UMBC
TRAINING CENTERS

Intermediate C

TCPRG3001-2021-10-04

Table of Contents

1. Compilation Units . . . 1
   - Building Programs . . . 2
   - Header Files . . . 6
   - External Linkage . . . 9
   - Internal Linkage . . . 11
   - Exercises . . . 12
2. Basic Pointers . . . 13
   - Pointers are Memory Addresses . . . 14
   - Pointer Operators . . . 15
   - Pointer Declarations . . . 18
   - The `NULL` Pointer . . . 22
   - Pointers as Function Arguments . . . 23
   - `struct` Pointers . . . 25
   - `->` Operator . . . 27
   - Exercises . . . 29
3. Intermediate Pointers . . . 31
   - Pointer Return Values . . . 32
   - Pointer Lifetime . . . 33
   - Array Decay . . . 36
   - Pointer Arithmetic . . . 38
   - Pointers to Pointers . . . 41
   - `void *` . . . 43
   - Casting Pointers . . . 44
   - Function Pointers . . . 45
   - Exercises . . . 51
4. Basic Dynamic Memory . . . 53
   - `malloc`, `free` . . . 55
   - Valgrind . . . 62
   - Exercises . . . 70
5. Intermediate Dynamic Memory . . . 71
   - `memset` and `calloc` . . . 72
   - A Truly Dynamic Array . . . 74
   - `realloc` . . . 76
   - Arrays of pointers . . . 79
   - Exercises . . . 83
6. C Program Design . . . 85
   - `typedef` . . . 86
   - Opaque Types . . . 88
   - Application Programming Interfaces (APIs) . . . 91
   - Exercises . . . 94

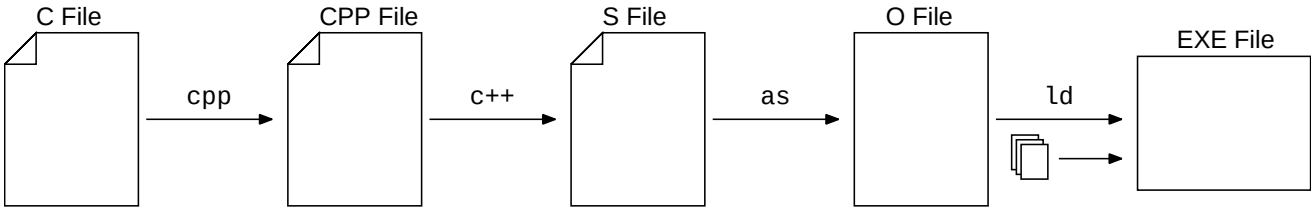
7. Unit Testing . . . 95
make check . . . 96
Check Organization . . . 97
assert() . . . 105
Exercises . . . 106
8. Bit manipulation . . . 109
Bitwise operators . . . 110
Bitfields . . . 113
Endianness . . . 118
Binary I/O . . . 122
Exercises . . . 129
9. Profiling and Optimization . . . 131
Gross Measurements . . . 132
Measuring with C Library Timing Functions . . . 133
Measuring with a Profile Build . . . 134
gprof . . . 134
callgrind . . . 137
Optimization . . . 140
Exercises . . . 144
10. Library Functions . . . 147
C vs. POSIX libraries . . . 148
stdlib.h . . . 149
qsort() . . . 154
bsearch() . . . 157
unistd.h . . . 160
getopt(), optarg, optind, opterr . . . 161
stdarg.h . . . 166
…, va_list, va_start, va_copy, va_arg, va_end . . . 167
time.h . . . 172
time, localtime, gmtime, timespec_get . . . 174
Exercises . . . 178

Chapter 1. Compilation Units

Few programs are small enough that they can be written in a single file. Many larger programs may take dozens, even hundreds of modules to build. It is important in C programming to understand how these larger programs are assembled.

Building Programs

Recall the build pipeline for C programs. The **preprocessor** performs simple textual substitutions, the **compiler** transforms the C code into architecture-specific assembly code, the **assembler** translates that assembly code one-to-one into machine code, and finally the **linker** combines code and libraries together to make an executable program.



The compiler has grown to be able to perform all these phases, often as a single step. And most of this complexity is abstracted away by using `make`.

```
$ ls
 Makefile   program.c
$ make program
cc -Wall -Wextra -Wpedantic -o program program.c
$ ls
 Makefile   program     program.c
```

This is totally sufficient for the majority of programs that fit in a single file. But code should be split into separate files or modules for maintainability and readability. Almost the entire build pipeline, however, operates on *one* file, or compilation unit, at a time. A *compilation unit* is a piece of code that passes through the pipeline and corresponds to one `.c` file.

Linking Multiple Compilation Units

The only stage that works with multiple compilation units is the linker. The linker will take any number of compilation units and link them together to form an executable program. Consider the following two files:

`program-ERROR.c`, *does not link without explicit* `make` *rule*

```c
#include <stdio.h>

int main(void)
{
    printf("Total amount: %d\n", calculate_amount());
}
```

library.c

```c
#include "library.h"

int calculate_amount(void)
{
    return 350 + 42;
}
```

Trying to build a program directly from these files will fail:

```
$ make program-ERROR
cc -Wall -Wextra -Wpedantic    program-ERROR.c   -o program-ERROR
program.c: In function ‘main’:
program.c:5:31: warning: implicit declaration of function ‘calculate_amount’ [-Wimplicit-
function-declaration]
    5 |  printf("Total amount: %d\n", calculate_amount());
      |                               ^~~~~~~~~
/usr/bin/ld: /tmp/ccRJtZMx.o: in function `main':
program-ERROR.c:(.text+0xe): undefined reference to `calculate_amount'
collect2: error: ld returned 1 exit status
make: *** [<builtin>: program] Error 1
```

But, it is important to note *how* it failed. It emitted one warning and one error. The warning happened during the **compile** phase. This can be asserted because the code is shown as part of the warning; only the preprocessor and compiler have access to the code as it is written. The error happened during the **link** phase. This is known because the message says `error: ld returned 1 exit status`, which (emphasis added) is the linker program.

The compiler warning states that there is an implicit declaration of `calculate_amount`. Recall that a function needs to be declared or defined before it can be invoked. `main` wants to call the `calculate_amount` function, but its compilation unit does not see it.

The linker errors states that there is a reference to the symbol `calculate` that is undefined. In other words, the definition of `calculate` cannot be found to build a complete program. Looking at the line emitted by `make` shows this to be clear. Only the `program.c` file is being compiled and linked.

This is because `make` does not know the relationships between the various source files, and must be told explicitly. To inform make that a given executable target uses multiple files, use a *target rule*. A target rule states the target desired, a colon, and then a list of files to build first. Any number of these prerequisites may be specified.

`Makefile` *defines target and its prerequisites*

```
CFLAGS += -std=c17

CFLAGS += -Wall -Wextra -Wpedantic
CFLAGS += -Wwrite-strings -Wfloat-equal -Wconversion
CFLAGS += -Waggregate-return -Winline

program-WARNING: program-WARNING.o library.o

program-UNPREFERRED: program-UNPREFERRED.o library.o

program-cached-WRONG: program-cached-WRONG.o library-cached-WRONG.o

program-cached: program-cached.o library-cached.o

program: program.o library.o

.PHONY: clean
clean:
    $(RM) *.o
```

There are two parts that make this work effectively. The first is that the name of the executable program matches a given input file. The second is that the prerequisites, those things to be built prior to the target, are all `.o` files. Running the build shows each of these steps:

```
$ make program-WARNING
cc -Wall -Wextra -Wpedantic   -c -o program-WARNING.o program-WARNING.c
program.c: In function ‘main’:
program.c:5:31: warning: implicit declaration of function ‘calculate_amount’ [-Wimplicit-
function-declaration]
    5 |  printf("Total amount: %d\n", calculate_amount());
      |                               ^~~~~~~~~
cc -Wall -Wextra -Wpedantic   -c -o library.o library.c
cc   program.o library.o   -o program
```

The compiler runs alone to build `program.o` and `library.o`. Building `program.o` results in a compiler warning, the same one seen earlier. But, the other compilation and the final linking happen with no error at all. Note that the linking (performed by the compiler) brings in multiple files to produce the program. The program is built and may be run.

```
$ ./program
Total amount: 392
```

Header Files

Solving the compiler warning ('implict declaration') requires a different strategy. The declaration of the function `calculate` must be made explicit. One way to solve this is to write the declaration in the file that will use it, `program.c`:

`program-UNPREFERRED.c`: *Unpreferred; Writing declaration in file where function is invoked*

```
#include <stdio.h>

int calculate_amount(void);

int main(void)
{
    printf("Total amount: %d\n", calculate_amount());
}
```

This is labor-intensive as well as error-prone. It is too easy to make a mistake copy and pasting a declaration, especially if the project has dozens or hundreds of source files. The better way to declare functions is to create a ***header file*** for the given compilation unit.

`program.c`: *Preferred; Including a header defined by the module*

```
#include <stdio.h>

#include "library.h"

int main(void)
{
    printf("Total amount: %d\n", calculate_amount());
}
```

Standard headers, such as `stdio.h`, get `#include`d using angle brackets (`<stdio.h>`). Custom headers, such as `library.h`, get `#include`d using quotation marks (`"stdio.h"`). These delimiters tell the preprocessor where to look for headers. In C, these headers **do not** have any executable code, **only** declarations.

`library.h`: *Preferred; a single file that holds declarations for a compilation unit*

```
#ifndef LIBRARY_H
#define LIBRARY_H

int calculate_amount(void);

#endif
```

A header file should hold declarations of all functions that the module makes available to its users. It should also hold declarations of any types that are needed by users of the module. If there are global variables exposed by the module, those should be declared as well, with `extern`. Finally, each header should have a *header guard* made up of some preprocessor directives.

Header Guard

The header guard is a set of three preprocessor directives: `#ifndef`, `#define`, and `#endif`. Together, they form a conditional block that wraps the entire contents of the header file. The goal of the header guard is to be read only *once* by the preprocessor. Remember that preprocessor directives are textual substitution, not program logic.

`#ifndef`: "If the following symbol is not defined"

The first preprocessor directive states that if a given symbol (`LIBRARY_H`) has not been defined for the preprocessor, then the conditional block of text should be emitted. If the symbol had already been defined, then the entire block (up to the `#endif`) would be skipped by the preprocessor. This, along with the second directive, ensures that the entire file is only seen *once* by any given compilation unit.

`#define`: "Define the following symbol"

The second directive defines the symbol for the preprocessor. It does not assign a value to the symbol, merely defines it to exist.

The symbol `LIBRARY_H` was chosen because it is similar to the filename, which tends to be a unique identifier in a given project. If the project is a library to be used in many projects, it may make sense to have a special library-specific prefix, along the lines of `CALC__LIBRARY_H`.

`#endif`: "End conditional block"

The third directive marks the end of the conditional started with `#ifndef`. This should be the last line in the file.

With a header guard in place, the given header will only be read once for any compilation unit, regardless of how many times it is `#include`d. While multiple `#include`s may seem easy to avoid, this is not the case when it comes to large projects. Especially when header files `#include` other header files to make use of others' types.

External Linkage

For types and functions, declarations are straightforward. But for variables, declarations need one additional part.

`library-cached-WRONG.h`: *Wrong; creates storage for* `cached_amount`

```
#ifndef LIBRARY_CACHED_WRONG_H
#define LIBRARY_CACHED_WRONG_H

extern int cached_amount;
extern int cached_amount;

int calculate_amount(void);

#endif
```

If a global variable (outside of any function) is named, the C compiler does two actions:

1. Reserves that name; and
2. Creates storage for it.

The problem arises when two compilation units both `#include` such a header. When compiled, they both create storage for that variable. At link time, these names are identical, and thus conflict with each other and fail to build.

Multiple storages created for a global variable

```
cc   program-cached-WRONG.o library-cached-WRONG.o -o program-cached-WRONG
/usr/bin/ld: library-cached-WRONG.o:(.data+0x0): multiple definition of `cached_amount';
program-cached-WRONG.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
make: *** [<builtin>: program] Error 1
```

The correct form would be to prevent the compiler from creating storage for the variable. This is accomplished with the `extern` keyword.

`library-cached.h`: *Correct; does not create storage for* `var_global`

```
#ifndef LIBRARY_CACHED_H
#define LIBRARY_CACHED_H

extern int cached_amount;

int calculate_amount(void);

#endif
```

The `extern` keyword tells the compiler that any storage needed for the symbol is ***external*** to the compilation unit, at least at that point in it. All functions declarations are `extern` by default, along with any type declarations. Variables are the exception, and require `extern` when declared, but not allocated.

Internal Linkage

Just as symbols can be `extern` and link externally, referencable by any compilation unit, they can also link *internally*, and only be referenced by the current compilation unit. This is accomplished via the `static` keyword.

The `static` keyword is twofold. When applied to storage *duration*, it means that a variable is only every initialized once. This is what allows `static` variables inside functions to retain their value between calls.

But, `static` also marks the linkage for a symbol as internal, and will not be handled by the linker. This allows for functions and variables to avoid conflicts with other libraries. This can be used on variables and functions to effectively mark them as "private".

`library-static.c`

```c
#include "library-static.h"

// this will only be usable internally
static int private_calculation(void)
{
    return 5*5*5*5*5;
}

// this will be usable externally
int calculate_amount(void)
{
    return private_calculation() + 42;
}
```

Exercises

Exercise 1

Write the following functions to fulfill the `main` function provided below.

Functions

- `add(int x, int y)`
- `subtract(int x, int y)`
- `divide(int x, int y)`
- `multiply(int x, int y)`

`starters/compilation/main.c`

```c
int main(void) {
      int a,b,c,d;
      a=add(5,6);
      b=subtract(10,5);
      c=divide(add(1,1),2);
      d=multiply(0,1);

      return a+b+c+d;
}
```

Put each of the four functions listed above in their own separate files—only have one file per function. Write six files: Ensure that `arithmetic.h` prototypes the 4 functions. Write a makefile that compiles all of these individually, then links them with `main.o` to form a program.

- `arithmetic.h`
- `add.c`
- `sub.c`
- `div.c`
- `mul.c`
- `Makefile`

Chapter 2. Basic Pointers

Pointers are Memory Addresses

Recall that the Most Important Picture is just a long array of bytes, one after the other. Every single byte in memory has an associated address, just like every box in a Post Office. That means one can talk about “the address 1234” or “the contents of box 1234”.

Just as every P.O. Box has a numerical address, so too does each byte in memory. This is advantageous because it allows a number of use cases:

- Multiple tasks can use the same ‘box’, reducing duplication.
- Large ‘packages’ can be referred to using their address.
- A ‘box’ can store a key to ***yet another*** ‘box’.
- Swapping ‘keys’ is faster than swapping ‘box’ contents.
- A ‘key’ can be shared to allow access to a particular ‘box’.

Pointers are addresses in memory, treated like a value.

Pointer Operators

There are two main unary operators when working with pointers.

`&`

: The *reference* or **the-address-of** operator

`*`

: The *dereference* or **the-thing-at** operator

The address of any data in memory can be found with `&`. Addresses are usually represented by unsigned integers, formatted in hexadecimal. The `printf` family of functions can print an address using the `%p` specifier.

`print-address.c`: *Get the address of a piece of data with* `&`

```
#include <stdio.h>

int main(void)
{
    int val = 17;
    printf("val = %d, &val = %p\n", val, &val);
}
```

On most systems, this will print something similar to:

```
val = 17, &val = 0x7ffc30f879d4
```

The exact address in memory will vary, but is likely a very large number in this case. This is because the `value` variable is on the Stack part of the Most Important Picture. In fact, this `&` operator allows great insight into the relative locations of the Most Important Picture.

`mip.c`: *The Most Important Picture locations*

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

int data = 17;

int main(void)
{
    int stack = 13;

    // This line has not be covered yet
    int *heap = malloc(sizeof(*heap));

    printf("Stack: %p\n", &stack);
    // Note the lack of & on the variable 'heap'
    printf("Heap: %p\n", heap);
    printf("Data: %p\n", &data);
    printf("Code: %p\n", &main);

    // This function has not be covered yet
    free(heap);
}
```

Not all of this code has been covered yet, particularly the `malloc` and `free` functions. Suffice to say that this program will print out addresses that belong to each of the four sections of the Most Important Picture.

The `&` operator can only work on variables stored in memory. It will not work on literal values, return values, or `register`s. It does work on something that would be on the left-hand side of an assignment, so these are sometimes called "lvalues".

WRONG: Cannot take address of literal value

```
int main(void)
{
    //ERROR: '4' is not a value in memory
    printf("%p\n", &4);
}
```

The reverse of the `&` operator is the `*` operator. Given an address, it will look up the value stored at that memory address.

print-address-silly.c: *& *cancel each other out*

```
#include <stdio.h>

int main(void)
{
    int val = 44;
    printf("val = %d, &val = %p\n, *&val = %d", val, &val, *&val);
}
```

```
val = 44, &val = 0x7ffc30f879d4, val = 44
```

The exact address in memory will vary from what is shown. Note that *& effectively cancel each other out. The & gets the address of a piece of data in memory, and * follows the address back to the data in memory and returns the value there.

Pointer Declarations

Pointers are addresses. Addresses are numeric values, such as `0x7ffc30f879d4`. This means that variables can be made that hold addresses. These variables are called pointer variables, or just pointers.

Declaring such a variable has unusual syntax.

```
int *irish_setter;
```

At first glance, this looks like the dereference operator! It is not. The rule for declaring a pointer variable is that the expression on the right *would evaluate to* the type on the left. In this case, the compiler reads that the expression `*irish_setter` should yield type `int`. The human interpretation of this is "`irish_setter` is a pointer to `int`".

> A common antipattern seen by C++ programmers writing C code is to write `int* x`, reading the declaration as "`x` is an `int` pointer." While C++ does greatly care about the *type* of data, C more highly values *expressions*. For more complex pointer types as explored in the next chapter, the C++ idiom will break down. Syntactically, these are identical to the compiler, so it becomes an issue for a style guide.

Once created, a pointer variable may hold an address.

`pointer-usage.c`

```c
#include <stdio.h>

int main(void)
{
    int value_in_memory = 77;
    int *irish_setter;

    // assign an address to the pointer
    irish_setter = &value_in_memory;

    printf("%d\n", *irish_setter); // Prints "77"
}
```

A pointer is only allowed to point to a specific type, specified in its declaration.

pointer-WRONG.c: *Pointer and data types do not match*

```
int main(void)
{
    double e = 2.71;
    int *pointer_to_integer;

    //ERROR: type mismatch
    pointer_to_integer = &e;

    printf("%lf\n", *pointer_to_integer);
}
```

However, there is no restriction on which instances a pointer may point to. A pointer may point at many different addresses throughout its lifetime.

pointer-usage-multiple.c: *Pointers may store the address of any instance of their type*

```
#include <stdio.h>

int main(void)
{
    double small = 1.0;
    double medium = 15.0;
    double large = 500.0;

    double *current;

    current = &small;
    printf("%lf\n", *current); // Prints "1.0"
    current = &medium;
    printf("%lf\n", *current); // Prints "15.0"
    current = &large;
    printf("%lf\n", *current); // Prints "500.0"
}
```

Like any other variable, a pointer variable can be initialized with a valid value.

`pointer-initialization.c`: *Pointers may be initialized*

```c
#include <stdio.h>

int main(void)
{
    int valid_value = 77;
    int *pointer = &valid_value;

    printf("%d\n", *pointer); // Prints "77"
}
```

Pointer variables also allow modification of the data being pointed at. The whole expression, `*px`, can be read from or assigned to. This affects the data being pointed at, ***not*** the pointer variable.

`pointer-updates.c`: *Pointers may be used to modify what they point at*

```c
#include <stdio.h>

int main(void)
{
    int valid_value = 77;
    int *pointer = &valid_value;

    printf("%d\n", *pointer); // Prints "77"
}
```

Nothing prevents multiple pointer variables from pointing to the same location in memory.

`pointer-aliases.c`: *multiple pointers may point to the same data*

```c
#include <stdio.h>

int main(void)
{
    int poison = 55;
    int *poison_for_kuzco = &num;
    int *kuzcos_poison = &num;

    printf("%d : %d\n", *kuzcos_poison, poison); // Prints "55 : 55"
    *poison_for_kuzco = 66;
    printf("%d : %d\n", *kuzcos_poison, poison); // Prints "66 : 66"
}
```

Since pointers are also values (the address), they can also be compared. Ordering pointers is not very useful, but comparing for equality is.

Comparing pointers by their address

```
void copy_int(int *dst, int *src)
{
    // Compares the addresses pointed at, not the data within
    if (dst == src) {
        fprintf(stderr, "Copying to itself!\n");
        return;
    }

    // Copies the data
    *dst = *src;
}
```

The NULL Pointer

One of the possible values of an address (effectively an unsigned integer) would be 0. This value is distinct because it is also the C value for Boolean `false`, while any other pointer would be nonzero, or `true`. As such, it is given a special name, `NULL`.

The `NULL` pointer should never be dereferenced. Doing so will result in trying to read from a forbidden place in memory, which will crash the program with a Segmentation Fault.

This means that any unknown pointer should be checked against `NULL` before it is used. So, every public function that receives pointers as arguments must check them against `NULL`.

WRONG: Dereferencing a pointer that is known to be null / a pointer that could be null.

```
// setting a pointer to null
int *pointer_to_integer = NULL;

...

// ERROR: Dereferencing a pointer that may be null.
printf("the integer within: %d", *pointer_to_integer);
```

Preferred: Checking whether or not a pointer is null prior to dereferencing it.

```
// setting a pointer to null
int *pointer_to_integer = NULL;

...

if (pointer_to_integer) {
    printf("the integer within: %d", *pointer_to_integer);
}
```

Note that a pointer in a Boolean expression is effectively testing it to see if it is `NULL`.

Pointers as Function Arguments

Since they are values, addresses can be passed in as arguments to functions. The type in the function signature must match the type of pointer, just as with other parameters. Note that modifying the data a pointer points at is a way for a function to change things outside the scope of itself. This is a very useful tool. It allows for data owned by the calling function to be updated. It does not violate the scope of the function, because the caller has passed that address to the function; the caller opts in to the update. Note that the functions must test for valid (non-`NULL`) pointers.

`pointer-argument.c`: *Pointer arguments can manipulate data outside the function's scope*

```c
#include <stdio.h>

void print_whats_inside_and_increment(int *num)
{
    if (!num) {
        printf("nothing inside!");
        return;
    } else {
        printf("%d", *num);
    }

    // Note that the contents of num are changed here
    *num += 1;
}

int main(void)
{
    int secret_number = 5;
    print_whats_inside_and_increment(&secret_number); // Prints "5"

    printf("%d\n", secret_number); // Prints "6"
}
```

Output Parameters

The fact that pointer variables can modify data in nonlocal places means they can be used as ways of making a function "return" multiple values. This is known as having "output parameters". The address passed is held by the calling function. The called function populates that location.

```
// assigns the input double to 'pos' and the opposite of the input double to 'neg'
void positive_and_negative(double input, double *pos, double *neg)
{
    *pos = input;
    *neg = 0 - input;
}
```

`struct` Pointers

When it comes to `struct`s, pointers have a lot of value. Consider some large `struct`s:

```
struct formula_one_car {
    int mass_kg;
    int racing_number;
    double current_fuel;
    double fuel_capacity;
    char class;
};

struct formula_one_driver {
    int mass_kg;
    int age;
    int podiums;
    int pole_positions;
    char name[64];
};
```

If a `struct formula_one_driver` was passed to a function, or returned from a function, the computer would spend the time to copy all of its fields to a new location in memory. (Including the 64 byte array for the driver name!) Remember that in a function call, each argument is assigned (or copied) to a parameter. In the case of a large `struct`, that means a lot of copying.

Unpreferred: Copying a `struct` for a function call

```
void formula_one_car_print(struct formula_one_car c);

int main(void)
{
    struct formula_one_car senna = { ... };

    // 'senna' is copied, field for field, into 'c' within the print function.
    formula_one_car_print(senna);
}
```

On the other hand, a *pointer* to a `struct` is a single, scalar value: the memory address. Copying a pointer is much faster than copying seven integers.

Preferred: Passing a `struct` pointer to a function call

```
void formula_one_car_print(struct formula_one_car *);

int main(void)
{
    struct formula_one_car senna = { ... };

    // here, we only copy the address of senna
    formula_one_car_print(&senna);
}
```

-> Operator

Working with `structs` presents a problem: member access.

WRONG: Compiler error, failing to derefence

```
void formula_one_car_print(struct formula_one_car *c)
{
    printf("mass in kg: %d, racing number: %d, fuel: %lf / %lf",
        c.mass_kg, c.racing_number, c.current_fuel, c.fuel_capacity);
}
```

This error happens because *pointers* do not have fields, `structs` have fields. The pointer must be dereferenced to produce the `struct` before its members are available via the `.` operator. But, dereferencing at first does not appear to solve anything:

WRONG: Compiler error, incorrect precedence

```
void formula_one_car_print(struct formula_one_car *c)
{
    printf("mass in kg: %d, racing number: %d, fuel: %lf / %lf",
        *c.mass_kg, *c.racing_number, *c.current_fuel, *c.fuel_capacity);
}
```

This error occurs because the `.` operator has higher precedence than the `*` operator. The above code snippet is parsed by the compiler as:

```
void formula_one_car_print(struct formula_one_car *c)
{
    printf("mass in kg: %d, racing number: %d, fuel: %lf / %lf",
        *(c.mass_kg), *(c.racing_number),
        *(c.current_fuel), *(c.fuel_capacity));
}
```

The solution to this syntax would be to ensure that the dereference happens first, using parentheses. This results in some slightly unwieldly syntax.

Unpreferred: Dereference before member access

```
void formula_one_car_print(struct formula_one_car *c)
{
    printf("mass in kg: %d, racing number: %d, fuel: %lf / %lf",
        (*c).mass_kg, (*c).racing_number,
        (*c).current_fuel, (*c).fuel_capacity);
}
```

Because member access through a pointer is so common, there is special syntax for it, the `->` operator. This is called the *arrow* operator. It is a shortened form of the dereference/member access syntax.

Preferred: `->` operator

```
void formula_one_car_print(struct formula_one_car *c)
{
    printf("mass in kg: %d, racing number: %d, fuel: %lf / %lf",
        c->mass_kg, c->racing_number, c->current_fuel, c->fuel_capacity);
}
```

This is especially noticeable when `struct` members are *themselves* pointers to other `struct`s. The arrow operator makes such chains of `struct` pointers readable.

`->` operator chaining

```
struct f1_team {
    char *name;
    struct formula_one_car *lead_car;

    ...
}

void f1_lead_car_print(struct f1_team *scuderia)
{
    printf("%d", scuderia->lead_car->mass_kg);
}
```

Exercises

Exercise 1 — `swap.c`

Write a function `void swap(int *a, int *b)` that swaps the values pointed at by its arguments.

Exercise 2 — `sumproduct.c`

Write a function `void sumproduct(int a, int b, int *sum, int *product)`. It should compute the sum of `a + b` and place it in `sum`, while the product `a * b` is placed in `product`. Only non-`NULL` pointers should be filled.

Exercise 3 — `quadratic.c`

Write a function `int quadratic(double a, double b, double c, double *first, double *second)`. It should compute the quadratic formula using inputs `a`, `b`, and `c`. It should return the number of real roots found (0, 1, or 2). The values of the roots should be placed in `first` and `second`. The function should still return the correct value even if `NULL` pointers are passed in.

Exercise 4 — `furniture.c`

Using the following structure definition:

```
struct furniture {
    char *name;
    int width_cm;
    int height_cm;
    int length_cm;
    int weight_kg;
};
```

Implement the following library functions:

```
// Returns the volume in m**3
double furniture_volume(const struct furniture *f);

// Returns the average density in kg/m**3
double furniture_average_density(const struct furniture *f);

// Prints; e.g. "sideboard, 60cm x 182cm"
void furniture_print(const struct furniture *f);
```

Put the structure definition and the function prototypes into a header file. What should be returned if the

parameter is `NULL`?

Chapter 3. Intermediate Pointers

Pointer Return Values

The same cost of passing large parameters to a function applies to a function's return value.

Unpreferred: Copying a `struct` for a return value

```
struct f1_driver f1_driver_create(...);

int main(void)
{
    // return value is copied, field for field, to 'senna'
    struct f1_driver senna = f1_driver_create(...);
}
```

Preferred: Returning a `struct *` from a function

```
struct f1_driver *f1_driver_create(...);

int main(void)
{
    // return pointer is copied to 'senna'
    struct f1_driver *senna = f1_driver_create(...);
}
```

However, it is vital to understand the lifetime of an object that is pointed at.

Pointer Lifetime

A pointer is a value, the value of the memory address. As long as that memory address represents the object pointed at, the pointer functions normally. But, it is possible for a pointer to live longer than the object does! Recall the Most Important Picture, and how the stack works.

The stack grows for each function call to allow for function-local variables and storage. When the function returns, the stack shrinks, and that storage may be reclaimed and edited by a different function call.

WRONG: Returning a pointer to a function-local object

```
struct f1_driver *f1_driver_create(...)
{
    struct f1_driver hunt = {...};

    //ERROR: hunt's lifetime is over when the function returns.
    return &hunt;
}

int main(void)
{
    // undefined behavior! Explained below
    struct f1_driver *hunt = f1_driver_create(...);
}
```

Confusingly, the program may seem to still work normally. This is because the area of the stack where the local `struct` was created *might* not be overwritten. *Might* is not the same as "will not".

Storage for data returned from a function needs to last longer than the function call, guaranteed. This can be achieved using a different area of the Most Important Picture. There are three ways to do this:

1. Lower on the Stack than the caller
2. Static Data
3. Heap Data

Lower On the Call Stack Than the Caller

Use When

1. Only a few instances are needed; AND
2. The object does not need to live longer than the calling function

The most readily available location for storage that persists across a function call is creating it inside a

function lower on the stack.

Returning a pointer lower in the call stack

```
void f1_driver_print(struct f1_driver *f);

int main(void)
{
    // 'senna' is on the stack frame for 'main'
    struct f1_driver senna = {...};
    f1_driver_print(&senna);
}

// 'd' is a pointer to data inside the 'main' function
void f1_driver_print(struct f1_driver *d)
{
    printf(...);
    // After closing bracket, d is reclaimed, but not its contents
}
```

There are numerous times when this model is used. It does have some drawbacks.

To begin with, the stack is limited in size. Building thousands of instances will quickly exhaust the stack space. Further, as each function stack frame is of a fixed size, it cannot be scaled up to arbitrary input.

Static Storage Duration

Use When

1. Passing around a pointer is more efficient than passing around the structure; AND
2. Only one instance of the data is needed

If a pointer is pointing to an area of memory that is only ever used for one purpose, that would be a way of assuring that the lifetime of the pointer and its referent are in lockstep. This would be a good use of the Data section of the Most Important Picture.

Recall that the Data section is fixed in size. It is used to store global data, not tied to any function call. It also stores any variables marked `static`; hence, the name “static storage duration” for these data.

Since its lifetime is the lifetime of the entire program, a pointer into this section of memory will continue to be valid.

Returning a pointer to the Data section

```
struct driver_roster *driver_listing(...)
{
    static driver_roster dr = {...};
    ...
    return &dr;
}
```

The structure could be a global variable, but more likely would be scoped to a function to prevent direct access or manipulation. The downside of this approach is that there can only ever be one instance of the data. In some cases, that may be a benefit, such as with certain resources such as database connections. But, for making a number of such objects, static storage duration will be insufficient.

Heap Storage Duration

Use When

1. Many instances are needed; OR
2. The object must live longer than the current function

The heap is an area that can grow far larger than the stack. Using this will be covered in the next chapter.

Array Decay

A similar argument to passing `struct`s would be passing arrays. An array could be thousands of items long. Unlike `struct`s, arrays are *always* passed to functions as pointers. This is known as **array decay to a pointer**. This means that for any array, `&array == array`.

Passing an array to a function call

```
void array_print(int *arr);

int main(void)
{
    int v[] = { ... };
    // The 'value' of an array is its address
    printf("v == &v → %d", v == &arr); // Prints "1"
    // The address of 'v' is copied to parameter 'arr': arr = v;
    array_print(v);
}
```

This means that using `[]` in a function signature for parameters is identical to using `*`. Even if the number of elements is specified for a parameter, the compiler cannot confirm that amount, and the type of the array decays to a pointer. The use of `[]` or `[n]` in a function signature is for informative purposes, to developers using the function, rather than proscriptive to the compiler.

```
// All these function declarations are identical
void array_print(int a[]);
void array_print(int a[7]);
void array_print(int *a);
```

This decay is why any function that takes an array as an argument must also pass in the size of the array (or have a special terminator element at the end). This is cleary demonstrated by the `sizeof` compile-time operator:

`array-decay.c`: *Arrays passed to functions decay to pointers in size*

```c
#include <stdio.h>

void array_print_size(int a[3])
{
    printf("Size in array_print_size: %zu\n", sizeof(a));
}

int main(void)
{
    int five_values[] = {0, 1, 2, 3, 4};

    // prints address size
    array_print_size(five_values);

    // prints total size occupied by array elements
    printf("Size in main: %zu\n", sizeof(five_values));
}
```

Pointer Arithmetic

The fact that arrays decay to pointers reveals another feature of pointers: arithmetic may performed on pointer values. Pointers can be treated like arrays, and retrieve a specific index.

Requesting specific elements from a pointer to an array

```
int main(void)
{
    int odds[] = {1, 3, 5, 7, 9};
    // request element 2 down from the start of odds array
    printf("%d", *(odds+2)); // Prints "5"
}
```

Consider an array `a`.

Recall that the index into an array is better described as an 'offset' from the start of the array. `a[1]` is the object that is offset from the start of the array by `1 * sizeof(a[0])`. Similarly `a[2]` is the object that is offset from the start of the array by `2 * sizeof(a[0])`. In other words, `a[n]` is the object offset from the start of the array by `n * sizeof(a[0])`.

This means that the expression `*a` is the same as `a[0]`. `a` is where the array starts in memory. `a[0]` is the object at the start of the array. `*a` is the object at the start of the array.

C further extends this idea by allowing arithmetic on pointers. Doing so 'moves' the address lookup by the `sizeof` of the pointed-at object.

Pointer arithmetic moves by `sizeof(*argv)` each

```
void array_print(int a[], size_t sz)
{
    for (int n=0; n < sz; ++n) {
        printf("a + %d -> %p\n", n, a + n);
    }
}
```

In the above sample code, `sizeof(*a)` is an `int`, so 4 bytes on this platform. The address of each object in the array are offset by that much. Memory addresses will be output from this function, in a sequential fashion.

Since the expression `a + 2` is still a pointer, it can be dereferenced: the result is the same as `a[2]`.

Pointer arithmetic moves by `sizeof(*argv)` each

```
void array_print(int a[], size_t sz)
{
    for (int n=0; n < sz; ++n) {
        printf("*(a + %d) -> %d\n", n, *(a + n));
    }
}
```

In C, `a[n]` is a syntactic shortcut for `*(a + n)`. Array syntax is a stand-in for pointer arithmetic.

This also yields the bizarre result that `a[n] == n[a]`.

Because pointers are values, those values may be manipulated. It is common to add to or subtract from a pointer to move through an array.

Manipulating a pointer via arithmetic operations

```
void print_every_other(const char *s)
{
    if (!s) {
        return;
    }

    while (*s) {
        // print the character at the start
        printf("%c\n", *s);

        Confirm that the next char can be skipped
        if (s[1] == '\0') {
            break;
        }

        // Skip a character and move s forward by two.
        // This will print every other char
        s += 2;
    }
}
```

Note that, in the above code, the ***string*** is not changed at all. Rather, the function-local pointer `s` is changed. Since the function does not need to track the entire array (but instead only the current position), `s` can be manipulated without fear.

One of the factors into this working is that the end of a string is the NUL byte, `'\0'`. This evaluates to `false`,

thus providing an easy test to end the loop. The code could have been written with a regular `for` loop over the underlying array. This is not preferred for arrays that have a "final zero" element, such as strings. Writing and reading pointer arithmetic code is part of C literacy and is idiomatic.

Unpreferred: Walking an array instead of pointer arithmetic.

```
void interleave_string(const char *s)
{
    if (!s) {
        return;
    }

    for (size_t i=0; s[i] != '\0'; ++i) {
        // print the character at the start
        printf("%c\n", s[i]);

        ++i; // Add one more to i, to skip by two.
        if (s[i] == '\0') {
            break;
        }
    }
}
```

Recall that arrays do not have bounds checking; neither too does pointer arithmetic. The only stop to calling `s += 50000` is the attentiveness of the developer.

Pointers to Pointers

Pointers allow memory addresses to be treated like values. Any value can be stored in memory. Therefore, a pointer can be stored in memory as well, and **its** address taken. This would be a pointer *to a pointer* to a value.

This is actually quite beneficial for a number of reasons. One use case is having an array of pointers to `struct f1_driver`s. Due to array decay, this type becomes `struct f1_driver **`.

```
void print_team_drivers(struct f1_driver **teams, size_t sz);

int main(void)
{
    struct f1_driver kimi = { ... };
    struct f1_driver seb = { ... };
    struct f1_driver *scuderia[] = { &kimi, &seb, ... };

    ...
    // As many as desired can be added

    print_team_drivers(scuderia, ...);
}
```

Note that an array of pointers is much easier to manipulate than an array of objects. Swapping two elements in an array is fast and easy if those elements are pointers, but more complicated if they are objects with their own fields.

An additional use case of pointers to pointers is using a pointer as an output parameter. Consider the signature of the `strtod` function:

```
double strtod(const char *nptr, char **endptr);
```

Given a string `nptr`, `strtod` will try and convert it to a `double`. The output parameter `endptr` is always passed the address of a pointer variable. The function can set `*endptr` to the location of the first character in `nptr` that is not part of the number being parsed.

strtod-usage.c

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

int main(int argc, char* argv[])
{
    if (argc != 2) {
        printf(stderr, "Use arguments appropriately");
        return 1;
    }
    char *err;

    double v = strtod(argv[1], &err);
    if (*err) {
        fprintf(stderr, "Invalid input -> %s, argv[1]");
        return 1;
    }

    printf("value of v -> %lf\n", v);
}
```

void *

One of the benefits of modern systems is that a pointer is just the size of an address in memory. This also means that, on the vast majority of systems, all pointers are the same size. This could allow for mixing together different types of objects in the same array.

A `void *` is a special kind of pointer whose pointed-at type is unspecified or unknown. This allows any such pointer to be converted to a pointer of any type.

```
{
    char *name = "Lewis";

    void *anything = name;

    char *world_champ = anything;
}
```

The `NULL` pointer is actually a `void *`. This is why it can be assigned as a value for any pointer; its type is implicitly converted.

`NULL` is a `void *` and can be converted to any type

```
int main(void)
{
    char *track_name = NULL;
    int *track_length = NULL;
    ...
}
```

Casting Pointers

In the same vein as `void *`, pointers can be cast from one to another in a straightforward manner. This does not mean that doing so is often a good idea.

Unpreferred: Changing types of pointers

```
int main(void)
{
    struct driver *pierre = driver_create(...);

    // Casting one pointer type to a different pointer type
    struct pilot *p;
    p = (struct pilot *)pierre;

    // Overlaps at the same point in the `struct` definitions
    printf("%d\n", p->age); // Prints "25"

    //ERROR: Actually uses the `length_steering_wheel` field
    printf("%d\n", p->miles_flown); // Prints "1058"
}
```

In the above code, the variable `p` is now a kind of interface to `pierre`, but through a different `struct` definition! Because the two types are very similar in structure, many fields appear to be the same. But, any fields that differ can easily lead to errors. Since casting is the developer asserting to the compiler "I know better", the compiler will not issue a warning, and this error could become silent. If this is truly desired, it may be better to use a `union` rather than casting pointers.

Casting pointers can also add or remove `const` qualifiers, if needed.

Function Pointers

Recall that a function exists in the Code section of the Most Important Picture. This means that functions also have addresses in memory where they are located. As such, there can be pointers that point to functions. Their syntax is slightly more complex.

The name of a function is also a pointer to its location in memory.

Function names are pointers

```
int main(void)
{
    printf("%zu\n", sizeof(main)); // Prints "8"
    // The 'value' of a function is its address
}
```

Declaring a variable that will hold a pointer to a function requires an extra set of parentheses.

Declaring a pointer to a function

```
int *function(double, int);
int (*function_ptr)(double, int);
```

Without the parentheses, this is parsed as "`actual_function` is a function taking two `int`s and returning a pointer to an `int`". With the parentheses, this is parsed as "`pointer_to_function` is a pointer to a function taking two `int`s and returning an `int`". The full type of a function pointer is the types of the parameters and the return type.

WRONG: Function pointer types must agree on all parameters and return types

```
int multiply(double a, double b)
{
    return a * b;
}

//ERROR; types of 'multiply' and 'op' do not match
int (*op)(int, int) = multiply;
```

Function pointers allow for more generic code to be written to solve problems. Invoking a function through a pointer is the same syntax as invoking a function.

Using function pointers

```
enum operation { SHIFT_UP, SHIFT_DOWN };

size_t upshift(size_t, size_t);
size_t downshift(size_t, size_t);

int main(void)
{
    ...
    // ensure type agreement
    int (*shift)(size_t, size_t);

    // decide what shift to do based on current rpm
    switch (rpm) {
    case SHIFT_UP:
        shift = upshift;
        break;
    case SHIFT_DOWN:
        shift = downshift;
        break;
    }

    // invoke function pointed to by 's'
    size_t new_gear = shift(curr_gear, 1);
    ...
}
```

Function pointers are extremely useful as parameters to more complex functions. For instance, consider a function that can print an object. Consider another function that can print an array of objects (`void **`). This means that the array-printing function would need to know how to print an individual object via a function pointer parameter.

Using function pointers to reduce repetition

```
// Two different print functions called in simple steps.
void array_print(void **arr, size_t arr_size, void (*printer)(void *))
{
    for (size_t i=0; n < arr_size; ++i) {
        printer(arr[i]);
    }
}

void car_print(void *arg)
{
    struct car *c = arg;
    ...
}

void driver_print(void *arg)
{
    struct driver *d = arg;
    ...
}

int main(void)
{
    struct car *lineup[] = ...
    struct driver *drivers[] = ...

    // Prints the cars in 'lineup'
    team_print(lineup, sizeof(lineup)/sizeof(lineup[0]), car_print);

    // Prints the drivers in 'drivers'
    team_print(drivers, sizeof(drivers)/sizeof(drivers[0]), driver_print);
}
```

This is much simpler than writing a `team_cars_print` and `team_drivers_print`, which would result in a lot of unnecessarily copied code.

Note that both the car and driver print functions utilized void pointers in their input parameters.

This is because function pointer type signatures must match exactly; there is no implicit conversion allowed on either its parameters or return type.

The Spiral Rule

It can be difficult to parse out what the type of a given declaration is.

Confusing declarations

```
int *kimi[3];
int (*romeo)(int *inp, size_t len);
void (*bagels(int win, void (*cars)(int)))(int);
```

Remember that declarations of pointers always result with the leftmost type being the type of the expression to its right. But, attempting to parse these declarations in an English-readable way greatly benefits from the ***Spiral Rule***, first coined by David Anderson in 1994.

1. Begin with the unknown identifier.
2. Move in a clockwise spiral from the identifier, reading off each piece of syntax.
3. Resolve the entirety of a parenthetical before moving outside of it.

Parsing `int *kimi[3]`

```
    +-------+
    | +--+  |
    | ^  v  |
int *kimi[3];
 ^  ^    |  |
 |  +----+  |
 +----------+
```

1. The name `kimi` is...
2. an array of 3...
3. pointers to...
4. an `int`

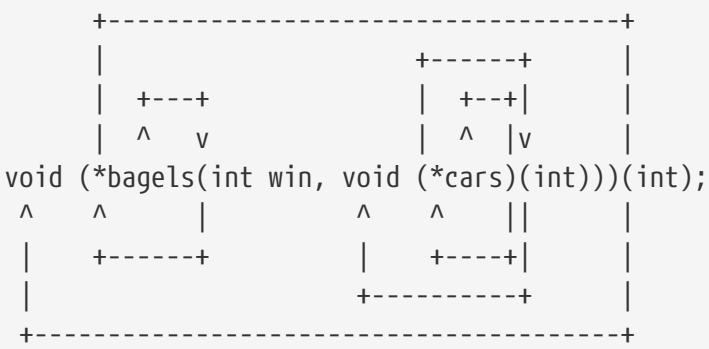
Parsing `int (romeo)(int *inp, size_t len)`

```
    +-------+    +----+
    |   +--+|    | +-+|
    |   ^  ||v   | ^ ||
int (*romeo)(int *inp, size_t len);
 ^   ^     || ^  ^   ||
 |   +-----+| |  +---+|
 |          | +-------+
 |          |
 +----------+
```

1. The name `romeo` is...
2. a pointer to...
3. a function that takes...
    a. the name `inp`, which is...
    b. a pointer to...
    c. an `int`...
    d. and `len`, which is...
    e. a `size_t`...
4. returning `int`

Even complex declarations involving function pointers can be parsed using the Spiral Rule. Note that the spiral rule can be applied recursively.

Parsing `void (*bagels(int win, void (*cars)(int)))(int)`



1. The name `bagels` is...
2. a function that takes...
    a. the name `win`, which is...
    b. an `int`...
    c. and `cars`, which is...
    d. a pointer to...
    e. a function that takes...
        i. an `int`
        ii. returning `void`
3. returning a pointer to...
4. a function that takes...
    a. an `int`
    b. returning `void`

Exercises

Exercise 1

Write a one-liner C expression to fulfill the following description, based on the spiral rule explained above.

1. The name `lando` is...
2. an array of 4...
3. pointers to...
4. a `char`

Exercise 2

Write a one-liner C expression to fulfill the following description, based on the spiral rule explained above.

1. The name `carlos` is...
2. a function that takes...
    a. the name `sainz`, which is...
    b. a `char`...
    c. and `junior`, which is...
    d. a pointer to...
    e. a function that takes...
        i. a name `jamon` which is...
        ii. an `int`...
        iii. and `norris` which is...
        iv. a 'char'...
        v. returning `void`
    f. returning `void`

Exercise 3

Write a one-liner C expression to fulfill the following description, based on the spiral rule explained above.

1. The name `charles` is...
2. an array of 10...

3. pointers to…
4. an `int`

Exercise 4

Write a one-liner C expression to fulfill the following description, based on the spiral rule explained above. Recall that a pointer to an array is simply a pointer to a pointer.

1. The name `binotto` is…
2. a pointer to…
3. a function that takes…
    a. the name `mattia`, which is…
    b. a pointer to…
    c. an array of `ints`…
    d. and `sebastian`, which is…
    e. a `char`…
4. returning `int`

Exercise 5 – `calculator.c`

Write a simple integer calculator that uses function pointers to dispatch its operations of addition, subtraction, multiplication, and division.

Chapter 4. Basic Dynamic Memory

Pointers are a powerful tool. They are fast ways of communicating large objects around in memory.

But, they are also restricted; pointers to Stack objects become invalid when the Stack unwinds. Pointers to Data segment objects are fixed in size and cannot respond to user requests.

These shortcomings are solved by using the Heap. While the Stack is restricted in size, the Heap is not. It can be used to allocate thousands of objects. Heap objects can live as long as the program does, or only for a few function calls as desired.

malloc, free

```
#include <stdlib.h>


void *malloc(size_t size);
```

The `malloc` function allocates memory from the Heap. Its argument is the number of bytes to allocate. It returns a `void *` to the start of that section of Heap memory. Since it is `void *`, it can be assigned to any type of pointer. Since it is allocated on the Heap (rather than the Stack), there is more room for additional objects. It is found in the `stdlib.h` header.

Using `malloc`

```
int main(void)
{
    // Allocates an array of size BUF_SIZE
    char *buf = malloc(BUF_SIZE);


    // Always Be Checking to see if malloc has failed.
    if (!buf) {
        perror("malloc() failed");
        return 1;
    }

    // Perform any operation desired
    printf("Enter name: ");
    if (!fgets(buf, BUF_SIZE, stdin)) {
        ...
    }

    // always free the allocated memory before exiting
    free(arr);
}
```

This version, using the Heap rather than the Stack, can be preferable because a large string buffer (`arr`) is no longer using up precious Stack space.

The `malloc` function is a library function that manages space on the Heap. It keeps track of which areas are being used and which are available. When the program calls `malloc()`, the function checks for and returns an available piece of Heap memory. It may occasionally increase the size of the Heap for large allocations, but there is far more memory available on the Heap than anywhere else in the Most

Important Picture.

If `malloc` is unable to allocate enough memory, it will return `NULL`. Like other functions that may fail, this should always be checked for. Often, the right thing to do in such a situation is exit the program. If a program can no longer get Heap memory, something catastrophic has likely occurred. At the very least, it could be a bug in how much memory is requested (a negative amount requested, e.g.).

```
#include <stdlib.h>


void free(void *ptr);
```

One of the most important things to do after calling `malloc` is to call `free`. `malloc` allocates memory from the Heap, and `free` marks that allocated memory as reusable again. Failing to do so is known as a *memory leak*.

If a function allocates an object, and that object is never freed, it will continue to take up space in the Most Important Picture. When such a function is called multiple times, more space will be used by the process. This is a drain on the resources of the system and tends to lead to "out of memory" errors. Every `malloc` allocation must have a corresponding call to `free`.

All memory must be relinquished back to the Heap before a program exits.

Allocation Expressions

There are many possible errors that can be made with Heap allocation. Many of these can be mitigated with good practices.

An extremely common error is to request the wrong amount of memory. Requesting too much is a resource drain. Requesting too little is far worse: it means that the program can overrun the bounds set out, which sets up the program for crashing at best and exploitation at worst.

WRONG: Allocation size and type do not agree

```
struct formula_one_car *formula_one_car_create(...)
{
    // ERROR: sizeof(struct formula_one_car) != sizeof(struct indycar_car)
    struct formula_one_car *c = malloc(sizeof(struct indycar_car));

    ...
    return c;
}
```

To avoid this, the size expression passed to `malloc` should *always* use `sizeof(*object)`. This approach prevents errors from copy-pasting lines, as well as updating automatically if the type is changed. Be careful to remember the `*`, otherwise just a pointer's worth of bytes are allocated!

Preferred: Allocation size based on destination

```
struct formula_one_car *formula_one_car_create(...)
{
    struct formula_one_car *c = malloc(sizeof(*c));

    ...
    return c;
}
```

A common antipattern seen by C++ programmers writing C code is known as "casting from `malloc`". This should not be done in C code. Recall that `void *` can be implicitly converted to any pointer type without a cast. The cast adds visual noise to code that is unnecessary. Casts, especially pointer casts, should only be performed when a compiler warning needs to be overridden.

Unpreferred: Do not cast from `malloc`

```
struct formula_one_car *formula_one_car_create(...)
{
    struct formula_one_car *c = (struct formula_one_car *)malloc(sizeof(*c));

    ...
    return c;
}
```

Initialization

Memory that is allocated with `malloc` is *not* initialized. Not "initialized to zero", but instead just filled with whatever used to be in that section of the Heap. This means that after being allocated, that memory almost certainly needs to be initialized.

WRONG: Using uninitialized values

```
int main(void)
{
    // Use preferred technique to allocate memory
    struct formula_one_car *c = malloc(sizeof(*c));

    if (!c) {
        perror("failed to create car");
        return 1;
    }

    // Attempting to access memory that has not yet been initialized.
    printf("car number: %d", c->racing_number);

    free(c);
}
```

Preferred: Setting initial values first

```
int main(void)
{
    // Use preferred technique to allocate memory
    struct formula_one_car *c = malloc(sizeof(*c));

    if (!c)
    {
        perror("failed to allocate memory");
        return 1;
    }

    // Accessing memory that has been initialized
    c->racing_number = 4;
    printf("car number: %d", c->racing_number);

    free(c);
}
```

The memory does not (and should not) be initialized if it is about to be overwritten, such as a buffer. It is inefficient to write a value only to immediately overwrite it, and can hide some forms of errors from tools like Valgrind.

Not Preferred: Initialization for memory about to be overwritten

```
int main(void)
{
    char *content = malloc(BUF_SIZE);
    if (!content) {
        perror("Unable to create content file using malloc");
        return 1;
    }

    // Anything done before the fgets() will be nullified by the fgets()
    // The following 3 lines are useless
    content[0] = 'a';
    content[1] = 'a';
    content[2] = '\0';

    // This will nullify all preview initialization, rendering it extraneous.
    fgets(content, BUF_SIZE, stdin);
    ...

    free(content);
}
```

Allocating Arrays

Because of the way that arrays decay to pointers, allocating space for arrays is the same as allocating heap space. The size of the array in elements is multiplied by the size of a single element from the array. From that point, either array syntax or pointer arithmetic will help manipulate that memory.

```c
int main(void)
{
    // Use unsigned integer for array size
    size_t memory_size = 10000;

    // Preferred: use size of dereferenced pointer
    double *addressable_memory = malloc(memory_size * sizeof(*addressable_memory));
    if (!addressable_memory) {
        perror("Unable to allocate array");
        return 1;
    }

    // iterate through array, initializing values.
    for (size_t i = 0; i < memory_size; ++i) {
        addressable_memory[i] = 1.0;
        printf("set location %zu to 1.0", i);
    }

    ...

    free(addressable_memory);
}
```

As with all arrays, the number of elements in the array should accompany the array wherever it goes. C does not have bounded arrays. Overrunning the bounds of an array in the Heap means overwriting other Heap objects, which can cause errors in completely different parts of the program.

Object Lifetimes

One of the benefits of Heap memory is that its lifetime is independent of function calls. Unlike the Stack, which is constantly being rolled forward and back, the Heap only expands on instructions from `malloc` and only reclaims on `free`. Since these are user-controlled functions, the developer using them can control how long an object lives during the program, its *lifetime*.

```
struct formula_one_car *formula_one_car_create(...)
{
    struct formula_one_car *c = malloc(sizeof(*c));

    if (!c) {
        return NULL;
    }

    ...

    return c;
}

void formula_one_car_destroy(struct formula_one_car *c)
{
    // free memory allocated at the pointer c
    free(c);
}
```

With the above, a developer can create any number of `struct formula_one_car` objects, and `free` them when done. The developer has total control over when these functions are called.

```
int main(void)
{
    struct formula_one_car *carlos = malloc(sizeof(*carlos));
    ...
    formula_one_car_destroy(carlos);

    ...
    struct formula_one_car *lando = malloc(sizeof(*lando));
    struct formula_one_car *daniel = malloc(sizeof(*daniel));
    ...

    formula_one_car_destroy(daniel);
    ...
    formula_one_car_destroy(lando);
}
```

Valgrind

Valgrind is a dynamic analysis tool that is useful to developers. It is very easy to fail to `free` a `malloc`ed chunk of memory. It is also distressingly easy to perform incorrect pointer arithmetic, or overrun the bounds of a `malloc`ed segment of memory. Valgrind can help detect and pinpoint these errors.

Valgrind is an excellent tool for detecting problems with the Heap. It is especially useful for detecting leaks and buffer overruns. Valgrind takes one required argument, the name of the program to run.

`hello.c`

```c
#include <stdio.h>

int main(void)
{
    puts("Hello World!");
}
```

```
$ valgrind ./hello
==23630== Memcheck, a memory error detector
==23630== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==23630== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==23630== Command: ./hello
==23630==
Hello World!
==23630==
==23630== HEAP SUMMARY:
==23630==     in use at exit: 0 bytes in 0 blocks
==23630==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==23630==
==23630== All heap blocks were freed -- no leaks are possible
==23630==
==23630== For lists of detected and suppressed errors, rerun with: -s
==23630== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The output from `valgrind` is always prefixed with the process ID, e.g. `==23630==`. Any output from the program under test will be displayed normally, but may end up being interleaved with `valgrind` output. The last part of Valgrind's output is a Heap summary: it shows how much memory has been used by the Heap. With the simple program, the library function `puts` makes one allocation, which is freed. This means that there are no leaks.

`hello-LEAKS.c`: *WRONG: Leaks memory*

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

int main(void)
{
    puts("Hello World!");

    int *array = malloc(10 * sizeof(*array));
    array[0] = 3;

    puts("Goodbye World!");
}
```

Valgrind output of leaked-memory program

```
$ valgrind ./hello-LEAKS
Hello World!
Goodbye World!
==25093==
==25093== HEAP SUMMARY:
==25093==     in use at exit: 40 bytes in 1 blocks
==25093==   total heap usage: 2 allocs, 1 frees, 1,104 bytes allocated
==25093==
==25093== LEAK SUMMARY:
==25093==    definitely lost: 40 bytes in 1 blocks
==25093==    indirectly lost: 0 bytes in 0 blocks
==25093==      possibly lost: 0 bytes in 0 blocks
==25093==    still reachable: 0 bytes in 0 blocks
==25093==         suppressed: 0 bytes in 0 blocks
==25093== Rerun with --leak-check=full to see details of leaked memory
==25093==
==25093== For lists of detected and suppressed errors, rerun with: -s
==25093== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind will helpfully tell the user that it should be rerun with `--leak-check=full` to see more details about the leaked memory. This will provide information about which function allocated the memory that was leaked.

Valgrind output of where leaked memory was allocated.

```
$ valgrind  --leak-check=full  ./hello-LEAKS
Hello World!
Goodbye World!
==25240==
==25240== HEAP SUMMARY:
==25240==     in use at exit: 40 bytes in 1 blocks
==25240==   total heap usage: 2 allocs, 1 frees, 1,104 bytes allocated
==25240==
==25240== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25240==    at 0x483A7F3: malloc (in /usr/lib/...)
==25240==    by 0x10918A: main (in f1)
==25240==
==25240== LEAK SUMMARY:
==25240==    definitely lost: 40 bytes in 1 blocks
==25240==    indirectly lost: 0 bytes in 0 blocks
==25240==      possibly lost: 0 bytes in 0 blocks
==25240==    still reachable: 0 bytes in 0 blocks
==25240==         suppressed: 0 bytes in 0 blocks
==25240==
==25240== For lists of detected and suppressed errors, rerun with: -s
==25240== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The output shows that the leak was from a call to `malloc` that was made in `main`. The full call stack is listed at the time the allocation happened. If the program is built in debugging mode, then even line numbers can be extracted.

Valgrind output of where leaked memory was allocated with a debug target

```
$ valgrind  --leak-check=full  ./hello-LEAKS
Hello World!
Goodbye World!
==25240==
==25240== HEAP SUMMARY:
==25240==     in use at exit: 40 bytes in 1 blocks
==25240==   total heap usage: 2 allocs, 1 frees, 1,104 bytes allocated
==25240==
==25240== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25240==    at 0x483A7F3: malloc (in /usr/lib/...)
==25240==    by 0x10918A: main (f1.c:9)
==25240==
==25240== LEAK SUMMARY:
==25240==    definitely lost: 80 bytes in 1 blocks
==25240==    indirectly lost: 0 bytes in 0 blocks
==25240==      possibly lost: 0 bytes in 0 blocks
==25240==    still reachable: 0 bytes in 0 blocks
==25240==         suppressed: 0 bytes in 0 blocks
==25240==
==25240== For lists of detected and suppressed errors, rerun with: -s
==25240== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Memory Overrun Detection

Even further, Valgrind will detect when parts of the Heap that were ***not*** allocated are accessed. This can be seen with the incorrect boundary condition on the `for`-loop:

`hello-LEAKS-OVERRUNS.c`: *WRONG: Leaks memory AND overruns array*

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

int main(void)
{
    puts("Hello World!");

    int *array = malloc(10 * sizeof(*array));

    for (int n=0; n <= 10; ++n) {
        array[n] = n;
    }

    puts("Goodbye World!");
}
```

```
$ valgrind  --leak-check=full  ./hello-LEAKS-OVERRUNS
Hello World!
==27379== Invalid write of size 4
==27379==    at 0x10919F: main (f1.c:10)
==27379==  Address 0x4a544d0 is 0 bytes after a block of size 40 alloc'd
==27379==    at 0x483A7F3: malloc (in /usr/lib/...)
==27379==    by 0x10918A: main (f1.c:9)
==27379==
Goodbye World!
==27379==
==27379== HEAP SUMMARY:
==27379==     in use at exit: 40 bytes in 1 blocks
==27379==   total heap usage: 2 allocs, 1 frees, 1,104 bytes allocated
==27379==
==27379== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==27379==    at 0x483A7F3: malloc (in /usr/lib/...)
==27379==    by 0x10918A: main (f1.c:9)
==27379==
==27379== LEAK SUMMARY:
==27379==    definitely lost: 40 bytes in 1 blocks
==27379==    indirectly lost: 0 bytes in 0 blocks
==27379==      possibly lost: 0 bytes in 0 blocks
==27379==    still reachable: 0 bytes in 0 blocks
==27379==         suppressed: 0 bytes in 0 blocks
==27379==
==27379== For lists of detected and suppressed errors, rerun with: -s
==27379== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

The “invalid write of size 4” indicates that unallocated memory in the Heap was written to, a violation of memory safety. Valgrind shows the line where the write happened. A usually-helpful message of any close-by allocations in the Heap is also printed, along with the stacktrace of when it was allocated. This is usually helpful as buffer overruns are the most common kind of Heap error.

Fixing errors and warnings from Valgrind should be done rigorously, in the same way that fixing compiler errors and warnings is done. Begin at the first chronological warning or error, and correct it. Do not try to address later errors without addressing the first ones first, as they are likely to cause cascading errors through the program run.

Uninitialized Value Detection

One important warning that Valgrind may emit is “uninitialised values” (sic).

hello-UNINITIALIZED.c

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

int main(void)
{
    puts("Hello World!");

    int *array = malloc(10 * sizeof(*array));

    for (int n=0; n < 5; ++n) {
        array[n] = 1;
    }

    printf("Good start: %d\n", array[2]);

    printf("Uh-oh: %d\n", array[7]);

    puts("Goodbye World!");
    free(array);
}
```

This kind of warning is emitted when the unitialized memory in the heap is used by the program, either in a conditional or an expression. This is definitely an error; those values may happen to be zero, but are more likely to be filled with garbage data.

```
$ valgrind --leak-check=full ./hello-UNINITIALIZED
Hello World!
Good start: 1
==28810== Conditional jump or move depends on uninitialised value(s)
==28810==    at 0x1091C3: main (hello-UNINITIALIZED.c:16)
==28810==
Uh-oh: 17
Goodbye World!
==28810==
==28810== HEAP SUMMARY:
==28810==     in use at exit: 0 bytes in 0 blocks
==28810==   total heap usage: 2 allocs, 2 frees, 1,104 bytes allocated
==28810==
==28810== All heap blocks were freed -- no leaks are possible
==28810==
==28810== Use --track-origins=yes to see where uninitialised values come from
==28810== For lists of detected and suppressed errors, rerun with: -s
==28810== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Exercises

Exercise 1: `fill_times.c`

Write a function `int *fill_times(size_t sz, int fill)` that will return a new array of length `sz` filled with the value `fill`. e.g., `fill_times(5, -3)` will yield `[-3, -3, -3, -3, -3]`.

Exercise 2

How large a space can be allocated before `malloc` returns `NULL`? Try to find this limit by writing a program to allocate and free until it fails.

Exercise 3: `reverser.c`

Write a function `char **read_lines(const char *filename, size_t *count)` that reads `count` lines from the file `filename` and stores each of them in a character array. The number of lines successfully read is stored in `count` when it returns. Assume that each line is no more than 1000 characters long.

Write a program that takes these inputs from the command line, with error-checking. It should then invoke the function to get the first `count` lines from the file. Then print all of these lines in reverse order to the console. Be sure the program reports no errors under `valgrind`.

Chapter 5. Intermediate Dynamic Memory

memset and calloc

```
#include <string.h>


void *memset(void *b, int c, size_t len)
```

It is often the case that an allocated chunk of memory should be set to some initial value. The `memset` function is useful for this kind of operation. Given a memory address and a byte value, it will fill up that memory with the specified value.

Use `memset` to fill memory with 01000100 bit pattern

```
int main(void)
{
    // size of char is 1, so no need to multiply with sizeof
    uint8_t *bytes = malloc(MAX_ARR_LEN);
    if (!bytes) {
        perror("Cannot allocate byte array");
        return 1;
    }

    // use memset to fill memory with 01000100 (0x44) bit pattern
    memset(bytes, 0x44, MAX_ARR_LEN);

    ...
    free(bytes);
}
```

However, if the desired value is 0 (i.e., 0, `'\0'`, or `NULL`), the better approach is to use the `calloc` function. `calloc` takes two arguments, the number of members, and the size of each one. Like `malloc`, it returns the address where the block of resevered Heap memory lies. Also like `malloc`, any such allocated memory must be `free`d by the end of the program.

```
#include <stdlib.h>


void *calloc(size_t count, size_t size);
```

`calloc-demo.c`: *Use* `calloc` *to zero memory with* `NULL` *pointers*

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

enum { COST_LEN = 10 };

int main(void)
{
    double *costs = calloc(COST_LEN, sizeof(*costs));
    if (!costs) {
        perror("Unable to allocate costs");
        return 1;
    }

    // show all values in calloc'd array
    for (int i = 0; i < COST_LEN; i++) {
        // will always print 0.0 for each value in array, all set to 0.0
        printf("%lf\n", costs[i]);
    }

    free(costs);
}
```

The `calloc` call both allocates and clears the memory. This can actually hide certain classes of errors, making debugging more difficult. Recall that the Valgrind tool can detect the use of uninitialized memory. Since `calloc` initializes all its memory to 0, there will be no use of uninitialized memory for Valgrind to report! Only use `calloc` when the use case is fulfilled:

1. A block of memory is needed; AND
2. It needs to be all zeroes.

A Truly Dynamic Array

Primitive arrays in C neither grow nor shrink. But, Heap-allocated memory can. Consider an array of `double`s.

```
    size_t array_max = 7;
    double *list = malloc(array_max * sizeof(*list));
```

It is important to track which elements of this array are valid, and which are not. Note that this is different from the total capacity available: the array might only be used for three values, but has space for 7.

```
    int counter = 0;
    list[0] = 6;
    ++counter;
    list[1] = 1;
    ++counter;
    list[2] = 2;
    ++counter;

    ...
```

But, each of these variables (`list`, `array_max`, and `counter`) are very tightly coupled. Changing one variable probably involves a change to the others. When dealing with variables with tight coupling, making them a `struct` is usually a good idea.

```
struct array {
    double *list;
    size_t array_max;
    size_t counter;
};
```

Now, rather than copying around three different variables, only the `struct`, or a pointer to the `struct`, need be tracked.

Unpreferred: many separate, tightly coupled variables

```
bool add_to_list(double *list, size_t counter, size_t array_max, double to_add);
```

Preferred: tightly coupled variables are grouped in a `struct`

```
bool add_to_list(struct list *l, double to_add);
```

This dynamic array now has both a size (how many valid elements it has) and a capacity (how many elements it can hold). The next step is to resize it at runtime to allow for more capacity.

realloc

Eventually, more memory may be needed for an array. If there is space in the Heap past the end of the current block, then that space can just be marked as being used as well. This means the value of the pointer (the address in the Heap) *will not* change.

But, if the Heap is very fragmented, there may not be enough space to expand into at the current location. In this case, the memory allocator will find a different area in the Heap that gives an unbroken, contiguous chunk of memory. It will then copy the data from the old location to the new location. This means the value of the pointer (the address in the Heap) *does* change, as it is in a new location.

```
#include <stdlib.h>


void *realloc(void *ptr, size_t size);
```

This operation is known as **reallocation**, and a function exists to perform it, `realloc`. `realloc` takes the pointer to be reallocated, and the newly-desired size. It will return the (possibly new) pointer to where the data live, or `NULL` if it is unable to do so.

`realloc-demo.c`: *Preferred: Use of* `realloc`

```c
#include <stdbool.h>
#include <stdlib.h>

struct ilist {
    int *data;
    size_t curr_size;
    size_t total_capacity;
};

bool ilist_append(struct ilist *l, int value)
{
    if (!l) {
        return false;
    }

    l->data[l->curr_size++] = value;

    if (l->curr_size == l->total_capacity) {
        int *tmp = realloc(l->data, (l->total_capacity + 1) * sizeof(*l->data));

        if (!tmp) {
            l->curr_size--;
            return false;
        }

        l->total_capacity++;
        l->data = tmp;
    }

    return true;
}
```

Despite it not being done here, it is ***generally*** a good idea to grow memory by doubling the current amount for a given allocation. Consider two arrays that start with an initial capacity of 8. One will just grow by 1 element each time (as shown above), while the other will double each time it needs more capacity.

Table 1. Starting capacity of 8

Strategy	Number of Elements Added	Number of realloc calls
+1	10	2
	100	92
	1000	992
×2	10	1
	100	4
	1000	7

The number of `realloc` calls decreases dramatically by simply asking for twice as much each time: 8, 16, 32, 64, 128... This can result in unused memory, but will still be on the same order of magnitude as what is truly needed.

Arrays of pointers

Arrays of data are common, arrays of pointers even more so. Consider how to build an array that would store some unknown number of variable-length strings. "Variable-length strings" definitely implies that they are pointers; most likely allocated space on the Heap. "Unknown number" implies that their container must *also* live on the Heap.

This requires first creating the container. Because the container requires tightly-coupled variables, a `struct` is a good choice.

Building a container that will be an array of pointers

```
struct starting_grid {
    char **driver_names;
    size_t size;
    size_t capacity;
};

struct starting_grid *starting_grid_create(void)
{
    struct starting_grid *sg = malloc(sizeof(*sg));
    if (!sg) {
        perror("malloc failed");
        return NULL;
    }

    // Start grid size at 32, chosen arbitrarily
    sg->capacity = 32;
    sg->size = 0;

    // allocate a char pointer for each driver name
    // sizeof(*sg->driver_names) == sizeof(*char)
    sg->driver_names = malloc(sg->capacity * sizeof(*sg->driver_names));
    if (!sg->driver_names) {
        perror("malloc failed");
        // perform appropriate teardown
        ...
    }

    return sg;
}
```

Pay special attention to what happens if the array cannot be created: the containing `struct` must have its Heap storage `free`d. This will prevent a leak if the code follows that path. At any place where control flow might leave the function early, it needs to make sure that the `struct` is either completely constructed or

not at all.

When it comes to storing pointers in a container, the decision of whether to store copies or originals needs to be made. This is a case-by-case decision. For this example, copies will be stored, which means a form of duplicating strings is needed. But, the current set of functions takes a bit of work to duplicate a string.

Unpreferred: Library function `strdup` is simpler

```
{
    // +1 for the terminating NUL byte
    size_t str_size = 1 + strlen(s);

    char *new = malloc(str_size);
    if (!new) {
        perror("malloc failed");
        return;
    }

    // Not preferred in comparison to strdup
    strncpy(new, s, str_size);
}
```

strdup

The `strdup` function is a useful utility for making a copy of a string, **str**ing **dup**lication.

```
#include <string.h>


char *strdup(const char *s);
```

This function performs a `malloc()` for the returned string. It may return `NULL` if it is out of memory. Since this function performs such a `malloc()`, any duplicated string must later be `free`d.

When it comes to storing a variable array of strings, `strdup` makes the code much simpler. The `realloc` call (and the need to carefully handle its return value) has already been covered.

For the example below, suppose that the lines struct contains the array of char pointers, `char** strings`

Set all values in lines to duplicates of string s

```
bool lines_set_all(struct lines *l, const char *s)
{
    if (!l || !s) {
        return false;
    }

    // set all string values in the lines struct's strings array to s
    for (size_t i = 0; i < MAX_LINES; ++i) {
        l->strings[i] = strdup(s);
        if (!l->strings[i]) {
            perror("Cannot copy string");
            return false;
        }
    }

    return true;
}
```

Extracting objects from this container should be straightforward. Because array syntax is a shortcut for pointer arithmetic, it can be used on such an array of pointers.

Printing each element of an array of pointers

```
void lines_print(const struct lines *l)
{
    if (!l) {
        return;
    }

    for (size_t i=0; i < MAX_LINES; ++i) {
        printf("%s <- The line at %d\n", l->strings[i], i);
    }
}
```

Finally, the container deallocation must be sure to `free` *all* allocations. There are the individual strings, the array of pointers to those strings, and the container itself. It makes sense to capture all this work in a function.

Deallocation of a container

```
void lines_destroy(struct lines *l)
{
    if (!l) {
        return;
    }

    for (size_t i=0; i < MAX_LINES; ++i) {
        free(l->strings[i]);
    }

    free(l->strings);
    free(l);
}
```

Like all deallocations, the pointer in question is no longer valid after being `free`d.

WRONG: Cannot use a Heap pointer after `free`

```
int main(void)
{
    ...
    free(l);

    // the functionality above can also be replicated by a call to lines_destroy()

    //ERROR: l's lifetime is over after it has been free'd.
    wordlist_print(l);
}
```

Exercises

Exercise 1: `observations.c`

Build out a generic `struct observations` that stores a growable array of `double` values. Include functions:

- `struct observations *observations_create(size_t starting_size)`
- `void observations_destroy(struct observations *o)`
- `bool observations_add(struct observations *o, double value)`
- `double observations_get(struct observations *o, size_t idx)`

Exercise 2: `filebuffer.c`

Build out a generic `struct filebuffer` that stores the contents of a text file. Include functions:

- `struct filebuffer *filebuffer_create(FILE *fp)`
- `void filebuffer_destroy(struct filebuffer *fb)`
- `char *filebuffer_getline(struct filebuffer *fb, size_t line)`
- `void filebuffer_print(struct filebuffer *fb)`

Chapter 6. C Program Design

With an understanding of pointers and unit tests, more attention can now be given to designing programs in C. C lacks object-orientation as part of its syntax, unlike C++ or Python. This does not mean C lacks objects, just that they are not given privileged syntax. C's objects are `struct`s, and the methods on them are functions.

typedef

The `typedef` keyword defines a new type. This is a far cry from defining new `class`es in Python!

It takes a type declaration and the name of the new type.

Define a new type 'fuel_kgs'

```
typedef unsigned int fuel_kgs;
```

In the above example, there is now a new type called `fuel_kgs`. Note that unsigned integers were used, as negative fuel values are invalid. New variables can now be created with this type.

```
fuel_kgs merc_fuel_capacity = 32;
```

The compiler treats the new variable as the original type declaration; in this case, `unsigned int`. The compiler *does not* create a brand-new type that can only interact with other values of that type. This means that there is no "type safety" or "type domains" in C stemming from the use of `typedef`s.

Unpreferred: Combination of the same underlying type pass without warning in C

```
typedef  unsigned int  fuel_kgs;
typedef  unsigned int  miles;

{
    miles track_length = 5;
    liters fuel_kgs = 32;

    // Misleading: combining units with no warnings to compiler
    // this addition won't mean anything
    unsigned int misleading = track_length * fuel_kgs;
}
```

Still, this concept can be used to better communicate what type of unit a given piece of data is. The above example may be misleading, but the developer may recognize that the types `fuel_kgs` and `miles` should not be added together!

Generally, `typedef`s should not be used. They tend to obfuscate more than they make clear. This is especially so when the `typedef` contains pointers.

Unpreferred: Pointers in a `typedef`

```
typedef int **matrix;
```

Here, an array of integer arrays is defined as a `matrix`. (Unpreferred)

There is one major use for `typedef`, and that is opaque types.

Opaque Types

Because of C's explicit nature, it does not easily allow for information hiding. A `struct` object can have its fields manipulated manually, causing problems.

WRONG: Tightly-coupled fields should not be meddled with indiscriminately

```
{
    struct paragraph *p = paragraph_create();

    // BAD, changing values in size of paragraph manually without changing other fields
    // could cause problems to the rest of it.
    paragraph->lines[0] = "Less Than Ten Lines";
    paragraph->num_lines = 10;
    ...
}
```

Note that a `typedef` requires a type *declaration*, not a type *definition*.

This allows for a special form of hiding data type implementations in C. These are known as "opaque types".

Consider a paragraph object. These fields are tightly coupled and generally should *only* be manipulated by code that understands their relationship.

```
struct paragraph_ {
    char *title;
    char **lines;
    size_t line_nums;
};
```

To prevent users from modifying the members directly, a `typedef` is created in the header file.

```
typedef struct paragraph_ paragraph;
```

The declaration of the `struct` is used. Because the definition of the `struct` will be hidden, an underscore is appended to the `struct` name. This has no programmatic effect; it is a style choice. Do **not** prepend an underscore, as would be done in Python. Prepended underscores in C are reserved for use by the C standard, not for user code.

Once the `typedef` is in place, the rest of the header exposes functions that will manipulate the `struct`. But, all functions declarations will only use pointers to the new `typedef`.

paragraph.h

```
// define only if not defined
#ifndef WORDLIST_H
#define WORDLIST_H

#include <stdbool.h>

typedef struct paragraph_ paragraph;

// create and destroy functions
paragraph *paragraph_create(void);
void paragraph_destroy(paragraph *p);

// other functionalities
int paragraph_addline(paragraph *p, char* line, int linenum);
int paragraph_changesize(paragraph *p, int lines);
void paragraph_print(const paragraph *p);


#endif
```

Because pointer values/addresses are a consistent size, the compiler knows how to pass around and store pointers to `struct`s whose size it does **not** know. This allows use of the opaque type without knowing its implementation details. Consider the type `FILE *` from the standard library. The size or members of a `FILE` are never used by a developer; all interaction is through a `FILE *`.

```
int main(void)
{
    paragraph *p = paragraph_create();
    if (!p) {
        perror("Could not create paragraph");
        return 1;
    }

    // Unpreferred; the return value should be checked
    paragraph_changesize(p, 2);
    paragraph_addline(p, "You're Supposed to be my Friend", 0);
    paragraph_addline(p, "The Cookies", 1);

    paragraph_print(p);

    paragraph_destroy(p);
}
```

Only the implementation code for this header file knows and manipulates the `struct`'s internals.

paragraph.c

```c
#include "paragraph.h"

#include <stdlib.h>
#include <string.h>

struct paragraph_ {
    char *title;
    size_t line_nums;
    char **lines;
};

paragraph *paragraph_create(void)
{
    // allocate memory for paragraph struct and sets values appropriately
    ...
}

void paragraph_destroy(paragraph *p) {
    // free all associated allocated memory
    ...
}

// other functionalities
int paragraph_addline(paragraph *p, char* line, int linenum) {
    ...
}

int paragraph_changesize(paragraph *p, int lines) {
    ...
}

void paragraph_print(const paragraph *p) {
    ...
}
```

The implementation file should always have the corresponding header as the very first `#include`, before any others. This makes it clear what the interface for the compilation unit is. It also would set any global variables that need to be set to use the library (macro definitions, etc.).

Application Programming Interfaces (APIs)

Any large library of code can be reused. This is done by calling library functions, which hide their complexity from the programmer. This hiding is beneficial, as it allows a programmer to use a library without needing to know its implementation details. This set of functions, plus any public data structures, constitute an *Application Programming Interface*, or *API*. Designing good APIs can be challenging. It takes skill and practice to design an API that is useful to others.

In C, extra care is needed to produce a useful API. This can be aided by identifying which parts need to be publicly accessible.

C API Design steps

1. Identify any opaque types that are needed.
2. Identify any `struct`s that are needed.
3. Identify any constants that are needed.
4. Identify any additional functions that are needed.

These guidelines are presented from the perspective of this course. A style or architecture guide for a specific project may specify; otherwise, in which case, it takes precedence.

Always use appropriate header guards. Ensure that there are no blank lines before or after the guarded section. This makes any `#include` of the header minimal.

Unpreferred: Blank lines before header guard 'engages'

```
#ifndef WILLIAMS_H

#define WILLIAMS_H

...

#endif
```

Opaque Types

Opaque types should only be made when some element of their implementation needs to be hidden from outside modification.

Use the method outlined earlier to define an opaque type and its compilation unit.

Any methods should always take the pointer to the relevant object as the first parameter. Think of this as the explicit `self` in a Python class method. That is, in fact, exactly what `self` is.

Methods should begin with a consistent name prefix; this means `wordlist_append` instead of `append_wordlist`. A consistent prefix to all functions makes it easy to tell at a glance which ones are related to the object, and prevents naming conflicts across compilation units.

`struct`s

Any `struct` defined for use in a module should be organized with care. Check alignment requirements of its members to avoid padding when possible.

A `struct` in an API means that the user of the API may need to work with any of its fields. It further means that its member fields may be manipulated. If that is unacceptable to the library, perhaps converting the `struct` to an opaque type may be in order.

Constants

As with any other integer constants, prefer defining them in an `enum` over `const` or `#define`. This ensures their type-safety, visibility, and constancy. Group different kinds of constants in their own `enum`s. It is **strongly** recommended to give the API-exposed constants a consistent prefix, to avoid naming conflicts across other headers.

```
// Possible return values from functions
enum { PARAGRAPH_SUCCESS, PARAGRAPH_MEM_OVERFLOW, PARAGRAPH_PRINT_FAILURE };

// Possible constant values to use
enum { PARAGRAPH_MAX_LINES=64 };
```

Functions

Functions should be flexible. They should work well with the basic types of the C or POSIX libraries.

For instance, instead of a `cars_print` function, that prints to `stdout`, consider instead a `cars_fprint` function that takes a `FILE *` destination to print to. This is not to say that this change should always be done! Always value code that solves the problem at hand over the hypothetical future. But, the more generic 'destination' makes it easier for others to use the module.

Interface vs. Implementation

Keep the interface of a module, its header file, as small as possible. This will mean that the `#include` of the header is a fast operation.

Audit the `#include` headers in the interface to ensure that they are needed in that header (defining types, e.g.).

Every `struct` specified should need to be used externally. But do question: can this complexity be captured in a method?

Confirm that constants specified in the header are needed by users of the module. If not, they should be repaired to the `.c` file.

Find any functions in the public interface that should be made private. Most modules have a number of such 'helper' functions that should be moved to the implementation file only, and made `static`.

Documentation

The public interface of a module, its header file, is the place for public documentation. Each function should have some explanation to the user of its purpose, parameters, and pitfalls. There do exist custom comment formats that can be automatically extracted into browsable documentation, such as Doxygen.

This *definitely* is not to say that comments can be left out of the implementation file. The difference is the audience of the comments. In the header file, the audience is anyone who uses the compilation unit. They should not need to worry about implementation details of the module. In the implementation file, the audience is the authors and maintainers of the code.

`static` Linkage

Any variable or function that is not part of the public interface should only exist in the implementation file, and be marked `static`. Marking a name as static means that it has "`static` linkage", or "internal linkage". The linker will only use that function locally, in the current compilation unit.

This prevents name conflicts at link time.

`paragraph.c`: *Non-public items should be marked* `static`

```
// only visible inside the compilation unit
static size_t total_paragraphs_initialized;

// Function unavailable for call outside of file
static bool check_valid_size(paragraph *p)
{
    return (p->line_nums <= PARAGRAPH_MAX_LINES);
}
```

Exercises

Exercise 1

Implement a currency type. There should be methods to create different kinds (USD vs EUR, e.g.) that may be converted between each other. Support at least 5 different kinds. It should be possible to add two currency objects together. There should also be a function for displaying the object, as well as one for stringification.

Chapter 7. Unit Testing

In C, unit testing is generally more difficult than in other languages. In languages like Python, ***reflection*** allows the program to refer to source code details like the names of variables. This concept is not present in C, which means that any unit tests must be explicitly labeled and called as such. This means additional bookkeeping for the developer.

Even worse, because of C's low-level access to memory, a test may overwrite critical parts of the Most Important Picture, ruining the test framework.

This lack of straightforward unit testing means that a number of different C libraries have been made to run unit tests on C code. This chapter uses 'Check'.

Check creates a separate binary program (therefore having a separate `main` function). It will almost certainly link in all other parts of the system under test, to exercise them.

make check

In the interests of automation, `make(1)` should handle any unit test building and execution. While not as universal as `make clean`, the target of `make check` is commonly used for building and running tests.

When building and running unit tests, do **not** automatically rebuild the system under test. The user may very well be testing a different version of the target system, checking for regressions.

Unpreferred: Making the program a dependency of target `check` to force a rebuild

```
fullprog: fullprog.o dependency_one.o dependency_two.o

...

testfullprog: testfullprog.o dependency_one.o dependency_two.o

# fullprog is always rebuilt when mytest is run. Unpreferred!
check: fullprog mytest
    ./mytest
```

It is typical to put all relevant test code in a subdirectory. Automated tests built with `make check` should also be run, which usually requires a `make(1)` target like the following:

Preferred: Typical `check` target

```
# test is the test directory

# Extra Library for Check
test/test_all: LDLIBS += -lcheck

# test_all can be considered to be comprehensive
check: test/test_all
    ./$^

# Comprehensive runner testing all dependencies
# Assume testing of  dependency one and two, introduced in the previous example
test/test_all: test/test_all.o test/test_dependency_one.o test/test_dependency_two.o
# Modules under test
test/test_all: dependency_one.o dependency_two.o
```

Check Organization

With Check, there are five key pieces to writing tests. Other automated test systems will have similar equivalents.

1. Suite Runner
2. Test Suites
3. Test Cases
4. Unit Tests
5. Assertions

A Suite Runner runs Test Suites, collecting their results to report. A given system will usually just need one Suite Runner.

A Test Suite is a high-level grouping of Test Cases. There is usually one Test Suite per compilation unit. Test Suites get added to the Suite Runner.

A Test Case is a low-level grouping of Unit Tests. The goal is usually to test some specific aspect on the system under test. Test Cases get added to a Test Suite.

Unit Tests are a series of Assertions that the system under test is working correctly. They should ideally be fairly small in scope. Unit Tests get added to a Test Case.

Assertions are simple true-or-false statements. If the assertion is true, it is successful. If the assertion fails, that indicates a problem with the system under test (or the test as written).

Suite Runner

The `main` function in the `check` target is the site of the Suite Runner.

test_all.c

```
extern Suite *test_dependency_one_suite(void);
extern Suite *test_dependency_two_suite(void);

int main(void)
{
    int tests_failed;
    SRunner *sr = srunner_create(NULL);

    // Here, we invoke the test suites for dependencies one and two
    srunner_add_suite(sr, test_dependency_one_suite());
    srunner_add_suite(sr, test_dependency_two_suite());
    // You are free to add as many more suites as needed

    ...

    srunner_run_all(sr, CK_NORMAL);

    // report the test failed status
    tests_failed = srunner_ntests_failed(sr);

    srunner_free(sr);

    // return 1 or 0 based on whether or not tests failed
    if (tests_failed == 0) {
        return 0;
    }
    else {
        return 1;
    }

}
```

In the case of a single Suite, `srunner_create()` may take that Suite as an argument. Any number of Suites may be added to the Suite Runner. Most of the time, all tests should be run. It is possible to run only a subset of tests with `srunner_run` and `srunner_run_tagged`; see Check documentation for more details.

The output of the Suite Runner can be configured to no output, error-only output, or all tests. This is controlled by the second argument to `srunner_run_all` et. al. Most of the time, the `CK_NORMAL` level is appropriate.

`CK_SILENT`	No output
`CK_MINIMAL`	Summary output only

`CK_NORMAL`	Failed tests and summary
`CK_VERBOSE`	All tests
`CK_ENV`	Use level from environment variable `CK_VERBOSITY`

The Suite Runner has pointers to all its Suites, which have pointers to all their Test Cases. The entire structure is freed by the call to `srunner_free()`.

The Suites themselves are generally in separate compilation units. Hence, the `extern` providing a hint to the developer that the function is defined elsewhere. The C compiler would see an `extern` storage class on a function as being redundant.

It is important that a test program return an appropriate value from `main` as to whether the tests succeeded or not. This is found by checking the number of tests that failed with `srunner_ntests_failed`: 0 is success and should return that from `main`, any other number is failure.

Note the unusual construct `!!failed`. The `!!` looks like a null set of operations, but it has an important side effect. This coerces the value of `failed` to be either `0` or `1`. If the last line were `return failed`, it is possible for the value of `failed` to be truncated. Even though `main` has a return type of `int`, many systems truncate the value to just one unsigned byte. This means that if 256 tests had failed, the program could report the testing as having succeeded!

Test Suites

Test Suites roughly correspond to compilation units. A given Test Suite should collate and gather all the Test Cases for that compilation unit.

`test_dependency_one.c`

```
Suite *test_dependency_one_suite(void)
{
    Suite *suite_one = suite_create("DEPENDENCY_ONE");
    TFun *curr = NULL;

    TCase *case_core = tcase_create("core");
    ...
    suite_add_tcase(s, case_core);

    // More TCases can be created as needed

    ...

    return suite_one;
}
```

The main export of a compilation unit's worth of tests needs to be its Test Suite. This can be accomplished fairly easily with a function that builds and returns the Test Suite. It is unlikely that anything else need be exported from the compilation unit. This means all other functions and variables will be marked as `static` for the linker to ignore.

Every Test Suite has a name that it is created with via `tcase_create`. The name is useful for filtering tests to run or results to parse.

Each Test Case then needs to be created and added to the Test Suite with `suite_add_tcase`.

Test Cases

Test Cases are testing a functional group. Like Test Suites, each Test Case has a name that it is created with, using the `tcase_create` function. The name is useful for filtering tests to run or results to parse.

A Test Case is composed of multiple Unit Tests, individual functions of type `TFun`. Each Unit Test is added to the Test Case with `tcase_add_test`. This is most easily done with a number of `static`, `NULL`-terminated arrays (`static` to 'hide' from the linker).

```
static TFun test_list[] = {
    test_dependency_one_create,
    test_dependency_one_odd,
    NULL
};

Suite *test_dependency_one_suite(void)
{
    Suite *suite_one = suite_create("DEPENDENCY_ONE");
    TFun *curr = NULL;
    curr = test_list;
    while (*curr) {
        // add the test from the core_tests array to the tcase
        tcase_add_test(tc_core, *curr);
        curr++;
    }
    // add the tcase to the suite
    suite_add_tcase(suite_one, tc_core);

    ...

    return suite_one;
}
```

Test Fixtures

A given test case may have 'fixtures', setup and teardown functions to run for each contained Unit Test. These can be 'checked', meaning that they are run before and after every Unit Test. They may also be 'unchecked', in which case the setup is run at the start of the Test Case, and the teardown is run at its end. These fixtures need to be registered when a Unit Test is added to a Test Case.

`test_dep_one.c`

```
// pointer for dependency one
static dep_one *dep_one_ptr;

// setup and teardown functions for the struct
static void first_setup(void) { dep_one_ptr = one_create("Monaco"); }
static void first_teardown(void) { one_destroy(dep_one_ptr); }

// create a suite and return it
Suite *test_dep_one_suite(void)
{
    Suite *suite_one = suite_create("DEPENDENCY_ONE");
    TFun *curr = NULL;

    TCase *tc_first = tcase_create("first");

    // provide setup and teardown functions
    tcase_add_checked_fixture(tc_first, in_setup, in_teardown);
    curr = input_tests;

    while (*curr) {
        tcase_add_test(tc_first, *curr);
        curr++;
    }
    suite_add_tcase(suite_one, tc_input);

    return suite_one;
}
```

Unit Test

A Unit Test is the actual set of steps to take to ascertain correct functionality. The Unit Test is set up and ended with a set of Check macros rather than just defined functions. `START_TEST()` takes an argument that will be the name of the function. The function will be `static`, so there will be no naming conflicts at link-time.

```
START_TEST(test_name_constructor)
{
    // check string equality with driver->name and the string literal, "Checo"
    ck_assert_str_eq(driver->name, "Checo");
}
END_TEST
```

Since it is a function body, it can contain arbitrary C code. Every Unit Test will have a number of Check Assertions inside its function body.

Test Loops

Building out successful Unit Tests may involve running the given Unit Test against a variety of data. Rather than building individual Unit Tests for each piece of data, it may make more sense to build an array of test data (input and expected values), and run a given Unit Test through that array in a loop.

```
static int cube_data[][2] = {
    {0, 0},
    {1, 1},
    {2, 8},
    {-2, -8},
    {3, 81}
};
enum { CUBE_DATA_SZ = sizeof(cube_data)/sizeof(cube_data[0]) };

START_TEST(test_cube)
{
    // assert that each sub-array cubes the 0th element for the 1st element
    ck_assert_int_eq(cube(cube_data[_i][0]), cube_data[_i][1]);
}
END_TEST

int main(void)
{
    // add test to loop through test_cube
    tcase_add_loop_test(tc_core, test_cube, 0, CUBE_DATA_SZ);
}
```

The `tcase_add_loop_test` adds a Unit Test to a Test Case, but will run it in a loop. The loop variable is `_i`, and will range from `begin` ≤ `_i` < `END`. These sort of table-driven tests can be very easy to maintain. In the example shown above, the data are simply arrays of numbers. For more complex tests, it can be useful to define `struct` that tracks the data members.

```
struct furniture_measurement {
    int length;
    int width;
    int height;

    // expected value and tolerance, as doubles
    double ex_value_dollars;
    double tol_value;
};

// structs can be listed in bracket form for the purpose of this test
struct complex_function_test furniture_inventory[] {
    { 3, 4, 5, 500.0, 0.25 },
    { 5, 9, 10, 750.0, 0.50 },
    { 5, 15, 10, 1750.0, 0.70 },
    ...
};

START_TEST(complex_test)
{
    struct complex_function_test *table = &furniture_inventory[_i];

    /*
     * Ensure that the cost that we produced for the table during the test is within
     * the range 'tol_value' of the expected dollar value of the table indicated in
     * the struct.
     */

    ck_assert_double_eq_tol(furniture_measurement(table->length, table->width, table-
>height), table->ex_value_dollars, table->tol_value);
}
END_TEST
```

Assertions

A Unit Test is made up of a number of Assertions. These are simple macros to compare values. Here is a sample of common Assertions.

`ck_assert_int_eq`

Confirms that two signed values are equal

`ck_assert_uint_eq`

Confirms that two unsigned values are equal

`ck_assert_double_eq_tol`

Confirms that two doubles are within tolerance `tol` of each other

`ck_assert_str_eq`

Confirms that two strings have the same contents

`ck_assert_ptr_eq`

Confirms that two pointers point to the same object

`ck_assert_mem_eq`

Confirms that two areas of memory are the same

`ck_assert_null`

Confirms that a pointer is `NULL`

`ck_assert_abort`

Fails unconditionally; useful in complex conditional logic tests

Many more Assertions exist in a wide variety (`_eq`, `_ne`, `_lt`, etc.). Consult the Check documentation for a full accounting.

assert()

There does exist a C function called `assert`. It will test a condition, passed in as an argument. If the condition is false, the program is halted with an error message. If the condition is true, the program continues normally.

However, this function macro has some drawbacks that must be understood. When building a program for release (rather than debugging), all `assert`ions are stripped out. This means that if the `assert` call had a side effect, that side effect no longer happens.

Unpreferred: Should avoid use of `assert`

```
void destroy_f1_car(car *c)
{
    //DANGER: This line may be omitted by the preprocessor
    assert(c != NULL);

    // The following decrements/increments will not happen if DEBUG is disabled.
    assert(c->fuel_in_kg-- != 0);
    assert(c->driver_experience++ != 0);
}
```

When the symbol `NDEBUG` ("not debugging") is set, all `assert` lines are stripped out. A program **must** be robust even in this case. That means that error-checking, non-`NULL`ness, and the like should happen even if there were no `assert` calls present. At that point, using `assert` at all seems questionable.

Exercises

Exercise 1

Fill out the `rect_permieter` function such that it passes the unit tests described below. Ensure that you have written checks to pass the given tests.

The function below will return the perimeter of a rectangle with the given dimensions. If erroneous input is detected, it will return `-1`. Ensure you are writing the function to meet these requirements. Refer to the unit tests detailed below for more guidance.

```
int rect_perimeter(int len, int width) {
    ...
}

// INDIVIDUAL TESTS FOLLOW:

START_TEST(test_rect_reg)
{
    ck_assert_int_eq(rect_perimeter(4,5), 18);
}
END_TEST

...

START_TEST(test_rect_negative)
{
    ck_assert_int_eq(rect_perimeter(-9,5), -1);
}
END_TEST

...

START_TEST(test_rect_zero)
{
    ck_assert_int_eq(rect_perimeter(0,0), -1);
}
END_TEST
```

Exercise 2

Understand the following directives:

Error Handling Directives

- Any calculation where the inputs are negative numbers OR the output is a negative number will return the double `-1.0`
- Any calculation where the END result is `0` will return the double `0.0`

Given that, write unit tests using `ck_assert` functions as described above, for each of the functions below. Ensure that you are covering every case described for each function.

```
double area(double len, double width){
    return (len * width);
}

double volume(double len, double width, double height){
    return (len * width * height);
}

// assume sqrt() is a function that reliably returns square roots, and if its input is
negative, it will return NaN
double diagonal_of_cube(double side) {
    return sqrt((side * side) + (side * side))
}

double surface_area_of_cube(double side) {
    return 6 * area(side);
}
```

After writing this comprehensive set of unit tests, do all of the functions above meet the standards outlined in the set of Error Handling Directives? Why, or why not?

Chapter 8. Bit manipulation

This chapter explores what C has to offer when dealing with ***bit data*** is required.

Bitwise operators

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a byte. C language is very efficient in manipulating bits. C contains *bitwise operators* specifically designed to operate on the individual bits of a number.

Table 2. Bitwise operators:

Operator	Description
&	Bitwise AND (Binary operator - both operands are 1)
|	Bitwise OR (Binary operator - at least one operand is 1)
^	Bitwise Exclusive OR (XOR) (Binary operator - ONLY one operand is 1)
~	Bitwise complement (Unary operator - convert 1 to 0; convert 0 to 1)
<<	Bitwise shift LEFT (Binary operator - shift left operand LEFT bits specified by right argument)
>>	Bitwise shift RIGHT (Binary operator - shift left operand RIGHT bits specified by right argument)

bitwise.c

```c
#include <stdio.h>

int main(void)
{
    unsigned int a = 44; /* 44 = 0010 1100 */
    unsigned int b = 77; /* 77 = 0100 1101 */

    /* For reference: Results in order below. */
    /* AND -> 12 = 0000 1100 */
    /* OR -> 109 = 0110 1101 */
    /* XOR -> 97 = 0110 0001 */
    /* COMPLEMENT -> 45 = 0010 1101 */
    /* LSL -> 176 = 1011 0000 */
    /* LSR -> 11 = 0000 1011 */

    printf("BIT AND: %d & %d is %d\n", a, b, a & b );
    printf("BIT OR : %d | %d is %d\n", a, b, a | b );
    printf("BIT EXCLUSIVE OR: %d ^ %d is %d\n", a, b, a ^ b);
    printf("COMPLEMENT: ~%d is %d\n", a, ~a );
    printf("LEFT SHIFT 2: %d << 2 is %d\n", a, a << 2 );
    printf("RIGHT SHIFT 2: %d >> 2 is %d\n", a,a >> 2 );
}
```

which produces this output:

```
BIT AND: 44 & 77 is 12
BIT OR : 44 | 77 is 109
BIT EXCLUSIVE OR: 44 ^ 77 is 97
COMPLEMENT: ~44 is -45
LEFT SHIFT 2: 44 << 2 is 176
RIGHT SHIFT 2: 44 >> 2 is 11
```

It is instructive to use the bit representations of the variables `a` and `b` and compare the results of the various bitwise operators.

- Bitwise operators in compound assignment statements

Bitwise operators may be used in *compound assignment statements*.

compoundbit.c

```c
#include <stdio.h>

int main(void) {

    unsigned int a = 44; /* 44 = 0010 1100 */
    unsigned int b = 77; /* 77 = 0100 1101 */

    /* 12 = 0000 1100 */
    printf("COMPOUND BIT AND: %d & %d is %d\n", a, b, a &= b );
    printf("a= %d\n", a);

    /* 176 = 1011 0000 */
    printf("COMPOUND LSL 2 BITS: %d =<<2 is ", a );
    printf("%d\n", a <<= 2);
    printf("a= %d\n", a);
    /* a is now 48 = 0011 0000 */

    /* Using compound assignments to print will not yield changes. */
    printf("\nCOMPOUND LSL 2 BITS: %d =<<2 is %d\n", b, b <<= 2 );
}
```

which prints:

```
COMPOUND BIT AND: 44 & 77 is 12
a= 12
COMPOUND LSL 2 BITS: 44 =<<2 is 176
a= 176

COMPOUND LSL 2 BITS: 77 =<<2 is 77
```

The other operators behave in a similar fashion and use the familiar compound assignment expression.

NOTE

Avoid using compound assignment statements in `printf` statements. Note how the last line of output prints the *new* value of b in the output `COMPOUND LEFT SHIFT 2 BITS: 77 =<<2 is 77`. The above applies to *any compound operator* - not just the bit operators.

- Adding `1` to a number's complement negates the number

Worthy of note is that *adding one to the complement of a number negates the number*.

`twos_comp.c`

```c
#include <stdio.h>

int main(void)
{
    int a = 61;
    int b = -14;

    printf("NEGATED ~%d is %d\n", a, ~a + 1);
    printf("NEGATED ~%d is %d\n", b, ~b + 1);
}
```

which prints:

```
NEGATED ~61 is -61
NEGATED ~-14 is 14
```

The above program prints the *twos complement* of the numbers `61` and `-14`.

Bitfields

C allows for ***bitfields***, which are fields that may occupy less than a byte of storage. Using bitfields instead of `char` or `int` types to represent values is storage-efficient.

Bitfields are ideal when the value of a field or group of fields will never exceed a limit or is withing a small range.

Bitfields may be specified as `struct` or `union` members.

The general form of bitfield usage is:

```
struct {
    type [member_name] : width ;
};
```

`type` is an integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.

`member_name` is the name of the bit-field.

`width` is the number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

For example:

```
struct date_with_bit_fields {
    // day has value between 1 and 31, 5 bits
    unsigned int d : 5;

    // month has value between 1 and 12, 4 bits
    unsigned int m : 4;
};
```

The program `bitfield1.c` shows how bitfields can hold the same data yet be memory efficient:

bitfield.c

```c
#include <stdio.h>

struct date {
    unsigned int d;
    unsigned int m;
};

struct date_with_bit_fields {
    // d has value between 1 and 31, so 5 bits
    unsigned int day : 5;
    // m has value between 1 and 12, so 4 bits
    unsigned int month : 4;
};

int main(void)
{
    struct date dt = { 9, 12 };
    printf("Size with int fields is %lu bytes\n", sizeof(dt));
    printf("Date: %d/%d\n\n",
            dt.m, dt.d);

    struct date_with_bit_fields dtbf = { 9, 12 };
    printf("Size with bit fields %lu bytes\n", sizeof(dtbf));
    printf("Date: %d/%d\n",
           dtbf.month, dtbf.day);
}
```

which prints:

```
Size with int fields is 8 bytes
Date: 12/9

Size of date struct with bit fields 4 bytes
Date: 12/9
```

- Bitfields must be large enough to hold assigned values

The program must ensure that the bit fields are declared large enough to hold the value.

Consider the following assignment statement:

```
struct date_with_bit_fields invalid = { 40 , 12 };
```

`40` is too large to be accurately represented with the bit field coded:

```
unsigned int day : 5;
```

`cc` complains:

```
bitfield.c: In function ‘main’:
bitfield.c:16:39: warning: large integer implicitly truncated to unsigned type [-
Woverflow]
  struct date_with_bit_fields dtbf = { 40, 12 };
```

but the program executes:

```
Size of date struct with bit fields 4 bytes
Date created from is date struct with bit fields 4/12
```

`40` in binary is `10100`. The bitfield `day` takes the first 5 bits, yielding `01000`, which is the number `8` as shown in the output.

Another unexpected result is using **signed integers** with bitfields.

The program `bitfields-signed.c`, not shown here, has the above struct `date_with_bit_fields`, coded with `int` members and used in the above program. When executed, `bitfields-signed` produces the following output:

```
Size of date struct with int fields 12 bytes
Date created from is date struct with int fields 12/31/2019

Size of date struct with bit fields 8 bytes
Date created from is date struct with bit fields -4/-1/2019
```

Note, the output from the date struct with bit fields where the members are `int` as opposed to `unsigned int` are *negative*. The value `31` was stored in five bit *signed integer* which is equal to `11111`. The sign bit is a 1 (all the bits are 1), so `11111` is a *negative number*. Internally, the **two's complement** of the binary number is computed to get its actual value.

The two's complement of `11111` is `00001` which is equivalent to decimal number 1, and since it was a negative number the program assigns a -1. A similar thing happens to `12` in which case you get 4-bit representation as `1100 `which on calculating two's complement yields a value of -4.

- Forcing byte alignment of bit fields

A special, unnamed bit field of size `0` is used to force alignment on next boundary. The boundary (byte, two-byte, etc) depends on the type of the next field. For example, consider the following program:

`force-alignment.c`

```c
#include <stdio.h>

// Using bitfields - forcing alignment
struct force_align {
    unsigned int day : 5;
    unsigned int : 0;   // alignment will be forced on the boundary for next
    unsigned int month : 4; // forced onto boundary
};

struct date_with_bit_fields {
    unsigned int day : 5;
    unsigned int month : 4;
};

int main(void)
{
    struct force_align a = { 7, 5 };
    printf("Size with alignment %lu bytes\n", sizeof(a));

    struct date_with_bit_fields a = { 7, 5 };
    printf("Size with bit fields %lu bytes\n", sizeof(b));

}
```

which prints:

```
Size with alignment 8 bytes
Size with bit fields 4 bytes
```

Forcing alignment may be useful when writing struct data in binary format to a file that requires a specific format.

- Restrictions on bitfield use

No pointers to bitfields: Since bitfields are not guaranteed to be aligned on a byte, there cannot be a pointer containing the address of a bitfield. Pointers to structs containing bitfields is permissable.

Arrays of bit fields are not allowed: An array name is a pointer to its first element. Since pointers to bitfields are forbidden, arrays of bitfields are forbidden as well.

Endianness

Different platforms store basic data types differently. The terms used to describe the difference are *little endian* and *big endian*.

In **little endian** byte order, the *least significant byte (LSB) stored in the lowest address*.

Table 3. Little-endian storage of 0xFEEDFACE

CE	FA	ED	FE

In **big endian** byte order, the *LSB is stored in the highest address*.

Table 4. Big-endian storage of 0xFEEDFACE

FE	ED	FA	CE

The following program outputs the byte order of the host platform:

endian.c

```
#include <stdio.h>

int main(void)
{
    unsigned int i = 1;
    char *test = (char*)&i;
    int little_endianness = (*test) ? 1 : 0;

    if (!little_endianness) {
        printf("Big\n");
    } else {
        printf("Little\n");
    }
}
```

which prints:

```
Little
```

In the above program, a character pointer `c` is pointing to an integer `i`. Since size of character is 1 byte, when the character pointer is de-referenced it will contain only first byte of the integer `i`. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

When performing I/O on the same platform the byte order is not that important but **when sending data over a network (which assume big endian)** or **operating in a heterogenous platform environment** where different platforms may have different byte order the program may need to convert from big to little and vice-versa.

There are no builtin functions that convert big to little or vice-versa for **all data types**. The *ntohl, ntohs* functions convert long and short integers from network (big) to host; the *htonl, htons* functions convert network (big) to host long or short integers.

These functions are the subject of the next chapter.

- The `htons`, `ntohs`, `htonl` and `ntohl` functions

As previously mentioned, the programmer usually does not concern themselves with byte order *other than sending data over the network*.

C provides the following functions to convert from *host* (big or little endian) to *network* (big endian) order:

Table 5. Byte order conversion functions:

Function	Description
`htons(uint16_t)`	Convert `uint16_t` *short* argument from host to network order
`ntohs(uint16_t)`	Convert `uint16_t` *short* argument from network to host order
`htonl(uint32_t)`	Convert `uint32_t` *int or long* argument from host to network order
`ntohl(uint32_t)`	Convert `uint32_t` *short* argument from network to host order

These functions are available in the `arpa/inet.h`. The *man page* for these functions states that some systems use `netinet/in.h` instead of `arpa/inet.h`.

These functions obey the identities:

```
htonl(ntohl(a)) == a                    ntohl(htonl(a)) == a
htons(ntohs(a)) == a                    ntohs(htons(a)) == a
```

The *data types* `uint16_t` and `uint32_t` are *unsigned 16 and 32 bit integers*, respectively and are found in the include file `inttypes.h`.

The following program shows a little-endian byte order integer converted to a network (big endian) byte order integer, then back:

endian-functions.c

```
#include <stdio.h>
#include <inttypes.h>
#include <arpa/inet.h>

int main(void)
{
    uint16_t example_short = 0x1357;
    uint32_t example_long = 0x13572468;

    // print originals unchanged
    printf("Example short - 0x%x\n", example_short);
    printf("Example long - 0x%x\n", example_long);

    // host to network order examples
    printf("Network short - 0x%x\n", htons(example_short));
    printf("Network long - 0x%x\n", htonl(example_long));

    // network to host order examples
    printf("Host short - 0x%x\n", ntohs(example_short));
    printf("Host long - 0x%x\n", ntohl(example_long));

    // print originals after converting twice
    printf("Originals - 0x%x\n\n, 0x%x\n\n", htons(ntohs(example_short)), htonl(ntohl
(example_long)));
}
```

The output follows:

```
Example short - 0x1357
Example long - 0x13572468
Network short - 0x5713
Network long - 0x68245713
Host short - 0x5713
Host long - 0x68245713
Originals - 0x1357, 0x13572468
```

These functions *swap bytes*. The functions do not know what byte order their arguments are.

The lines below write the same number:

```
printf("-> %x\n", htons(short_num));
printf("-> %x\n", ntohs(short_num));


-> 5713
-> 5713
```

These functions are coded such that if executed on a little endian system, the functions ***swap bytes***. If executed on a big endian system, they are **no ops**.

Binary I/O

Use the functions `fread` and `fwrite` to operate on binary data. The files must be opened in *binary* mode.

- `fread` - Read a number of bytes from a file

```
size_t fread(void * buffer, size_t size, size_t count, FILE * stream)
```

`buffer`: Pointer to the buffer where data will be stored. A buffer is a region of memory used to temporarily store data.

`size`: The size of each element to read in bytes.

`count`: Number of elements to read.

`stream`: Pointer to the FILE object from where data is to be read.

The file position indicator for the stream is advanced by the number of characters read.

`fread` returns an integer equal to `count` when the call is successful and `EOF` *is not reached*. `fread` returns the number of characters read when successful and `EOF` *is reached*. If an error occurs, a value less than `count` is returned. Also, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

- `fwrite` - Write a number of bytes to a file

```
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);
```

`buffer`: Pointer to the buffer where data will be written from. A buffer is a region of memory used to temporarily store data.

`size`: The size of each element to write in bytes.

`count`: Number of elements to write.

`stream`: Pointer to the FILE object from where data is to be written.

The file position indicator for the stream is advanced by the number of characters written.

`fwrite` returns The number of objects written successfully, which may be less than count if an error occurs.

If size or count is zero, `fwrite` returns zero and performs no other action.

This program shows using `fwrite` to create a binary file and `fread` to read the created file and print its contents:

bin-io-array.c

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

enum { SIZE = 5 };

int main(void)
{
    int counter;
    double first[SIZE] = {1.,3.,5.,7.,9.};
    double second[SIZE];
    FILE *filep;
    int i;

    size_t num_read, num_written;

    filep = fopen("output.bin", "wb");
    if (filep == NULL) {
        // failed to open test.bin
        exit(EXIT_FAILURE);
    }

    // use fwrite to WRITE in binary.
    num_written = fwrite(first, sizeof(*first), SIZE, filep);
    fclose(filep);

    // if we did not read the right amount, exit with failure.
    if (num_written != SIZE) {
        exit(EXIT_FAILURE);
    }

    // open file to read from it
    filep = fopen("output.bin", "rb");

    if (filep == NULL) {
        // failed to open output.bin
        exit(EXIT_FAILURE);
    }

    // use fread to READ in binary.
```

```
    num_read = fread(second, sizeof(*second), SIZE, filep);
    fclose(filep);

    // if we did not read the right amount, exit with failure.
    if (num_read != SIZE) {
        exit(EXIT_FAILURE);
    }

    // print all values in array
    for (i = 0; i < 5; i++) {
        printf("value #%d: %f\n", i,  second[i]);
    }
}
```

The above program creates the file `example.bin` and outputs:

```
value #0: 1.000000
value #1: 3.000000
value #2: 5.000000
value #3: 7.000000
value #4: 9.000000
```

The statement:

```
fwrite(a, sizeof(*first), SIZE, filep);
```

dereferences the pointer `a` and applies the `sizeof` operator to derive the number of bytes to write. A similar expression is used to derive the number of bytes to read in the call to `fread`. However, the value returned by `fread` and `fwrite` is obtained from the `SIZE` parameter.

The program includes checks for successful file open and checking for the correct number of elements written and read. The program may be refactored into *functions* that validate the file open and read/write operations.

- Reading and writing structures in binary mode

Given a struct `a_struct`, write an instance of `a_struct` named `struct_instance`, using `fwrite` to the file pointer `fp` as follows:

```
size_t num_elems_written = fwrite(&struct_instance, sizeof(struct_instance), 1, fp);
```

Reading the instance using `fread` is as follows:

```
size_t num_elems_read = fread(&struct_instance, sizeof(struct_instance), 1, fp);
```

For both functions, the return value should be `1` (the `count` argument).

The program `bin_io_struct.c` creates five instances of a struct, prints the instance data to the console, and writes the instances in binary mode one at a time to an output file. The program opens the output file, reads the data back and prints the instance data to the console.

`bin-io-struct.c`

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

enum { MAX_NAME_SIZE = 20, NUM_DRIVERS = 3 };

// Will be used to initialize driver structs below
char *driver_names[MAX_NAME_SIZE] = {"Aki Norris", "Hrishik Vettel", "Rahul Sainz"};
int driver_counter = 0;

typedef struct driver {
    // driver name, arbitrarily sized string that has to be allocated for
    char *name;
    // racing number from 0 to 99
    unsigned int racing_number : 7;
    // 20 drivers total in f1, driver standing from 0 to 19 from their latest season
    unsigned int driver_standing : 5;
    // salary is a double
    double salary;
} driver;

// Random number generator
unsigned int rand_bounded(int upper_b) { return rand() % (upper_b + 1); }

// initialize new driver- can only be used thrice here (names array)
void create_driver(driver * new_driver) {
    new_driver -> name = driver_names[driver_counter++];
    new_driver -> racing_number = rand_bounded(99);
    new_driver -> driver_standing = rand_bounded(20);
    new_driver -> salary = 150000 * rand_bounded(10);
}

void print_driver(driver * f1_driver) {
    printf("Formula 1 Driver %s, racing as #%u\tLatest Standing: P%u\tSalary: %lf\n",
```

```
            f1_driver->name, f1_driver->racing_number, f1_driver->driver_standing,
f1_driver->salary);
}

int main(void) {

    FILE *filep;
    int local_driver_count;
    int i = 0;

    // initialize file pointer in write mode
    filep = fopen("drivers.bin", "wb");
    if (filep == NULL) {
        // failed to open drivers.bin
        perror("failed to open bin file.\n");
        exit(EXIT_FAILURE);
    }

    // create racing driver structs and write them to file
    for (i = 0; i < NUM_DRIVERS; i++) {
        size_t num_written;
        driver f1_driver;
        create_driver(&f1_driver);
        printf("Writing driver with following details...\n");
        print_driver(&f1_driver);

        // write our driver struct to a file using fwrite
        num_written = fwrite(&f1_driver, sizeof(f1_driver), 1, filep);
        if (num_written != 1) {
            perror("failed to write properly to file.");
            fclose(filep);
            exit(EXIT_FAILURE);
        }

        printf("Write successful.\n");
    }

    // close file pointer
    fclose(filep);

    // initialize file pointer in read mode
    filep = fopen("drivers.bin", "r");
    if (filep == NULL) {
        // failed to open drivers.bin
        perror("failed to open bin file.\n");
        exit(EXIT_FAILURE);
    }
```

```
    printf("---\nReading from file:\n---\n");

    // read racing driver structs from file and print them
    for (i = 0; i < NUM_DRIVERS; i++) {
        size_t num_read;
        driver f1_driver;

        // read data from file into f1_driver struct
        num_read = fread(&f1_driver, sizeof(f1_driver), 1, filep);
        if (num_read != 1) {
            perror("failed to read properly from file.");
            fclose(filep);
            exit(EXIT_FAILURE);
        }

        print_driver(&f1_driver);
    }
}
```

The output is:

```
Writing driver with following details...
Formula 1 Driver Aki Norris, racing as #7   Latest Standing: P7 Salary: 0.000000
Write successful.
Writing driver with following details...
Formula 1 Driver Hrishik Vettel, racing as #58  Latest Standing: P13    Salary:
450000.000000
Write successful.
Writing driver with following details...
Formula 1 Driver Rahul Sainz, racing as #44 Latest Standing: P8 Salary: 1350000.000000
Write successful.
---

.Reading from file
---
Formula 1 Driver Aki Norris, racing as #7   Latest Standing: P7 Salary: 0.000000
Formula 1 Driver Hrishik Vettel, racing as #58  Latest Standing: P13    Salary:
450000.000000
Formula 1 Driver Rahul Sainz, racing as #44 Latest Standing: P8 Salary: 1350000.000000
```

The data for the most part loaded into the struct instances is randomly generated.

This program uses a struct with bitfields but use of bitfields is not required.

The program verifies that reads and writes were performed as needed after each read/write. This is the preferred practice, and can be accomplished with helper functions as needed. Helper functions may prove far more useful in larger files and projects, and are recommended.

Exercises

Exercise 1 – `divisible.c`

Write a function `divisible` to fulfill the following requirements. It should accept an `unsigned int` as an argument. Without using `/` or `%`, determine whether or not the unsigned integer is divisible by the numbers 2, 4, and 16. Return how many of the three (2, 4, or 16) that the input is divisible by.

Examples

- `divisble(32)` will return `3`
- `divisible(12)` will return `2`
- `divisible(1)` will return `0`
- `divisible(6)` will return `1`

Chapter 9. Profiling and Optimization

Profiling code involves running it in such a way as to measure its performance. The ***kind*** of performance needs to be specified:

- Execution speed (faster is better)
- Memory use (lower is better)
- Disk use (lower is better)
- Function calls (fewer is better)
- Data processed (more is better)

Armed with knowledge of how a program is currently performing, it can then be optimized along these metrics.

Gross Measurements

The first tool for measurement is the user of the program. While not reliable or consistent, in the end programs have users, and that user's experience is relevant. A program that runs in 10 seconds with a progress bar will *seem* faster than one that runs in 11 seconds with no user feedback.

Measuring with C Library Timing Functions

The simplest way to time a section of code is to check the current time, run the relevant code, then check the time again. This does mean that the overhead of "checking the current time" is part of the measurement. However, as long as the other measurements *also* have that overhead, there should be no disparity.

Make sure that only the relevant part is being measured. It is very easy to accidentally put in an extra calculation in the part being measured. In practical terms, it means that every measurement will look like the following:

```
int main(void)
{
    struct timespec start, end;

    // Critical section, the part being measured
    timespec_get(&start, TIME_UTC);
    function_to_measure();
    timespec_get(&end, TIME_UTC);

    time_t seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    if (nanoseconds < 0) {
        seconds -= 1;
        nanoseconds += 1000000000;
    }

    printf("%lu.%09ld\n", seconds, nanoseconds);
}
```

`struct timespec` is a data structure designed to represent an interval. It is made up of a `time_t`, capable of measuring the number of seconds since Jan 1, 1970; and a `long` that tracks the number of nanoseconds for the current second.

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};
```

Measuring with a Profile Build

One of the problems with measuring performance via the C Library is that the areas to measure must be surrounded with code. Further, if there are user-input or disk-reading activities, they may dramatically affect any measurement taken.

It is possible to have **every** single function be time-tracked. This is mechanically added during compiling and linking when building a *profiling build* of a program or library.

If all functions are measured, then user-input parts are easier to filter out for the true performance of a program (since that time can be measured accurately and then subtracted from the total time).

Making such a build requires custom compiler and linker flags, both to build in the profiling symbols and to know how to call functions with the profiling metrics. For GCC, the `-pg` flag ("profile generate") is passed to both parts of the pipeline. Note that this is a linker *flag*, not a linker *library*.

```
profile: $(TARGET)
profile: CFLAGS += -pg
profile: LDFLAGS += -pg
```

Running the program with profiling information **will** be slower than normal. But, it still provides highly actionable feedback.

`gprof`

When a program with profiling symbols is run, it will generate a file `gmon.out` in the current working directory of the program. This is a call graph profile. It tracks when functions are called, and how much time is taken inside the function. Be warned that running the program a second time will replace the existing `gmon.out` with the new execution's call graph profile. If the program crashes or is terminated by a signal, it may fail to write out the complete `gmon.out`. This file is consumed by the profile grapher, `gprof`.

`gprof` takes in the binary to check and (optionally) the call graph profile file to show. Multiple profiles may be passed in; the resultant statistics will be across all runs. It will then generate some extremely useful measurement data.

```
$ gprof program gmon.out
```

By default, `gprof` will print out textual prose descriptions of the output. For purposes of this module, this helper text will be omitted from any output. This is equivalent to running `gprof` with the `-b`, 'brief' command-line option.

Flat Profile

The *flat profile* shows the total amount of time spent in each function.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.01        1     0.00     0.00  path_find
  0.00      0.00     0.00    59821     0.00     0.00  point_create
  0.00      0.00     0.00      776     0.00     0.00  node_set
  0.00      0.00     0.00        1     0.00     0.00  bound_check
  0.00      0.00     0.00        1     0.00     0.00  cleanup
  0.00      0.00     0.00        1     0.00     0.00  finput
  0.00      0.00     0.00        1     0.00     0.00  maze_print
```

The output is sorted by `self seconds`, followed by `calls`. Each column may help in identifying bottlenecks. The two `self` columns, `self seconds` and `self ms/call` only count time in the function, not in any functions it calls.

This helps provide a view of where time is spent in the program. In the sample above, the vast majority of the time is spent in `path_find`. Examining that function may provide insights into optimization. At the very least, it probably needs to be split into more functions from an architectural perspective.

Similarly, the fact that `point_create` is called sixty thousand times is possibly cause for concern. Compare the number of calls against how much data is processed: is it growing quickly or slowly?

Call Graph Analysis

The *call graph analysis* shows the time spent in each function and its children.

```
                     Call graph

granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       8/59821       bound_check [3]
                0.00    0.00   59813/59821       path_find [7]
[1]      0.0    0.00    0.00   59821         point_create [1]
-----------------------------------------------
                0.00    0.00     776/776         finput [5]
[2]      0.0    0.00    0.00     776         node_set [2]
-----------------------------------------------
                0.00    0.00       1/1           path_find [7]
[3]      0.0    0.00    0.00       1         bound_check [3]
                0.00    0.00       8/59821       point_create [1]
-----------------------------------------------
                0.00    0.00       1/1           main [13]
[4]      0.0    0.00    0.00       1         cleanup [4]
-----------------------------------------------
                0.00    0.00       1/1           main [13]
[5]      0.0    0.00    0.00       1         finput [5]
                0.00    0.00     776/776         node_set [2]
-----------------------------------------------
                0.00    0.00       1/1           main [13]
[6]      0.0    0.00    0.00       1         maze_print [6]
-----------------------------------------------
                0.00    0.00       1/1           main [13]
[7]      0.0    0.00    0.00       1         path_find [7]
                0.00    0.00   59813/59821       point_create [1]
                0.00    0.00       1/1           bound_check [3]
-----------------------------------------------
```

The output is separated into a number of different entries. Each entry has one primary line that is numbered; this is the function being analyzed for that entry. Entries are ordered by time spent in the primary function. Lines prior to the primary line are callers to that analyzed function. Lines after it are calls made by the analyzed function.

Looking at entry `[3]`, the `bound_check` function is being analyzed. It is called once, from the `path_find` function. It makes a total of 8 calls to `point_create`, out of the sixty thousand calls to `point_create`.

This call graph analysis shows that the main bottleneck seems to be in the `path_find` function, specifically how many calls it makes to `point_create`.

callgrind

The Valgrind tool has a number of plugin-based programs to do dynamic analysis. One of those tools is `callgrind`, which will generate a call graph similar to `gprof`. While slightly slower, `callgrind` tends to be more accurate in its measurements. Another benefit of `callgrind` over `gprof` is that it will work regardless of how the binary was built. As a downside, a program run though `callgrind` will likely run much slower.

```
$ valgrind --tool=callgrind ./program
...
==24835==
==24835== Events    : Ir
==24835== Collected : 41408739
==24835==
==24835== I   refs:      41,408,739
```

The initial output will state a value for `Ir`, that is, "instructions read". This gives a sense of how many instructions were executed during the program run. The run also will dump events to a file, `callgrind.out.PID`. The true analysis comes when running the paired `callgrind_annotate` program.

```
$ callgrind_annotate --auto=yes --include=.
```

The `auto=yes` option will print any accompanying source code, and the `include=` option tells `callgrind_annotate` where to look for source files. Note that a program would need to be built with debugging symbols to take advantage of these options. Regardless of such debugging symbols, the `Ir` number for each function will be displayed.

```
--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
41,408,739  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir          file:function
--------------------------------------------------------------------------------
18,421,283  maze.c:path_find [/home/student/maze/maze]
 8,436,744  /build/glibc-5mDdLG/glibc-2.30/malloc/malloc.c:_int_malloc
 4,849,110  /build/glibc-5mDdLG/glibc-2.30/malloc/malloc.c:_int_free
 4,668,498  /build/glibc-5mDdLG/glibc-2.30/malloc/malloc.c:calloc
 1,615,167  point.c:point_create [/home/student/maze/maze]
 1,493,383  /build/glibc-5mDdLG/glibc-2.30/malloc/malloc.c:malloc_cons...
 1,257,059  /build/glibc-5mDdLG/glibc-2.30/malloc/malloc.c:free
   119,716  ???:0x00000000001090e0 [???]
   119,714  ???:0x0000000000109130 [???]
    66,742  /build/glibc-5mDdLG/glibc-2.30/elf/dl-addr.c:_dl_addr
...
```

This shows how many instructions were run in each function. Note that both library functions and application functions are tracked. This can make it easier to narrow down where a performance problem might be, even if it is in a library call. However, this will not clearly show slowness in I/O, as it is only measuring CPU instructions.

```
--------------------------------------------------------------------------------
-- Auto-annotated source: point.c
--------------------------------------------------------------------------------
Ir

      .  point *point_create(int y, int x, int w)
418,747  {
239,284          point *tmp = calloc(1, sizeof(*tmp));
13,216,064  => ???:0x0000000000109130 (59,821x)
179,463          tmp->x = x;
179,463          tmp->y = y;
179,463          tmp->weight = w;
119,642          tmp->visited = false;
119,642          tmp->next = NULL;
 59,821          return tmp;
119,642  }
```

The annotated source displays the number of instructions executed throughout the program for each

line of source. These are x86 Assembly instructions, of which there may be multiple such instructions for a single line of C source code. Notice that all the numbers are multiples of the smallest, 59,821. That means some assignments took 3 instructions each, some take 2 instructions each. The preamble, or entry to the function, takes 7 instructions. This analysis can help the developer find opportunities for improvement.

Not every instruction takes the same amount of time on the x86-64 architecture. However, this is still a good starting point to find the lines that are executed the most. Reducing the number of times they are taken is a good path to start optimization.

`callgrind` can also be used to check L1 and L2 cache reads and writes. Consult the documentation for more information on this functionality. It takes a variety of tools to profile code effectively.

Optimization

With the knowledge gained from profiling code, an eye can be turned towards optimization. This topic is fraught with traps; it is surprisingly easy to spend hours optimizing code a mere 1%. There are **three major rules** when it comes to optimization.

1. Don't Optimize.
2. (For experts only!) Don't Optimize Yet.
3. Measure before optimizing.

Taking these in reverse, the first step is to measure what a program is doing. This is the job of profiling. Identify the actual bottleneck, not just the imagined one. It is amazingly easy to spend time optimizing a part of the program that does not matter significantly.

Look at the total time spent in a function. Whichever function spends the most time is the most ripe for a performance improvement. Further, how can calls to that function be limited? Can results be cached? Can some number of calls be eliminated? The best optimization is *running less code*. After all, if running code takes time and effort, *not* running the code saves that time and effort.

When it comes to optimizing, there is a hierarchy of how effective (in cost/time) certain categories of optimization happen to be.

1. Buy faster hardware
2. Organize program data efficiently
3. Use a better algorithm
4. Engage in micro-optimizations

Buying faster hardware is almost always the fastest, cheapest, and easiest way to gain performance. If a program gains a 10% speedup on better hardware, that often will be cheaper than the developer time it takes to speed up a program by the same amount. And, if the program is retired, the hardware is still ready to be used elsewhere.

Organizing data efficiently is often more important than adjusting the algorithms used. Very few programs are limited based on CPU usage, more are limited in how quickly they can read data, either from memory, disk, or the network. Generally, flat arrays of data can be processed faster than complex records or collections. This will be explored further in *Data Structures & Algorithms*.

Better algorithms can make a difference, sometimes significantly. For instance, suppose that a large array had to be searched for 3–10 distinct values. It might very well be faster to *sort* the array first, and then use `bsearch()` to find those values. This will be explored further in *Data Structures & Algorithms*. Once again, though, the greatest performance gains tend to lie with reducing the number of times an algorithm

must run.

Micro-optimizations are where highly clever-looking code constructs can be used to shave off a few instructions. These are almost never to improve performance greatly. But, if a line of code is run one million times, reducing that instruction count by even a small amount can result in significant improvement.

Common Micro-Optimizations

More optimization and understanding will be covered in *x86 Assembly Programming*. The techniques described here are partially based on that future understanding.

Every one of these optimizations is, to some extent, sacrificing readability and clarity for speed. Consider every code sample of micro-optimizations as marked with 'Unpreferred'.

Loop Unrolling

Loops are expensive computationally because the act of jumping around in code (via conditionals) tends to be less performant that sequentially executing statements. Therefore, if a loop removes its conditional, or even reduces how many times the conditional is evaluated, performance can increase.

Loop Rolled Up

```
for (size_t n=0; n < 100; ++n) {
    execute_function(n);
}
```

In this first case, the conditional is evaluated 101 times. 100 times, the loop is executed, and the conditional is evaluated before each iteration. The 101st time is when the conditional is finally false, and the program continues on its way.

Loop Partially Unrolled

```
for (size_t n=0; n < 100; n += 4) {
    execute_function(n);
    // Loop partially unrolled for performance reasons
    execute_function(n + 1);
    execute_function(n + 2);
    execute_function(n + 3);
}
```

With a partially unrolled loop, the code is harder to read, but the conditional is now only evaluated 26 times. This means fewer jumps in the resulting program, and *may* lead to increased performance.

`switch` over multiple `if`

C sports the `switch`/`case` construct. This control flow can actually be far more computationally efficient than a chain of `if`/`else` blocks.

`if`/`else` *Chain*

```
if (s == 't') {
    ...
} else if (s == 'h') {
    ...
} else if (s == 'x') {
    ...
} else if (s == '1') {
    ...
} ...
```

Once again, this comes down to evaluating many conditionals and jumping around in the program. If the value of `s` was `'3'`, then every conditional before it must be tested. The average chain of length ***N*** would execute ***N***/2 tests before finding the correct condition.

`switch`/`case` *Construct*

```
switch (s) {
    case 't': ...
    case 'h': ...
    case 'x': ...
    case '1': ...

    ...
}
```

A `switch`/`case` construct, however, will only execute the conditional ***once***. When faced with many possible conditions (especially when spread over a complete range), the `switch`/`case` can be extremely performant.

Preincrement over Postincrement

One of the least important micro-optimizations is to prefer preincrement/predecrement. As covered in earlier lessons, the postincrement version returns the value, then increments the variable.

Postincrement

```
int x = 53;
// y is 53
int y = x++;
```

The postincrement operation is ***never faster*** than an equivalent preincrement operation, but may, in some edge cases, be slower. This is due to the compiler needs to potentially maintain a 'temporary' of the variable's value to be used in calculations, rather than just incrementing the variable's value and using that.

Preincrement

```
int x = 53;
// y is 54
int y = ++x;
```

The general rule would be to always prefer preincrement, and resort to postincrement only if needed by the expression.

Exercises

Exercise 1

For the following code, what can be done to improve performance?

```
/// Returns a pointer to the first instance of c in s, NULL otherwise
char *strchr(const char *s, char c)
{
    for (size_t n=0; n < strlen(s); n++) {
        if (s[n] == c) {
            return s + n;
        }
    }

    return NULL;
}
```

Exercise 2

For the following code, what can be done to improve performance?

```
/// Prints prime numbers up to limit
void primes_to(int limit)
{
    for (int n=2; n < limit; n++) {
        bool is_prime = true;

        for (int i=2; i < sqrt(n); i++) {
            if (n % i == 0) {
                is_prime = false;
                break;
            }
        }

        if (is_prime) {
            printf("%d\n", n);
        }
    }
}
```

Exercise 3

Given the following code:

```
int arr[100];

// assume that the values of arr are as follows:
// [0,1,2,3,4 ... 98,99]

...

// ANALYZE THE FOLLOWING BLOCK OF CODE

for (int choice=1; choice<5;choice++) {
    if (choice == 1) {
        linear_search(arr, 5);
    } else if (choice == 2) {
        for (size_t i = 0; i < 100, i++) {
            arr[i] += 3;
        }
    } else if (choice == 3) {
        check(arr[3]);
        check(arr[4]);
    } else if (choice == 4) {
        int t = 1;
        t++;
        arr[5] += t;
    } else {
        // do nothing
    }
}

...

int check(int input) {
    for (size_t j=0; j<3; j++) {
        input += check(input);
    }
    return 3+input;
}
```

You are expected to run the profiler on it and analyze the report. How many times is `check` being called? Write a brief, 50-word analysis on the results of the call graph and the flat profile of the above code.

Additionally, use the timing code described in the lesson to manually time the `for` loop.

Exercise 4

Perform as many optimizations as you can to the code in the previous exercise. Use the techniques provided in the lesson, and name the ones you are using.

Use `gprof` once again to produce the call graph and flat profile of your new and improved code. Highlight the similarities and differences between the code that you have produced and the original code, using the profiler results as evidence. Write at least 200 words.

Exercise 5

Revisit a previous project using the tools enumerated in this chapter. What kind of performance bottlenecks exist? How can they be improved? Compare the before and after profiles.

Chapter 10. Library Functions

C vs. POSIX libraries

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

The goal of POSIX is to ease the task of cross-platform software development by establishing a set of guidelines for operating system vendors to follow. Ideally, a developer should have to write a program only once to run on all POSIX-compliant systems.

The C POSIX library is a specification of a C standard library for POSIX systems. It was developed at the same time as the ANSI C standard. Some effort was made to make POSIX compatible with standard C.

POSIX includes additional functions to those introduced in standard C.

stdlib.h

The header `stdlib.h` defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting. Often seen as a collection point for all the miscellaneous functionality that seems to fit nowhere else, the stdlib contains many interesting features for a variety of applications.

This module has used several items from `stdlib.h`.

- The `div_t`, `ldiv_t` data types and `div`, `ldiv` functions.

The `div_t` and `ldiv_t` types represent `struct`s that represent the result value of an integral division performed by the functions `div` and `ldiv`, respectively.

The `div_t` type and `div` function operate on the `int` type; the `ldiv_t` type and `ldiv` function operate on the `long` type.

The C standard defines the `div_t` type as the following struct:

```
typedef struct {
    int quot, rem;
} div_t;
```

and the `div` function as:

```
div_t div(int numer, int denom);
```

The `ldiv_t` type and `ldiv` functions are defined with the `long` type instead of `int`.

The short program `div_function.c` shows its use:

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

int main(void)
{
    div_t q_and_r = div(111, 11);
    printf("%dR%d", q_and_r.quot, q_and_r.rem);
}
```

which prints:

```
10R1
```

- Converting strings to numbers

The `strtod`, `strtof`, and `strtold` functions convert the *initial portion* of the string argument to double, float, and long double representation, respectively.

The `strtol`/`strtoul` and `strtoll`/`strtoull` functions scan a string and convert the *initial portion* of the string argument to an signed/unsigned `int` or long (`strtol`) or a `long long`. The integer type conversion functions also accept a `base` between `0` and `36`

These functions replace the deprecated functions `atoi`, `atol` and `atoll` functions which **should never be used**.

- Converting strings to floats/doubles/long doubles

The definition of `strtod` is shown. The definitions for the functions `strtof` and `strtold` are similar except for the data type of the returned value (`float`, `long double`):

```
double strtod(const char * npt, char **ept);
```

Note that the second parameter `ept` is a *pointer to a pointer*.

The `strtod` function returns the initial portion of the string `npt` converted to a type double value. Conversion ends upon reaching the first character that is not part of the number. Initial whitespace is skipped. Zero is returned if no number is found.

if conversion is successful, the address of the first character after the number is assigned to the location pointed to by `ept`. If conversion fails, `npt` is assigned to the location pointed to by `ept`.

The program `strtod_ex.c` shows how to use `strtod`:

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

void use_strtod(char * parse)
{
    char *remaining;
    printf("extracted -> %lf\nremaining: '%s'\n\n", strtod(parse, &remaining),
remaining);
}

int main (void)
{
    char * input = "1.1 2.2 3.3 ... it continues";
    char * remaining;
    double found;

    // examples with number at beginning and end
    use_strtod("3.14 is Pi");
    use_strtod("Pi is 3.14");

    printf("\n");

    // 0 means not found
    while( (found = strtod(input, &remaining)) != 0)
    {
        printf("extracted -> %lf\n", found);
        // Reassign input and continue
        input = remaining;
    }
}
```

which produces:

```
extracted -> 3.14
remaining ' is Pi'

extracted -> 0.000000
remaining 'Pi is 3.14'

extracted -> 1.100000
extracted -> 2.200000
extracted -> 3.300000
```

Note by reassigning the unparsed portion of the string `remaining` to `input`, all numbers may be parsed. The

parsing functions ***skip leading and trailing blanks***; if there were non-blank alpha characters between the numbers, the parsing functions would stop when reaching these non-blank characters.

- Converting strings to ints/longs/long ints, signed or unsigned

The definition of `strtol` is shown. The definitions for the functions `strtof` and `strtold` are similar except for the data type of the returned value (`float`, `long double`):

```
long int strtol(const char *str, char **endptr, int base)
```

Note that, like `strtod`, the second parameter `ept` is a *pointer to a pointer*.

The functions that convert string data to integer types takes a third parameter - the `base` of the result. If the value of base is zero, the syntax expected is similar to that of integer constants, which is formed by a succession of:

```
An optional sign character (+ or -)
An optional prefix indicating octal or hexadecimal base ("0" or "0x"/"0X" respectively)
A sequence of decimal digits (if no base prefix was specified) or either octal or
hexadecimal digits if a specific prefix is present
```

The program `strtol` repeatedly parses numbers in different bases and prints the results:

```c
/* strtol example */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char inputs[] = "2001 1100100";
    char *remaining;
    long first, second;

    first = strtol (numbers, &remaining, 10);
    second = strtol (remaining, &remaining, 2);

    printf ("Decimal values -> %d and %d.\n", li1, li2);
}
```

which produces:

```
Decimal values -> 2001 and 100.
```

If there were non-blank alpha characters between the numbers, the parsing would stop.

qsort()

The `qsort` function performs a quicksort on the data (array) you supply. It requires four arguments. The definition and parameter descriptions follow:

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const
void*))
```

`base`

Address of the data to be sorted (pointer to first element)

`nitems`

How many items to be sorted

`size`

The `sizeof` each item

`(compar)(const void *, const void)`

Address of a *comparison function* (a **function pointer**)

The *comparison function* accepts two arguments that represent *pointers to elements to be sorted*. The comparison function returns an `int`.

Given this definition of a comparison function:

```
int comparator(const void* p1, const void* p2);
```

The return value means:

<0: The element pointed by p1 goes before the element pointed by p2 (p1 'is less than' p2)
0 : The element pointed by p1 is equivalent to the element pointed by p2 (p1 'equals' p2)
>0: The element pointed by p1 goes after the element pointed by p2 (p1 'is greater than' p2)

`qsort` is *destructive*; the original array is replaced by a sorted version.

The following program `qsort_ex.c` shows sorting an array of strings and an array of numbers:

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

// We will intentionally have our comparator return in descending order
int double_comparator(const void *a, const void *b)
{
    return *(double *)b - *(double *)a;
}

// Our comparator for strings will compare lengths of a and b
int string_comparator (const void * a, const void * b ) {
    int a_len, b_len;

    a_len = strlen(a);
    b_len = strlen(b);

    // return whether or not a is longer than b
    return (a_len - b_len);
}

int main() {
    // Sort an array of strings:
    const char *drivers[] = {"charles","lando","sebastian","fernando"};
    int doubles[4] = {0.0,3.3,1.1,2.2};

    // use qsort with the appropriate comparators
    qsort(string_arr, 4, sizeof(char *), string_comparator);
    qsort(doubles, 4, sizeof(nums[0]), double_comparator);

    printf("\nSTRINGS: ");
    for (int i=0; i < 4; ++i)
        printf("%s,", string_arr[i]);

    printf("\nDOUBLES: ");
    for (int i=0; i < 4; ++i)
        printf("%d,", doubles[i]);

}
```

The output follows:

```
STRINGS: lando,charles,fernando,sebastian

DOUBLES: 3.3,2.2,1.1,0.0,
```

Recall that function pointer argument and return types *must match those coded in the function definition*. The comparison function expects arguments of `const void  which, inside the function, are *cast to types` that may be used to generate an integer return value. The integer return value follows the *< 0*, *== 0*, *> 0* shown above.

For the comparison function `comp_strings` shown above, the string arguments representing two strings of the string array must be cast *from const void* * to a string (char *) in order to apply the `strcmp` function.

The comparison function takes *pointers to two objects* that represent two elements of the items to be sorted. The argument type passed to the `comp_strings` function is `const char ; a pointer to a string. Once cast, the char` pointer *must be dereferenced* to get to the underlying `const char *` type needed for the `strcmp` function.

Comparison functions that compare numbers usually return the result from subtraction. The `const void *` arguments to the `compare_nums` function must be cast to `int *`, then dereferenced before the underlying argument data (two elements of the numeric array `nums`) may be subtracted.

bsearch()

The `bsearch` function performs a binary search on an array. **The array must be sorted first.** `bsearch` returns a `void *` type.

```
void * bsearch(const void *key, const void *base, size_t nitems, size_t size, int
(*compar)(const void *, const void *))
```

The `bsearch` function requires five arguments:

`key`
: Address of the key to be found

`base`
: Address of the data to be sorted (pointer to first element)

`nitems`
: How many items to be sorted

`size`
: The `sizeof` each item

`(compar)(const void *, const void)`
: Address of a *comparison function* (a **function pointer**)

This function returns a pointer to an entry in the array that matches the search key. If key is not found, a NULL pointer is returned.

Note the arguments to the right of `key` have the same meaning as the arguments for `qsort`. The comparison function used for `qsort` and `bsearch` is the same.

The program `bsearch_ex.c` searches the same arrays shown in the `qsort` example above. Note the compare functions ***are the same*** for the sort and search.

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

// We will intentionally have our comparator return in descending order
int double_comparator(const void *a, const void *b)
{
    return *(double *)b - *(double *)a;
}
```

```
// Our comparator for strings will compare lengths of a and b
int string_comparator (const void * a, const void * b ) {
    int a_len, b_len;

    a_len = strlen(a);
    b_len = strlen(b);

    // return whether or not a is longer than b
    return (a_len - b_len);
}

int main() {
    // Sort an array of strings:
    const char *drivers[] = {"charles","lando","sebastian","fernando"};
    int doubles[4] = {0.0,3.3,1.1,2.2};

    // set some strings, findable in arr and unfindable
    const char string_to_find = "fernando";
    const char string_unfindable = "barrichello"

    // set some doubles, findable in arr and unfindable
    double d_to_find = 1.1;
    double d_unfindable = 0.9;

    // use qsort with the appropriate comparators
    // ensure the arrays are sorted prior to bsearching
    qsort(string_arr, 4, sizeof(char *), string_comparator);
    qsort(doubles, 4, sizeof(nums[0]), double_comparator);

    printf("\nSTRINGS: ");
    for (int i=0; i < 4; ++i)
        printf("%s,", string_arr[i]);

    printf("\nDOUBLES: ");
    for (int i=0; i < 4; ++i)
        printf("%d,", doubles[i]);

    // Perform the bsearch
    if ( (char**)bsearch(&string_to_find, drivers, 4, sizeof(char *), string_comparator)
!= NULL) {
        printf("found fernando.")
    } else {
        printf(":(");
    }

    if ( (char**)bsearch(&string_unfindable, drivers, 4, sizeof(char *),
```

```
string_comparator) == NULL) {
        printf("could not find barrichello.")
    } else {
        printf(":(");
    }

    ...

    if ( (double*)bsearch(&d_to_find, doubles, 4, sizeof(doubles[0]), double_comparator)
!= NULL ) {
        printf("found 1.1");
    } else {
        printf(":(");
    }

     if ( (double*)bsearch(&d_unfindable, doubles, 4, sizeof(doubles[0]),
double_comparator) == NULL ) {
        printf("could not find 0.9");
    } else {
        printf(":(");
    }

    // (we do not want to yield the sad face in output)
}
```

The *return values of bsearch* must be cast to the appropriate type. When searching the character array `string_arr`, the return value is a *pointer within the array* where the key was found. For a string array, the correct type is `char **`. For an integer (or other non-pointer types), the correct type is `int *`.

The return value is checked using the following template:

```
key_type key = // an appropriate value
key_type * return_value_from_bsearch;
return_value_from_bsearch = (key_type *) bsearch(key, ...)

if (return_value_from_bsearch != NULL)
    // Reference the found key if needed by dereferencing return_value_from_bsearch
(*return_value_from_bsearch)
```

unistd.h

In the C programming languages, `unistd.h` is the name of the header file that provides access to the POSIX operating system API. It is defined by the POSIX.1 standard, the base of the Single Unix Specification, and should therefore be available in any POSIX-compliant operating system and compiler.

The `unistd.h` header defines miscellaneous symbolic constants and types, and declares miscellaneous functions.

Providing an exhaustive list of items available from `unistd.h` is not practical as there are hundreds of items defined in `untstd.h`. This lesson explores some functions in `unistd.h` in subsequent chapters.

getopt(), optarg, optind, opterr

The getopt function, defined in unistd.h, is part of any Standard C Library implementation that follows a POSIX standard. The general idea is that options use a format starting with - followed by a *single letter* to indicate something about what the user wants the program to do. As an example, many programs on Linux will have a -v option which instructs the program to print more verbose console output, or a -h option to print help on using the program.

getopt allows for more sophisticated command line parsing than using functions in the string.h header.

The getopt function accesses defined variables in unistd.h that represent the current internal state of its parsing system. The function and state variable definitions are:

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

The state variables are:

optind: The index value of the next argument that should be handled by the getopt() function.
opterr: Allows programmer control if getopt prints errors to the console.
optopt: If getopt does not recognize the option being given, optopt will be set to the character it did not recognize.
optarg: Set by getopt to point at the value of the option argument, for those options that accept arguments.

The first two arguments to getopt are the arguments passed to the main function. The third argument optstring is a *string of known options*. getopt fetches command line arguments until it exhausts the argument list.

getopt recognizes command line options *characters preceded by a* -. Any arguments passed as *not options* are considered *extra arguments* and are accessible. getopt returns the option if known, a ? for unknown options and -1 when it has processed all options.

Any program using getopt *need not declare these state variables*. Since they are declared external, their definition is already known to programs that include untstd.h.

The following program getopt_ex1.c shows using getopt:

```c
// Program to illustrate the getopt()
// function in C
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int opt;

    while((opt = getopt(argc, argv, "irx")) != -1)
    {
        if (opt == 'i') {
            printf("Option selected -> %c\n", opt);
        } else if (opt == 'r') {
            printf("Option selected -> %c\n", opt);
        } else if (opt == 'x') {
            printf("Option selected -> %c\n", opt);
        }
        else if (opt == '?') {
            printf("Unknown option: %c\n", optopt);
        }
    }

    // optind is for the extra arguments which are not parsed
    for(; optind < argc; optind++){
        printf("extra arguments: %s\n", argv[optind]);
    }
}
```

This program parses command line arguments which in this program are *characters preceded by a* - and 'processes' the option. Any arguments passed as *not options* are considered *extra arguments* and are printed as well.

Some executions of the above program are shown below:

```
$ cc -Wall -Wextra get*1.c -o get

$ ./get -i -f -x
Option selected -> i
./get: invalid option -- 'f'
Unknown option: f
Option selected -> x

$ ./get -i -f -x 'not an option' -i
Option selected -> i
./get: invalid option -- 'f'
Unknown option: f
Option selected -> x
Option selected -> i
extra arguments: not an option
```

The program as coded does not require any arguments and none of the options require a parameter. Note that the same option (`-r`, in this example) may be passed multiple times in the command line (`-irfrx`, in this example).

The options may be passed separately on the command line (`-i -f -x`) or combined (`-xri`).

Arguments that are not prefixed with `-` are considered _extra arguments (`./get -i -f -x 'not an option' -i` - 'not an option' is extra). These extra arguments are accessible as any other command line argument through the proper indexing of the `argv` array. The state variable `optidx` changes as `getopt` iterates over the command line arguments and processes options.

`getopt` is 'smart' inasmuch as it accepts arguments *in any order*, even for argument lists that contain a combination of valid (known) and invalid arguments, as well as extra arguments.

- Suppressing `getopt` diagnostics

Set the `opterr` state variable `0` to *suppress diagnostics from* `getopt`.

The above program with `opterr = 0;` coded in `main` suppresses the *invalid option* diagnostic shown above.

Building the program with `opterr = 0;` and executed with an invalid option on the command line produces:

```
$ ./get not_an_option -xtr
Option selected -> x
Unknown option: t
Option selected -> r
extra arguments: not_an_option
```

The program handles the unknown option by printing its own message. Note the previous diagnostic from `getopt` (./get: invalid option — 'f') is not printed.

- Tell `getopt` that an option requires a value

Code a colon (`:`) after the option in the third parameter of `getopt`. The value coded *immediately to the right* of the argument requiring a value is stored in the `optarg` state variable.

The program `getopts_ex2.c` is coded to require that a user entering the `-f` option supply a value.

```c
// Program to illustrate the getopt()
// function in C

#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int opt;

    while((opt = getopt(argc, argv, "if:r")) != -1)
    {
        if (opt == 'i' || opt == 'l' || opt == 'r') {
            printf("Option selected -> %c\n", opt)
        } else if (opt == 'f') {
            printf("Option selected -> f, filename -> %s\n", optarg);
        } else if (opt == '?') {
            printf("unknown option: %c\n", optopt);
        }
    }

    // optind is for the extra arguments which are not parsed
    for(; optind < argc; optind++){
        printf("extra arguments: %s\n", argv[optind]);
    }
}
```

Several runs of this program follow:

```
$ cc -Wall -Wextra get*2.c -o get2

$ ./get2 -if "filename.txt" -r
Option selected -> i
Option selected -> f, filename -> filename.txt
Option selected -> r

$ ./get2 -f "for opt f" -d -f "other f option val"
Option selected -> f, filename -> for opt f
./get2: invalid option -- 'd'
unknown option: d
Option selected -> f, filename: other f option val
```

Whatever string is coded after an option requiring a value will be used, even another (valid or invalid) option, as shown above.

stdarg.h

There are cases where a function needs to accept varying numbers of arguments of varying type. Often, such functions are called *variadic functions*.

`stdarg.h` is a header in the C standard library that allows functions to accept an indefinite number of arguments. `stdarg.h` define *macros* that can be used to access the arguments of a list of unnamed (arguments with no corresponding parameter declarations) arguments.

The next chapter discusses these macros.

..., va_list, va_start, va_copy, va_arg, va_end

In short, the variadic function has a special definition and code using the *macros* `va_start`, `va_copy` and `va_arg` to access and process the passed arguments.

The definition of a variadic function must include an *argument usually required for processing of the remaining arguments* and *ellipses* (...):

```
int add_em_up (int count, ...);
```

The parameter `count` is required to process the remainder (varying) arguments. In the example that follows, `count` (not a reserved name) represents the number of elements passed to the function. The function call to `add_em_up` **must include the count, coded as the first argument**.

For example, the call:

```
add_em_up(5, 1, 34, -45, 25, 5);
```

will assign a value of `5` of `count`.

The values `1, 34, -45, 25, 5` will be accessed by the special macros in `stdarg.h` as shown in the upcoming program example.

The macros `va_start`, `va_end`, and `va_end` are used to process the arguments accessed from the `va_list` object created within the function.

The program `variadic_ex1.c` is the classic example of adding numbers with a variadic function and is shown below:

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

int variadic_multiply (int count,...)
{
  int i, product, curr;

  // initialize variadic list from parameters
  va_list variadic_list;
  va_start(variadic_list, count);

  product = 1;
  for (i = 0; i < count; ++i) {
      // get next value in list using va_arg
      curr = va_arg (variadic_list, int);

      // multiply product by the next value in the list
      product  = product * curr;
  }

  va_end(variadic_list);
  return product;
}

int main (void)
{
  /* This call prints "product -> 6". */
  printf("product -> %d\n", variadic_multiply(3, 1, 3, 2));
}
```

which outputs:

```
product -> 6
```

A variadic function may take additional, non-variadic parameters. These additional parameters must be coded *before* the *argument count* and the ellipses (...).

- Type information for arguments *not passed*

There is *no type information* passed with the varying arguments. The variadic function must know the types of the arguments.

Replacing the last line of the above program with:

```
printf ("%d\n", variadic_multiply (3, 1, "hamilton"));
```

and running the program prints:

```
278923519
```

The results are indeterminate. There is no diagnostic from the compiler.

- The first argument to the variadic function should be the second argument to `va_start`

The expression:

```
va_start(ap, count);
```

does not initialize a list of 10 elements.

Changing the above line to:

```
va_start (ap, 10);
```

and rebuilding, `cc` complains:

```
variadic_ex1.c: In function ‘add_em_up’:
variadic_ex1.c:9:3: warning: second parameter of ‘va_start’ not last named argument [-
Wvarargs]
   va_start (ap, 10);         /* Initialize the argument list. */
   ^~~~~~~~
```

The code for `va_start` requires that the ***second arg*** (`count`, in this case) is coded as the _second argument to _`va_start`.

The program `variadic_ex2.c` shows processing a variadic list that uses a list terminator of `0`. The required first argument to the variadic function is not used to govern or direct processing in any way. In this example, the first, ***required***, argument to the variadic function is treated as the first argument to be processed.

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

int variadic_multiply (int first,...)
{
  int i, product, curr;

  // initialize variadic list from parameters
  va_list variadic_list;
  va_start(variadic_list, first);

  product = 1;
  curr = first;

  while (curr != 0) {
      product = product * curr;
      curr = va_arg (variadic_list, int);
  }

  va_end(variadic_list);
  return product;
}

int main (void)
{
  /* This call prints "product -> 18". */
  printf("product -> %d\n", variadic_multiply(3, 1, 3, 2, 0)); // things added after the
0 will be ignored
}
```

The output:

```
product -> 18
```

Rather than pass a count, the variadic function stops processing when the function fetches `0` from the list. The ***required*** parameter for the variadic function is treated as a value to be included in the sum. Note how this parameter is used when initializing the list:

```
va_start(arg_list, required_but_just_the_first_arg);
```

All proccessing of the variadic argument list is done between the `va_start` and `va_end` macros.

The template for variadic parameter processing is:

```
ret_val_type_or_void variadic_function(<other parameters not variadic>, <some_type>
parm_usually_required_to_process_elements, ...)
{
    va_list my_arg_list;
    va_start(my_list, parm_usually_required_to_process_elements);
    //

    {
        // Access/process element of my_arg_list with va_arg(my_arg_list,
<type_of_parameter>);

    }
    va_end(my_arg_list);

}
```

There is an additional function `va_copy` that makes a copy of a `va_list` object:

```
void va_copy (va_list dest, va_list src)
```

the `src` list must be initialized with `va_start` and the copy should be 'ended' with a call to `va_end` when no longer needed.

time.h

The time.h header defines several variable types, two macro and various functions for manipulating date and time.

Some types defined in time.h are clock_t, time_t, struct tm, and struct timespec.

clock_t: This is a type suitable for storing the processor time. clock_t is an *arithmetic* type (either an integer or floating point type).

The program clock_t_example.c uses the clock function, which returns a clock_t value, to time the CPU time of a piece of code:

```
// clock(), clock_t example
#include<stdio.h>
#include<time.h>

void perform_job(void)
{
    ...
}

int main(void)
{
    clock_t start = clock();
    // Time this function
    perform_job( );
    clock_t end = clock() - start;

    // print elapsed CPU time in terms of Clock cycles per second of CPU.
    printf("Processing took %f seconds\n", (double)end / CLOCKS_PER_SEC);
}
```

which outputs: (where X and Y are unsigned integer values)

```
...

Processing took X.Y seconds
```

clock_t represents an amount of CPU time used since a process was started. It can be converted to seconds by dividing by CLOCKS_PER_SEC (a macro from time.h). Its real intent is to represent CPU time used, not calendar/wall clock time.

`time_t`: This is a type suitable for storing the calendar time. A variable of type `time_t` holds the number of seconds between a call to the `time()` function and the ***epoch*** (January 1st, 1970). The next chapter has an example of using the `time` function.

`struct tm`: This is a structure used to hold the time and date.

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

The function `gmtime`, described in the next chapter, loads an instance of the `struct tm` with appropriate values.

`struct timespec`: Represents a simple calendar time, or an elapsed time, with sub-second resolution.

```
struct timespec {
    time_t tv_sec;      /* elapsed time in whole seconds*/
    long   tv_nsec;     /* the rest of the elapsed time in nanoseconds */
};
```

The function `timespec_get`, described in the next chapter, loads an instance of `struct timespec` with appropriate values.

time, localtime, gmtime, timespec_get

- time

This function returns the time since 00:00:00 UTC, January 1, 1970 (Unix timestamp) in seconds. If second is not a null pointer, the returned value is also stored in the object pointed to by second.

```
time_t time(time_t *second)
```

The program time_t_ex.c using the time function prints the number of hours between the current (wall) time and the epoch:

```
// time(), time_t example
#include<stdio.h>
#include<time.h>

int main (void) {
   time_t s;

   s = time(NULL); // time(&seconds) also works
   printf("Minutes since birth of UNIX -> %ld\n", s/60);
}
```

which prints:

```
Minutes since birth of UNIX -> 27014208
```

- localtime

The declaration for localtime() function:

```
struct tm *localtime(const time_t *timer)
```

localtime uses the time pointed by timer to fill a struct tm with the values that represent the corresponding local time. The value of the argument timer is broken up into the struct tm and expressed in the local time zone.

The program localtime_ex.c shows an example of thte localtime function in use:

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

int main (void)
{
   time_t raw;
   struct tm *t;

   time(&raw);
   info = localtime(&raw );
   printf("Date and time\t");
   printf("%d/%2d, %04d\n - %2d:%02d:%02d\n", (t->tm_mon) + 1, t->tm_mday, t->tm_year +
1900, t->tm_hour, t->tm_min, t-> tm_sec);
}
```

which outputs:

```
Date and time    9/12, 2020 - 0:03:39
```

The time reported is 12:03 AM

Note the offsets for `t -> tm_mon` and `t -> tm_year`. The month field in `struct tm`, `tm_mon` starts at 0; the year field `tm_year` is the number of years since `1900`.

- `gmtime`

Following is the declaration for gmtime() function.

```
struct tm *gmtime(const time_t *timer)
```

`gmtime` uses the value pointed by the argument `timer` to fill a `struct tm` with the values that represent the corresponding time, expressed in Coordinated Universal Time (UTC) or GMT timezone.

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

enum {PST = -8};

int main (void)
{
   time_t rawtime;
   struct tm *t;

   time(&rawtime);
   /* Get GMT time */
   t = gmtime(&rawtime );
   printf("Los Angeles Local Time : %2d:%02d\n", (t->tm_hour+PST)%24, t->tm_min);
}
```

which outputs:

```
Los Angeles Local Time : 12:53
```

Adding hours to `t->tm_hour` to generate times for the different time zones may result in an hour value > 23, hence, the *mod 24* operation on the hour fields.

- `timespec_get`

  ```
  int timespec_get(struct timespec *ts, int base)
  ```

the `timespec_get` function modifies the timespec object `ts` to hold the current calendar time in the time `base`. For a base value of the constant `TIME_UTC`, `ts->tv_sec` is set to the number of seconds since an implementation defined epoch, truncated to a whole value and `ts->tv_nsec` member is set to the integral number of nanoseconds, rounded to the resolution of the system clock.

The value of `TIME_UTC` is *implementation defined*.

The program `timespec_ex.c` shows the elapsed time in executing a function.

```c
// timespec_get function example
#include <stdio.h>
#include <time.h>

void perform_job(void)
{
    ...
}

int main(void)
{
    struct timespec start, end;

    // We want to measure 'perform_job'
    timespec_get(&start, TIME_UTC);
    perform_job();
    timespec_get(&end, TIME_UTC);

    time_t s = end.tv_sec - start.tv_sec;
    long n = end.tv_nsec - start.tv_nsec;
    if (n < 0) {
        printf("%ld.%09ld\n", (long)s - 1, n + 1000000000);
    } else {
        printf("%ld.%09ld\n", (long)s, n);
    }
}
```

which prints: (where X and Y are unsigned integer values)

```
X.Y
```

Exercises

Exercise 1

Write a program `main.c` with one function, `main`, which fulfills the following criteria. Ensure that you are importing the C standard library.

- Take input from the user in the form of a string, and convert it to an integer, then store that in the integer variable `max_size`. You may assume that it will be positive.
- Create an integer array of size `max_size` and fill it with randomly generated integers from -1000 to 1000 (inclusive). You may use the `rand` function available to you in the C standard library, which you will have to import. You will call this array `arr`.
- Take input from the user in the form of a string, and convert it to an integer. You may assume that it will either be 0, 1, or 2. Store this in the integer variable `choice`.
- Take another input from the user in the form of a string, and convert it to an integer. You may assume that it will be from -1000 to 1000 (inclusive). Store this in the integer variable `target`.
- If `choice` is 0, perform a linear search on your generated array `arr`, and print to standard out the time it took for your linear search to find `target`, or search the array exhaustively without finding it. You will have to implement the linear search yourself. Ensure that you are using the timing code detailed in the lesson.
- If `choice` is 1, perform the C standard library `qsort` function on the array of integers to sort them in descending order. You are expected to create your own comparator function for this purpose. Then, perform your linear search on the array until you find `target` or fail to do so. Print the total time it took for your code to perform the sorting, as well as the linear search, to standard out. Ensure that you are using the timing code detailed in the lesson.
- If `choice` is 2, use `qsort` to sort the array `arr`. Then, use `bsearch` to find `target` in the sorted array. Regardless of whether the search returned successfully or not, print the time it took to perform the sort and search to standard out. Ensure that you are using the timing code detailed in the lesson.

Ensure that you are properly timing the code's critical sections. Refer to the lesson for more guidance.