interfacing C code with other programming languages without APIs. You can think of this as a raw pointer, or a memory address. `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;
```

C automatically promotes `void*` to its appropriate type. `gcc` and `clang` are not totally ISO C compliant, meaning that they will permit arithmetic on a void pointer. They will treat it as a `char` pointer. Do not do this because it is not portable - it is not guaranteed to work with all compilers!

3.8 Common Bugs

3.8.1 Nul 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 NUL bytes are not copied. Here is a better version -

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

Note that it is also common to see the following kind of implementation, which does everything inside the expression test, including copying the NUL byte. However, this is bad style, as a result of doing multiple operations in the same line.

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

3.8.2 Double Frees

A double free error is when a program 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 is a good habit to set pointers to NULL, once the memory has been freed. This ensures that the pointer cannot be used incorrectly without the program crashing.

```
p = NULL; // No dangling pointers
```

3.8.3 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.

3.8.4 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));
```

3.8.5 Buffer overflow/ underflow

A famous example: Heart Bleed performed a memcpy into a buffer that was of insufficient size. A 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 fails to check if 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!
```

3.8.6 Strings require strlen(s)+1 bytes

Every string must have a NUL 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;
}
```

3.8.7 Using uninitialized variables

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

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

3.8.8 Assuming Uninitialized memory will be zeroed

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

Automatic (temporary variables) and heap allocations may contain random bytes or garbage.

3.9 Logic and Program flow mistakes

These are a set of mistakes that may let the program compile but perform unintended functionality.

3.9.1 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("The answer is %d", answer);}
```

The quick way to fix that is to get in the habit of putting constants first. This mistake is common enough in while loop conditions. Most modern compilers disallows assigning variables a condition without parenthesis.

```
  if (42 = answer) {printf("The answer is %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 parentheses around any assignment condition.

3.9.2 Undeclared or incorrectly prototyped functions

Some snippets of code may do the following.

```
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 fails to catch this error because the programmer omitted the 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 a parameter isn't being passed, 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.

3.9.3 Extra Semicolons

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

```
for(int i = 0; i < 5; i++) ; printf("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, **always use braces.** It increases the number of lines of code, which is a great productivity metric.

3.10 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
- file input / output. POSIX vs. C library
- C input output: fprintf and printf
- POSIX file IO (read, write, open)
- Buffering of stdout

3.11 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[] = "first one";
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 each of the following lines usually return?

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

- What is `malloc`? How is it different from `calloc`. Once memory is allocated how can we 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? 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 path as a string, and opens that file, prints the file contents 40 bytes at a time but, every other print reverses the string (try using the POSIX API for this).
- What are some differences between the POSIX file descriptor model and C's `FILE*` (i.e. what function calls are used and which is buffered)? Does POSIX use C's `FILE*` internally or vice versa?

3.12 Rapid Fire: Pointer Arithmetic

Pointer arithmetic is important! Take a deep breath and figure out how many bytes each operation moves a pointer. 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_)`

3.12.1 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] Fortran 72. FORTRAN IV PROGRAMMER'S REFERENCE MANUAL. Manual, DIGITAL EQUIPMENT CORPORATION, Maynard, MASSACHUSETTS, May 1972. URL `http://www.bitsavers.org/www.computer.museum.uq.edu.au/pdf/DEC-10-AFDO-D%20decsystem10%20FORTRAN%20IV%20Programmer%27s%20Reference%20Manual.pdf`.

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

[5] 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`.

[6] 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`.

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

[8] 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`.