ModuleJ - C Programming II

Release v2021.5.3

United States Army Cyber School

Sep 29, 2021

CONTENTS:

CHAPTER

ONE

MODULE J - C PROGRAMMING II - BLOCK 1

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.

1.1 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.

```
file.c   file.cpp   file.s   file.o   file.exe
       ->         ->       ->       ->
      cpp         cc       as       ld
```

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.

1.1.1 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.c`.

```
int main(void)
{
        printf("The result is %d\n", calculate());
}
```

`library.c`.

```
int calculate(void)
{
        return 8 * 6 + 7 / 5 - 3 + 0 * 9;
}
```

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

```
$ make program
cc -Wall -Wextra -Wpedantic    program.c   -o program
program.c: In function ‘main’:
program.c:5:31: warning: implicit declaration of function ‘calculate’ [-Wimplicit-
 ↪function-declaration]
    5 |  printf("The result is %d\n", calculate());
      |                               ^~~~~~~~~
/usr/bin/ld: /tmp/ccRJtZMx.o: in function `main':
program.c:(.text+0xe): undefined reference to `calculate'
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`. Recall that a function needs to be declared or defined before it can be invoked. `main` wants to call the `calculate` 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 += -Wall -Wextra -Wpedantic

program: program.o library.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
cc -Wall -Wextra -Wpedantic   -c -o program.o program.c
program.c: In function ‘main’:
program.c:5:31: warning: implicit declaration of function ‘calculate’ [-Wimplicit-
 ↪function-declaration]
    5 |  printf("The result is %d\n", calculate());
      |                               ^~~~~~~~~
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
The result is 46
```

1.2 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.c`: Unpreferred; Writing declaration in file where function is invoked.

```
#include <stdio.h>

int calculate(void);

int main(void)
{
        printf("The result is %d\n", calculate());
}
```

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("The result is %d\n", calculate());
}
```

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(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.

1.2.1 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.

1.3 External Linkage

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

`library.h`: Wrong; creates storage for `global_state`.

```
#ifndef LIBRARY_H
#define LIBRARY_H

char global_state;

int calculate(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.o library.o other.o  -o program
/usr/bin/ld: library.o:(.data+0x0): multiple definition of `global_state'; program.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.h`: Correct; does not create storage for `global_state`.

```
#ifndef LIBRARY_H
#define LIBRARY_H

extern char global_state;

int calculate(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.

1.4 Interal 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.c`.

```
static int secret_function(void)
{
        return 8;
}

int calculate(void)
{
        return secret_function() * 6 + 7 / 5 - 3 + 0 * 9;
}
```

1.5 Exercises

:!exercise:

1.5.1 Exercise 1

The `math.h` header file has functions for `pow`, `sqrt`, and `fabs`.

- Write a program that invokes these functions.
  - The program should print the return value of these functions.
  - The program must compile without errors or warnings.

1.5.2 Exercise 2

Write a program that uses preprocessor commands to define 3 values for a minimum limit, maximum limit, and pi.

- The program should print all three values.

1.5.3 Exercise 3

Modify the previous program to include several comments.

- Compile the program so that only the preprocessor is run.
  - What is the resulting file?
  - What happened to the comments?
  - What happened to the `#define` statements?

1.5.4 Exercise 4

Write a program that contains a function that prints the value of a number.

- After printing the number, the number should be incremented.
- The program should invoke the function several times.
- Run the program.
- Modify the program so that the function maintains the value of the variable after it has been incremented.
  - For example, if the number is 2, 2 is printed then is incremented to 3. The next time the function is invoked 3 should be printed.

1.5.5 Exercise 5

Write a header file that has a function prototype as well as a variable. The prototype is:

```
void my_proto(int x);
```

- Create a .c file that implements the function by printing the value of its argument.
- Create a second program that invokes the program. The second program should pass the variable defined in the header file.
- Compile and run the second program.

The output should resemble:

```
my_proto is 999
my_proto is 999
my_proto is 999
my_proto is 999
my_proto is 999
```

1.5.6 Exercise 6

Copy `do_arithmetic_with_more_funcs.c` found in `starter_code/compilation` into `do_arithmetic_with_header.c` and change `do_arithmetic_with_header.c` to use a **header file containing the needed function declarations**.

Create a directory in your work directory named `part6`. Place all files used for this part of the exercise in this folder.

Move the code for *all functions* in `do_arithmetic_with_header.c` to a (new) file named `arithmetic_print_funcs.c`

Code a **header file** named `print_funcs.h` that will contain *function prototypes* for **only** the functions called in the **`main` function in `do_arithmetic_with_header.c`**.

When done copying and editing, the folder `part6` should have **three files**:

```
do_arithmetic_with_header.c  main function that calls the needed functions
arithmetic_print_funcs.c     Source for the print and arithmetic functions
print_funcs.h                Header file with function prototypes
```

Use `cc` from the `part6` directory to compile all the source files and create an executable.

The output should be:

```
The SUM of the numbers 10 and 100 is 110
The DIFFERENCE of the numbers 10 and 100 is -90
The SUM of the numbers 1000 and 150 is 1150
The DIFFERENCE of the numbers 1000 and 150 is 850
```

1.6 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.

1.7 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.

Get the address of a piece of data with `&`.

```
int main(void)
{
        int value = 13;
        printf("value = %d, &value = %p\n", value, &value);
}
```

On most systems, this will print someting similar to:

```
value = 13, &value = 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`.

```
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.

WRONG: Cannot take address of literal value.

```
int main(void)
{
        //ERROR: '17' is not a value in memory
        printf("%p\n", &17);
}
```

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

Unpreferred: `*&` cancel each other out.

```
int main(void)
{
        int value = 13;
        printf("%p: %d\n", &value, *&value);
}

0x7ffc30f879d4: 13
```

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 it.

1.8 Pointer Declarations

Pointers are addresses. Addresses are 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 *px;
```

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 `*px` should yield type `int`. The human interpretation of this is "``px`` is a pointer to ``int``".

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.

```
int main(void)
{
        int value = 13;
        int *px;
        px = &value;

        printf("%d\n", *px); // Prints "13"
}
```

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

WRONG: Pointer and data types do not match.

```
int main(void)
{
        double value = 3.14;
        int *px;
        //ERROR: types do not agree
        px = &value;

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

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

Pointers may store the address of any instance of their type.

```
int main(void)
{
        double pi=3.14, e=2.72, ln10=2.30;

        double *current;

        current = &pi;
        printf("%lf\n", *current); // Prints "3.14"
        current = &ln10;
        printf("%lf\n", *current); // Prints "2.30"
        current = &e;
        printf("%lf\n", *current); // Prints "2.72"
}
```

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

Pointers may be initialized.

```
int main(void)
{
        int value = 19;
        int *px = &value;
```

(continues on next page)

(continued from previous page)

```
        printf("%d\n", *px); // Prints "19"
}
```

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.

Pointers may be used to modify what they point at.

```
int main(void)
{
        int value = 19;
        int *px = &value;

        printf("%d\n", value); // Prints "19"
        *px = 23;
        printf("%d\n", value); // Prints "23"
}
```

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

Multiple pointers may point to the same data.

```
int main(void)
{
        int value = 29;
        int *px = &value;
        int *py = &value;

        printf("%d : %d\n", *px, value); // Prints "29 : 29"
        *py = 31;
        printf("%d : %d\n", value, *px); // Prints "31 : 31"
}
```

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;
}
```

1.9 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`.

null.c.

```
```

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

1.10 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 arguments can manipulate data outside the function's scope.

```
void square(int *num)
{
        if (!num) {
                return;
        }

        // This changes the data pointed at by num
        *num *= *num;
}

int main(void)
{
        int favorite_number = 37;

        square(&favorite_number);
        printf("%d\n", favorite_number); // Prints "1369"
}
```

1.10.1 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.

```
// Calculates the positive and negative root of input
// pos and neg are output parameters
void two_roots(double input, double *pos, double *neg)
{
        double root = sqrt(input);
        if (pos) {
                *pos = root;
        }
        if (neg) {
                *neg = -root;
        }
}
```

1.11 `struct` Pointers

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

```
struct mrap {
        int mass_kg;
        int width_cm;
        int height_cm;
        int length_cm;
        int max_speed_kph;
        int id_number;
};

struct tank {
        int mass_kg;
        int width_cm;
        int height_cm;
        int length_hull_cm;
        int length_fwd_gun_cm;
        int max_speed_kph;
        int id_number;
};
```

If a `struct tank` 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. 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 tank_print(struct tank t);

int main(void)
{
        struct tank abrams = { ... };
        // 'abrams' is copied, field for field, to 't': t = abrams;
        tank_print(abrams);
}
```

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 tank_print(struct tank *t);

int main(void)
{
        struct tank abrams = { ... };
        // The address of 'abrams' is copied to 't': t = &abrams;
        tank_print(&abrams);
}
```

1.12 \–> Operator

Working with ``struct``s presents a problem: member access.

WRONG: Compiler error, failing to derefence.

```
void tank_print(struct tank *t)
{
        //ERROR: Type `struct tank *` has no member `height_cm`
        printf("%dcm\n", t.height_cm);
}
```

This error happens because *pointers* do not have fields, *``struct``*s 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 tank_print(struct tank *t)
{
        //ERROR: Type `struct tank *` has no member `height_cm`
        printf("%dcm\n", *t.height_cm);
}
```

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

```
        //ERROR: Type `struct tank *` has no member `height_cm`
        printf("%dcm\n", *(t.height_cm));
```

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 tank_print(struct tank *t)
{
        printf("%dcm\n", (*t).height_cm);
}
```

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 tank_print(struct tank *t)
{
        printf("%dcm\n", t->height_cm);
}
```

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 tank_commander {
        char *name;
        struct tank *tank;
};

void tank_commander_print(struct tank_commander *tc)
{
        printf("%dcm\n", tc->tank->height_cm);
}
```

1.13 Exercises

:!exercise:

1.13.1 Exercise 1

Write a program that declares and initializes a double, int, and a char.

- Create pointers to each of these variables.
- The program should print the following information for all of the variables:
  - The address
  - The value
  - The size in bytes

1.13.2 Exercise 2

Write a program that prompts for three numbers.

- Print the sum of the three numbers.
  - The program must use pointers to calculate the sum.

1.13.3 Exercise 3

Write a program that contains a function to swap two integers and another that swaps two characters.

- The program should print the before and after values to verify the swap occurred.
  - The program must use pointers.

Sample output:

```
BEFORE: 10 5
AFTER:  5 10
BEFORE: a b
AFTER:  b a
```

1.13.4 Exercise 4

Write a function with the following signature:

```
int calculateLength(char *);
```

which accepts a string from the console and prints the string length.

The function should determine the length by counting the characters *using pointers*.

Sample output follows:

```
Pointer : Calculate the length of the string :
 ---------------------------------------------------
Input a string : this is a string
The length of the given string this is a string
is : 16
```

1.13.5 Exercise 5

Write a function with the following signature:

```
void to_upper_case(char *)
```

that takes a string argument, converts the argument string to upper case, and prints the converted string.

Access each character in the string with a pointer.

The output should be:

```
Lower case string: this is a lower case string
Upper case string: THIS IS A LOWER CASE STRING
```

1.13.6 Exercise 6

Write a program that contains a `main` function that calls a function `load_struct` and a function `print_struct`. The `load_struct` function loads an instance of this structure:

```
struct address {
        int addr_num;           // from 1 to 2000
        char * street_name; // max 20 characters
        char * city;            // max 20 characters
        char * state;           // max 2 characters;
        int zip;                        // from 906 to 99557
};
```

with data.

The `print_struct` function prints the values of the instance previously loaded with `load_func`.

Choose applicable data for the fields specified.

Both functions should accept a pointer to an instance of this struct.

The output should resemble:

```
Address :
1200 holyoke ave
Anytown, WA     97654
```

1.13.7 Exercise 7

Write a program that contains a `main` function that calls the `load_func` above and another function named `copy_struct` that copies the data from the instance loaded in `load_struct` into a new instance.

After `copy_struct` completes, have the `main` function call the function `print_struct` coded above to print both structs.

Next, *change the address in one of the structs* in the `main` function and print the two instances again to verify that the `copy_funcs` performed a *deep* copy.

The output should resemble:

```
Source Address :
1200 holyoke ave
Anytown, WA     97654
Destination Address :
1200 holyoke ave
Anytown, WA     97654

Source Address :
1200 Changed street
Anytown, WA     97654
Destination Address :
1200 holyoke ave
Anytown, WA     97654
```

1.14 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 tank tank_create(...);

int main(void)
{
        // return value is copied, field for field, to 'abrams'
        struct tank abrams = tank_create(...);
}
```

Preferred: Returning a `struct *` from a function.

```
struct tank *tank_create(...);

int main(void)
{
        // return pointer is copied to 'abrams'
        struct tank *abrams = tank_create(...);
}
```

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

1.15 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 tank *tank_create(...)
{
        struct tank t = {...};

        //ERROR: t's lifetime is over when the function returns.
        return &t;
}

int main(void)
{
        struct tank *abrams = tank_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

1.15.1 Lower On the Call Stack Than the Caller

1. Only a few instances are needed; AND
2. The object does not need to live longer than the current 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 tank_print(struct tank *t);

int main(void)
{
        // 'abrams' is on the stack frame for 'main'
        struct tank abrams = {...};
        tank_print(&abrams);
}

// 't' is a pointer to data inside the 'main' function
void tank_print(struct tank *t)
{
        printf(...);
        // At the end of the function, t 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.

1.15.2 Static Storage Duration

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 db_connection *db_connect(...)
{
        static db_connection dbh = {...};
        ...
        return &dbh;
}
```

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.

1.15.3 Heap Storage Duration

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.

1.16 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 *a);

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

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:

Arrays decay to pointers when passed as arguments.

```
void array_print_size(int a[5])
{
        printf("Size in array_print_size: %zu\n", sizeof(a)); // Prints "8"
}

int main(void)
{
        int fibonacci[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
```

(continues on next page)

(continued from previous page)

```
        array_print_size(fibonacci); // Prints "8", sizeof(&fibonacci)
        printf("Size in main: %zu\n", sizeof(fibonacci)); // Prints "36"
}
```

1.17 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.

```
void array_print_first_two(int *a)
{
        printf("First element: %d\n", a[0]);
        printf("Second element: %d\n", a[1]);

        printf("a[0] == *a → %d", a[0] == *a); // Prints "1"
}

int main(void)
{
        int fibonacci[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
        array_print_first_two(fibonacci);
}
```

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.

Sample output from `array_print(fibonacci, 5)`.

```
a + 0 → 0x7fffffffc00
a + 1 → 0x7fffffffc04
a + 2 → 0x7fffffffc08
```

(continues on next page)

(continued from previous page)

```
a + 3 → 0x7fffffffc0c
a + 4 → 0x7fffffffc10
```

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]`.

WRONG: This is syntactically correct but very confusing to the reader.

```
int main(int argc, char *argv[])
{
        puts("argv[0] == 0[argv] → %d\n", argv[0] == 0[argv]); // Prints "1"
}
```

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 interleave_string(const char *s, char sep)
{
        if (!s) {
                return;
        }

        // While `*s` is not the NUL byte
        while (*s) {
                printf("%c%c", *s, sep);

                // Move `s` forward one step in the character array
                ++s;
        }
}
```

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, char sep)
{
        if (!s) {
```

(continues on next page)

(continued from previous page)

```
                return;
        }

        for (size_t n=0; s[n] != '\0'; ++n) {
                printf("%c%c", s[n], sep);
        }
}
```

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.

1.18 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 tank``s. Due to array decay, this type becomes `struct tank **`.

```
void tank_foreach_print(struct tank **tanks, size_t sz);

int main(void)
{
        struct tank abrams = { ... };
        struct tank patton = { ... };
        struct tank *platoon[] = { &abrams, &patton, ... };

        tank_foreach_print(platoon, sizeof(platoon) / sizeof(platoon[0]));
}
```

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.

```
int main(int argc, char* argv[])
{
        if (argc != 2) {
                fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        }
        char *err;

        double value = strtod(argv[1], &err);
        if (*err) {
                fprintf(stderr, "Error: '%s' is not a decimal number\n", argv[1]);
                return 1;
        }
```

(continues on next page)

(continued from previous page)

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

1.19 `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 = "US Army";

        void *value = name;

        char *other_name = value;
}
```

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)
{
        int *distance = NULL;
        struct tank *t = NULL;

        ...
}
```

1.20 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 tank *abrams = tank_create(...);

        struct mrap *m;

        // Casting one pointer type to a different pointer type
        m = (struct mrap *)abrams;

        // Overlaps at the same point in the `struct` definitions
        printf("%d\n", m->height_cm); // Prints "244"
```

(continues on next page)

(continued from previous page)

```
        //ERROR: Actually uses the `length_fwd_gun_cm` field!!
        printf("%d\n", m->max_speed_kph); // Prints "977"
}
```

In the above code, the variable `m` is now a kind of interface to `abrams`, 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.

1.21 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
        printf("a == &a → %d", main == &main); // Prints '1'
}
```

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

Declaring a pointer to a function.

```
int *actual_function(int, int);
int (*pointer_to_function)(int, 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 add(short a, short b)
{
        return a + b;
}

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

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 { ADD, SUB, MUL, DIV };

int add(int, int);
int subtract(int, int);
int multiply(int, int);
int divide(int, int);

int main(void)
{
        ...

        int (*op)(int, int);

        switch (button) {
        case ADD:
                op = add;
                break;
        case SUB:
                op = subtract;
                break;
        case MUL:
                op = multiply;
                break;
        case DIV:
                op = divide;
                break;
        }

        // Syntax for invoking function through a pointer
        int result = op(operand_1, operand_2);
        ...
}
```

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.

```
// Takes a function as a parameter, and prints each element in the array
void array_print(void **a, size_t sz, void (*print_func)(void *))
{
        for (size_t n=0; n < sz; ++n) {
                print_func(a[n]);
        }
}

void mrap_print(void *arg)
{
        struct mrap *m = arg;
        ...
}

void tank_print(void *arg)
{
        struct tank *t = arg;
        ...
```

(continues on next page)

(continued from previous page)

```
}

int main(void)
{
        // Prints the 'platoon' array using 'tank_print'
        array_print(platoon, platoon_sz, tank_print);

        // Prints the 'column' array using 'mrap_print'
        array_print(column, column_sz, mrap_print);
}
```

This is much simpler than having to write a custom `tank_array_print` function and a `mrap_array_print` function, plus whatever other types are needed.

Note that both `mrap_print` and `tank_print` had signatures that took in `void *` rather than `struct tank *`. This is because function pointer type signatures must match exactly; there is no implicit conversion allowed on either its parameters or return type.

1.21.1 The Spiral Rule

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

Confusing declarations.

```
int *alfa[3];
int (*bravo)(int *arr, size_t sz);
void (*signal(int sig, void (*func)(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.

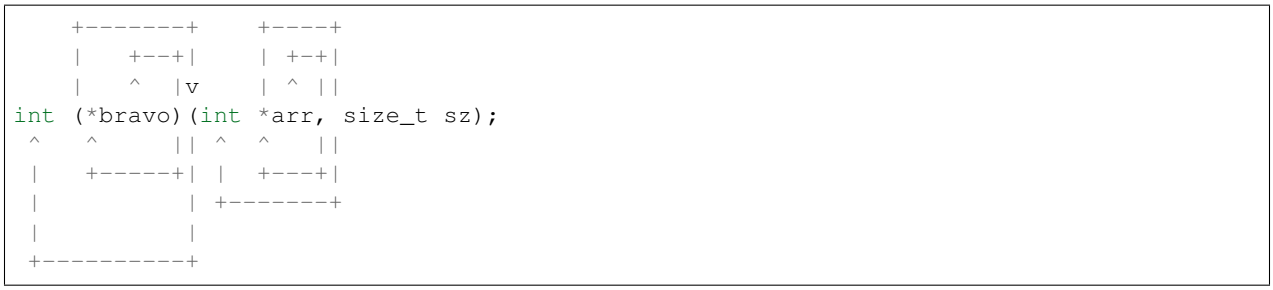
1. Begin with the unknown identifier.
2. Move in a clockwise spiral from the identifier, reading off each piece of syntax.
3. Resolve the entirety of a parenthetical before moving outside of it.

Parsing `int *alfa[3];`.

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

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

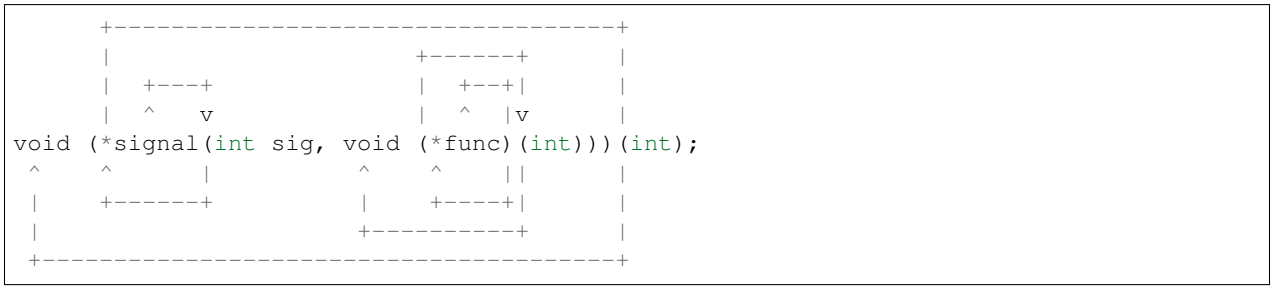
Parsing `int (bravo)(int *arr, size_t sz);`.



1. The name `bravo` is…
2. a pointer to…
3. a function that takes…
    1. the name `arr`, which is…
    2. a pointer to…
    3. an `int`…
    4. and `sz`, which is…
    5. 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 (*signal(int sig, void (*func)(int)))(int)`.



1. The name `signal` is…
2. a function that takes…
    1. the name `sig`, which is…
    2. an `int`…
    3. and `func`, which is…
    4. a pointer to…
    5. a function that takes…
        1. an `int`
        2. returning `void`
3. returning a pointer to…
4. a function that takes…
    1. an `int`

2. returning `void`

1.22 Exercises

:!exercise:

1.22.1 Exercise 1

Write a program `inter-pointers_01.c` that prints the minimum element of an array of doubles and an array of strings. The program should use **function pointers** to reference the functions that perform compares for numbers and strings.

Starter code for this exercise is in the file `starter_code/inter_pointers/inter_pointers_01.c`. The starter code has some arrays coded and comments on what could be useful.

The solution provided makes use of `void *` since return types and argument types used for functions referenced by function pointers must agree.

These arrays are coded in the starter file:

```
char * strings[] = {
                "this is string1",
                "Not string1 or string2",
                "String2",
                "fourth string",
                "A string",
                "a string"
        };

double some_nums[] = {2.5, -125.25, 3635.25, 78.45, -1000.25, 25};
```

The program output using these arrays is:

```
#strings: 6    # nums: 6
Min number is '-1000.250000'
Min string is 'A string'
```

1.22.2 Exercise 2

Write a program `inter_pointers_02.c` that fill instances of a struct with some data entered from the keyboard and some generated in the program.

When run, the output resembles:

```
Enter employee name for employee #1 (no more than 20 characters) ==> Emp A
Enter salary for 'Emp A' ==> rty678
Could not fetch salary from input 'rty678'
Enter salary for 'Emp A' ==> 258.25
Employee ID: 1  Name: Emp A      Salary: 258.250000

Enter employee name for employee #2 (no more than 20 characters) ==>
Enter salary for 'No Name Entered' ==> 258.25
Employee ID: 2  Name: No Name Entered    Salary: 258.250000
```

(continues on next page)

(continued from previous page)

```
Enter employee name for employee #3 (no more than 20 characters) ==>
Enter salary for 'No Name Entered' ==>
Could not fetch salary from input ''
Enter salary for 'No Name Entered' ==> 789.25
Employee ID: 3  Name: No Name Entered    Salary: 789.250000
```

Include some checks for valid numeric input and use the name *No Name Entered* when the user does not provide a name.

Note that the program *generates the employee ID*; the user does not provide it.

Starter code for this exercise is in the file `starter_code/inter_pointers/inter_pointers_02.c`.

The `main` function follows and is coded in the starter code file (along with the struct):

```
int main(void)
{
        for (int idx = 0; idx < 3; idx++) {
                emp_info * an_employee = create_emp_info_struct_object();
                print_employee_record(an_employee);
        }

}
```

Note, the `main` function calls the function `create_emp_info_struct_object` which returns a pointer to the struct. Also note, the *employee ID is NOT passed to any function*.

1.22.3 Exercise 3

Write a program `inter_pointers_03.c` that:

- Prompts user for three numbers representing the sizes of dimensions in a three-dimensional array of integers.
  - The entries should be *positive and less than 6*; else the output would get too large
- Creates an array using the entered dimensions.
- Loads the array with integers starting at 0 and ending with the number of array elements minus one.
  - The output below shows how the elements are arranged.
- Prints the array as shown in the output.
- Prompts the user for three indices for the row, column and plane, respectively.
- Accesses the array using *pointer arithmetic* to display the element corresponding to the entered indices.

When run, the output resembles:

```
Enter the number of rows
Entry must be non negative and < 6 (not going to check this)  ==> 5
Enter the number of columns
Entry must be non negative and < 6 (not going to check this)  ==>
Could not fetch num of elements from input ''
Enter the number of columns
Entry must be non negative and < 6 (not going to check this)  ==> wrong
Could not fetch num of elements from input 'wrong'
Enter the number of columns
Entry must be non negative and < 6 (not going to check this)  ==> 4
```

(continues on next page)

(continued from previous page)

```
Enter the number of planes
Entry must be non negative and < 6 (not going to check this)  ==> 3
Plane # 0
        Column # 0:     0               1               2               3               ␣
 ↪  4

        Column # 1:     5               6               7               8               ␣
 ↪  9

        Column # 2:     10              11              12              13              ␣
 ↪  14

        Column # 3:     15              16              17              18              ␣
 ↪  19

Plane # 1
        Column # 0:     20              21              22              23              ␣
 ↪  24

        Column # 1:     25              26              27              28              ␣
 ↪  29

        Column # 2:     30              31              32              33              ␣
 ↪  34

        Column # 3:     35              36              37              38              ␣
 ↪  39

Plane # 2
        Column # 0:     40              41              42              43              ␣
 ↪  44

        Column # 1:     45              46              47              48              ␣
 ↪  49

        Column # 2:     50              51              52              53              ␣
 ↪  54

        Column # 3:     55              56              57              58              ␣
 ↪  59

Enter the rightmost (row) index
Entry must be non negative and < 5 (not going to check this)  ==> wrong
Could not fetch num of elements from input 'wrong'
Enter the rightmost (row) index
Entry must be non negative and < 5 (not going to check this)  ==> 4
Enter the middle (column) index
Entry must be non negative and < 4 (not going to check this)  ==>
Could not fetch num of elements from input ''
Enter the middle (column) index
Entry must be non negative and < 4 (not going to check this)  ==> 1
Enter the leftmost (plane) index
Entry must be non negative and < 3 (not going to check this)  ==> 2
Array element [2] [1] [4] = 49
```

Check if the user entered a number and loop until the user enters a number. It is *not required* to check if the number is within bounds, but feel free to do so.

Note, the bottom prompts include the number for a dimension entered when the program first runs. The line prompting for a row index:

```
Entry must be non negative and < 5 (not going to check this)
```

contains the phrase `< 5`, where `5` is the number of rows entered:

```
Entry must be non negative and < 6 (not going to check this)  ==> 5
```

Starter code for this exercise is in the file `starter_code/inter_pointers/inter_pointers_03.c`. The starter code contains function names with *hints* on the function's use.

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.

1.23 `malloc`, `free`

```
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 enough storage for a filename, plus a terminating NUL
        char *filename = malloc(NAME_MAX + 1);
        // malloc() returns NULL on failure
        if (!filename) {
                perror("Could not allocate enough memory");
                return 1;
        }

        printf("Enter filename: ");
        fgets(filename, NAME_MAX + 1, stdin);
        printf("%s\n", filename);

        // Returns the memory to the Heap to be reused
        free(filename);
}
```

This version, using the Heap rather than the Stack, is preferable because a large string buffer (`filename`) 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.).

```
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`. There is a zero-tolerance policy throughout this course against memory leaks.

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

1.23.1 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 tank *tank_create(...)
{
        //ERROR: sizeof(struct tank) != sizeof(struct mrap)
        struct tank *t = malloc(sizeof(struct mrap));

        ...
        return t;
}
```

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 tank *tank_create(...)
{
        struct tank *t = malloc(sizeof(*t));

        ...
        return t;
}
```

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 tank *tank_create(...)
{
        struct tank *t = (struct tank *)malloc(sizeof(*t));

        ...
        return t;
}
```

1.23.2 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)
{
        struct tank *t = malloc(sizeof(*t));
        if (!t) {
                perror("Unable to create tank");
                return 1;
        }

        //ERROR: value is uninitialized garbage
        printf("%dcm\n", t->height_cm);
}
```

Preferred: Setting initial values first.

```
int main(void)
{
        struct tank *t = malloc(sizeof(*t));
        if (!t) {
                perror("Unable to create tank");
                return 1;
        }

        t->height_cm = 237;
        printf("%dcm\n", t->height_cm);
}
```

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.

Preferred: Avoid initialization for memory about to be overwritten.

```
int main(void)
{
        char *filename = malloc(NAME_MAX + 1);
        if (!filename) {
                perror("Unable to create filename");
                return 1;
        }

        fgets(filename, NAME_MAX+1, stdin);
```

(continues on next page)

(continued from previous page)

```
        ...
}
```

1.23.3 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.

```
int main(void)
{
        size_t array_size = 10;

        double *pies = malloc(array_size * sizeof(*pies));
        if (!pies) {
                perror("Unable to create list of pies");
                return 1;
        }

        for (size_t = n; n < array_size; ++n) {
                pies[n] = 3.141592653;
        }

        ...
}
```

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.

1.23.4 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 tank *tank_create(...)
{
        struct tank *t = malloc(sizeof(*t));
        if (!t) {
                return NULL;
        }

        ...
        return t;
}

void tank_destroy(struct tank *t)
{
        free(t);
}
```

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

```
int main(void)
{
        struct tank *abrams = tank_create(...);
        struct tank *patton = tank_create(...);

        ...
        tank_destroy(patton);
        ...
        tank_destroy(abrams);
}
```

1.24 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`.

```
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.

WRONG: Leaks memory.

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

(continues on next page)

(continued from previous page)

```
        puts("Hello World!");

        //ERROR: Never freed
        double *values = malloc(10 * sizeof(*values));

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

Valgrind output of leaked-memory program.

```
$ valgrind ./hello
Hello World!
Goodbye World!
==25093==
==25093== HEAP SUMMARY:
==25093==     in use at exit: 80 bytes in 1 blocks
==25093==   total heap usage: 2 allocs, 1 frees, 1,104 bytes allocated
==25093==
==25093== LEAK SUMMARY:
==25093==    definitely lost: 80 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
Hello World!
Goodbye World!
==25240==
==25240== HEAP SUMMARY:
==25240==     in use at exit: 80 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 hello)
==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)
```

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
Hello World!
Goodbye World!
==25240==
==25240== HEAP SUMMARY:
==25240==     in use at exit: 80 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 (hello.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)
```

1.24.1 Memory Overrun Detection

Even further, Valgrind will detect when parts of the Heap that were *not* allocated are accessed.

WRONG: Leaks memory AND overruns array.

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

        //ERROR: Never freed
        double *values = malloc(10 * sizeof(*values));

        //ERROR: writes past end of array
        values[10] = 3.14;

        puts("Goodbye World!");
}

$ valgrind  --leak-check=full  ./hello
Hello World!
==27379== Invalid write of size 8
==27379==    at 0x10919F: main (hello.c:10)
==27379==  Address 0x4a544d0 is 0 bytes after a block of size 80 alloc'd
==27379==    at 0x483A7F3: malloc (in /usr/lib/...)
==27379==    by 0x10918A: main (hello.c:9)
==27379==
Goodbye World!
==27379==
==27379== HEAP SUMMARY:
==27379==     in use at exit: 80 bytes in 1 blocks
```

(continues on next page)

(continued from previous page)

```
==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 (hello.c:9)
==27379==
==27379== LEAK SUMMARY:
==27379==    definitely lost: 80 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 8`" 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.

1.24.2 Uninitialized Value Detection

One important warning that Valgrind may emit is "`uninitialised values`" (sic).

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

  double *values = malloc(10 * sizeof(*values));

  if (values[3] > 3.14) {
    puts("Bigger than Pi!");
  }

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

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
Hello World!
==28810== Conditional jump or move depends on uninitialised value(s)
==28810==    at 0x1091C3: main (hello.c:11)
==28810==
Goodbye World!
==28810==
==28810== HEAP SUMMARY:
```

(continues on next page)

(continued from previous page)

```
==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)
```

1.25 Exercises

:!exercise:

Remember to **free all dynamically allocated memory**.

1.25.1 Exercise 1

- Part 1: Write a program `safe_malloc.c` that contains a function `safe_malloc` with the following prototype:

```
void *safe_malloc(size_t num_bytes, char * var_name, int line_num, const char * func_
↪name);
```

The function attempts to allocate `num_bytes` of memory. If successful, the function returns a pointer to the allocated memory. If not, the program issues a diagnostic including the *values of the arguments passed to the function*.

Starter code for this exercise is in the file `starter_code/basic_dynamic_memory/safe_malloc.c`.

The output should resemble:

```
Safely allocated 12345678901 bytes stored at location 0x7efe6d430010 in function 'main
↪'
Fatal: failed to allocate 1234567890100 bytes for variable 'mem_block' on line 23 in␣
↪function 'alloc_a_bunch'.: Cannot allocate memory
Aborted (core dumped)
```

The above output shows one successful allocation and one failure.

- Part 2: Create a source code file `safe_malloc_func.c` that contains *only* the function with the definition shown in part 1. Delete the code for the function from `safe_malloc.c` containing the function. Use the `include` directive to use the code in `safe_malloc_func.c` in `safe_malloc.c`.

Build and run the program. The output should be identical to that shown in Part 1.

Use this source file for subsequent exercises.

1.25.2 Exercise 2

Write a program `basic_dyn_mem_01.c` that is a rewrite of `inter_pointers_02.c` done in the previous chapter's exercises.

The program fills instances of a struct with some data entered from the keyboard and some generated in the program.

The program `inter_pointers_02.c` from the previous chapter uses *static memory* to ensure the returned pointer does not *address local data*. The program `basic_dyn_mem_01.c` for this chapter will dynamically allocate the data used for the structures.

Use the `safe_malloc_func` written in Exercise 1.

Run the program against `valgrind`.

Sample output is provided below:

```
==31232== Memcheck, a memory error detector
==31232== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31232== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==31232== Command: ./bas1
==31232==
==31232== error calling PR_SET_PTRACER, vgdb might block
Enter employee name for employee #1 (no more than 20 characters) ==> emp01
Enter salary for 'emp01' ==> 100
Employee ID: 1  Name: emp01      Salary: 100.000000

Enter employee name for employee #2 (no more than 20 characters) ==> emp02
Enter salary for 'emp02' ==> 200
Employee ID: 2  Name: emp02      Salary: 200.000000

Enter employee name for employee #3 (no more than 20 characters) ==> emp03
Enter salary for 'emp03' ==> 300
Employee ID: 3  Name: emp03      Salary: 300.000000

==31232==
==31232== HEAP SUMMARY:
==31232==     in use at exit: 0 bytes in 0 blocks
==31232==   total heap usage: 5 allocs, 5 frees, 8,288 bytes allocated
==31232==
==31232== All heap blocks were freed -- no leaks are possible
==31232==
==31232== For counts of detected and suppressed errors, rerun with: -v
==31232== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The program is not correct until `valgrind` reports that **no leaks are found** and that **all heap blocks were freed**.

Starter code for this exercise is in the file `starter_code/basic_dynamic_memory/basic_dyn_mem_01.c`.

1.25.3 Exercise 3

Write a program `basic_dyn_mem_02.c` that accepts string input from the console and assigns the input strings to dynamically allocated elements of a string array.

Assume the largest allowable string input from the console is *20 characters*.

Continue accepting inputs until the user enters `q` or `Q`. There is no set number of strings to input.

Use the `safe_malloc_func` written in Exercise 1.

There is *no starter code for this exercise*.

When run with **valgrind**, the output should resemble:

```
==3109== Memcheck, a memory error detector
==3109== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3109== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3109== Command: ./bas2
==3109==
==3109== error calling PR_SET_PTRACER, vgdb might block
Enter a string (< 20 characters) or 'q', 'Q' to quit ==> string1
Enter a string (< 20 characters) or 'q', 'Q' to quit ==> string2
Enter a string (< 20 characters) or 'q', 'Q' to quit ==> string3
Enter a string (< 20 characters) or 'q', 'Q' to quit ==> q
String Element 0 is 'string1'
String Element 1 is 'string2'
String Element 2 is 'string3'
==3109==
==3109== HEAP SUMMARY:
==3109==     in use at exit: 0 bytes in 0 blocks
==3109==   total heap usage: 9 allocs, 9 frees, 8,332 bytes allocated
==3109==
==3109== All heap blocks were freed -- no leaks are possible
==3109==
==3109== For counts of detected and suppressed errors, rerun with: -v
==3109== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The program is not correct until `valgrind` reports that **no leaks are found** and that **all heap blocks were freed**.

1.25.4 Exercise 4

Write a program that contains a `main` function that:

- Allocates an array of struct instances. Fix the array size to `2` elements.
- Calls a function `load_struct` *within a loop* to populate an array of struct instances. Have the function *return a pointer to the newly created struct instance*.
- Calls a function `print_structs` from within a loop that prints the structs.

Use *pointers* to access and pass struct instances.

The `load_struct` function loads an instance of this structure:

```
struct address {
        int addr_num;
        char * street_name;
        char * city;
        char * state;
```

(continues on next page)

(continued from previous page)

```
        int zip;
};
```

Assume no input string may exceed *20 characters*.

Run the program with valgrind. The program is successful when there are no memory leaks.

The output should resemble:

```
==11505== Memcheck, a memory error detector
==11505== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11505== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==11505== Command: ./b4
==11505==
==11505== error calling PR_SET_PTRACER, vgdb might block
Enter street address for address #1 ==> 100
Enter street name for address #1 (no more than 20 characters) ==> street 1
Enter city for address #1 (no more than 20 characters) ==> city 1
Enter state for address #1 (no more than 2 characters) ==> tx
Enter zip for address #1 ==> 12345


Enter street address for address #2 ==> 200
Enter street name for address #2 (no more than 20 characters) ==> street 2
Enter city for address #2 (no more than 20 characters) ==> city 2
Enter state for address #2 (no more than 2 characters) ==> az
Enter zip for address #2 ==> 98765



Addresses entered

100 street 1
city 1, tx
12345

200 street 2
city 2, az
98765

==11505==
==11505== HEAP SUMMARY:
==11505==     in use at exit: 0 bytes in 0 blocks
==11505==   total heap usage: 10 allocs, 10 frees, 8,392 bytes allocated
==11505==
==11505== All heap blocks were freed -- no leaks are possible
==11505==
==11505== For counts of detected and suppressed errors, rerun with: -v
==11505== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.26 `memset` and `calloc`

```
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 01010101 bit pattern.

```
int main(void)
{
        size_t array_len = 1024;
        unsigned char *bytes = malloc(array_len);
        if (!bytes) {
                perror("Unable to allocate byte array");
                return 1;
        }
        memset(bytes, 0x55, array_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.

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

Use `calloc` to zero memory with NULL pointers.

```
int main(void)
{
        size_t array_len = 1024;
        double *ptrs = calloc(array_len, sizeof(*ptrs));
        if (!ptrs) {
                perror("Unable to allocate numeric array");
                return 1;
        }

        ...
        free(ptrs);
}
```

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.

1.27 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_capacity = 8;
        double *values = malloc(array_capacity * sizeof(*values));
```

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 two values, but has space for eight.

```
        size_t array_size = 0;
        values[0] = 3.14;
        ++array_size;
        values[1] = 2.78;
        ++array_size;
```

But, each of these variables (`values`, `array_capacity`, and `array_size`) 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 *values;
        size_t size;
        size_t capacity;
};
```

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 array_append(double *values, size_t size, size_t capacity, double new_value);
```

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

```
bool array_append(struct array *a, double new_value);
```

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.

1.28 `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.

```
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.

Preferred: Use of `realloc`.

```
struct array {
        double *values;
        size_t size;
        size_t capacity;
};

// Returns true if the value is appended, false otherwise
bool array_append(struct array *a, double new_value)
{
        if (!a) {
                return false;
        }

        if (a->size == a->capacity) {
                // Store result in temporary to test for success
                double *tmp = realloc(a->values, 2 * a->capacity);
                if (!tmp) {
                        return false;
                }

                // Overwrite old pointer, and update available capacity
                a->values = tmp;
                a->capacity *= 2;
        }

        a->values[a->size++] = value;
        return true;
}
```

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, while the other will double each time it needs more capacity.

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.

1.29 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 wordlist {
        char **words;
        size_t size;
```

(continues on next page)

(continued from previous page)

```
        size_t capacity;
};

struct wordlist *wordlist_create(void)
{
        struct wordlist *wl = malloc(sizeof(*wl));
        if (!wl) {
                return NULL;
        }

        // 8 seems like a reasonable starting capacity
        wl->capacity = 8;
        wl->size = 0;

        wl->words = malloc(wl->capacity * sizeof(*wl->words));
        if (!wl->words) {
                // Remember to free any partial allocations
                free(wl);
                return NULL;
        }

        return wl;
}
```

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
        size_t buf_size = 1 + strlen(s);
        char *new_string = malloc(buf_size);
        if (!new_string) {
                perror("Unable to duplicate string");
                return;
        }
        strncpy(new_string, s, buf_size);
}
```

1.29.1 `strdup`

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

```
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.

Store duplicates of strings in container.

```
bool wordlist_append(struct wordlist *wl, const char *s)
{
        if (!wl) {
                return false;
        }

        if (wl->size == wl->capacity) {
                char **tmp = realloc(wl->values, 2 * wl->capacity);
                if (!tmp) {
                        return false;
                }

                wl->values = tmp;
                wl->capacity *= 2;
        }

        wl->values[wl->size] = strdup(s);
        // Confirm that `strdup` succeeded
        if (!wl->values[wl->size]) {
                return false;
        }

        wl->size++;
        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 wordlist_print(const struct wordlist *wl)
{
        if (!wl) {
                return;
        }

        for (size_t n=0; n < wl->size; ++n) {
                printf("%s\n", wl->words[n]);
        }
}
```

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 wordlist_destroy(struct wordlist *wl)
{
        if (!wl) {
                return;
        }

        for (size_t n=0; n < wl->size; ++n) {
                free(wl->words[n]);
        }
```

(continues on next page)

(continued from previous page)

```
        free(wl->words);
        free(wl);
}
```

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)
{
        struct wordlist *wl = wordlist_create();
        ...
        wordlist_destroy(wl);

        //ERROR: wl's lifetime is over after the call to wordlist_destroy
        wordlist_print(wl);
}
```

1.30 Exercises

:!exercise:

Remember to **free all dynamically allocated memory**.

The student may use the `safe_malloc` function for all exercises except for exercise 2.

1.30.1 Exercise 1

Write a program `inter_dynamic_mem_01.c` that provides an implementation of a *stack of integers*.

Starter code for this exercise is in the file `starter_code/inter_dynamic_memory/inter_dynamic_mem_01.c`. The starter code has **most of the stack implementation already coded**, including the *pop*, *push*, and *peek* operations, as well as the *stack structure definition*.

The starter code lacks implementation of several key features:

- Code required to *create the stack*
- Code required to *reallocate the stack when full*
- Code required to *free dynamically allocated memory objects*.

The program pushes 20 numbers, *0 through 19*, onto the stack, prints the stack capacity and current number of elements, *pops three items*, and prints the stack capacity and current number of elements.

Do not change any code in the `main` function. This code has calls to create the stack, pop and push items, and prints stack information.

Initially, the `main` function calls a function (that you'll complete) that creates a stack capable of holding *two integers*. As the program adds (pushes) numbers onto the stack, the program *calls a function `reallocate_stack` when the stack is full*.

The exercise is to have the student malloc/realloc and free memory, not get into stack processing.

The output when run with `valgrind` is:

```
==9760== Memcheck, a memory error detector
==9760== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9760== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==9760== Command: ./t
==9760==
==9760== error calling PR_SET_PTRACER, vgdb might block
0 pushed to stack
1 pushed to stack
Stack filled - attempting to reallocate array to 4 elements
2 pushed to stack
3 pushed to stack
Stack filled - attempting to reallocate array to 8 elements
4 pushed to stack
5 pushed to stack
6 pushed to stack
7 pushed to stack
Stack filled - attempting to reallocate array to 16 elements
8 pushed to stack
9 pushed to stack
10 pushed to stack
11 pushed to stack
12 pushed to stack
13 pushed to stack
14 pushed to stack
15 pushed to stack
Stack filled - attempting to reallocate array to 32 elements
16 pushed to stack
17 pushed to stack
18 pushed to stack
19 pushed to stack

Stack has capacity: 32
Currently, stack has 20 elements

19 popped from stack
18 popped from stack

Stack has capacity: 32
Currently, stack has 18 elements

Freeing data used for the stack...
==9760==
==9760== HEAP SUMMARY:
==9760==     in use at exit: 0 bytes in 0 blocks
==9760==   total heap usage: 7 allocs, 7 frees, 4,360 bytes allocated
==9760==
==9760== All heap blocks were freed -- no leaks are possible
==9760==
==9760== For counts of detected and suppressed errors, rerun with: -v
==9760== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As an aside, *do not use the* `safe_malloc` *function* used in previous exercises.

1.30.2 Exercise 2

Write a program `inter_dynamic_mem_02.c` that:

- Allocates an array of strings of size `3`.
- Prompts user for string input or "quit" to quit accepting input
- *If there is room* in the array, save the string.
- *If there is no room*, **reallocate the array** to make room, then save the string.
- After user enters `quit`, print all the entered strings; one per line.

Run the program to *force reallocation* by entering more than three strings.

Run the program with valgrind; there should be no leaks or memory errors.

Include *checks for successful allocation* after every statement that *dynamically allocates memory*.

Sample output follows:

```
==18424== Memcheck, a memory error detector
==18424== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==18424== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18424== Command: ./old
==18424==
==18424== error calling PR_SET_PTRACER, vgdb might block
Enter input string or `q` ==> string 1
Enter input string or `q` ==> string 2
Enter input string or `q` ==> string 3
Enter input string or `q` ==> string 4
Size BEFORE calling realloc: 3   Realloc to 6
Enter input string or `q` ==> string 5
Enter input string or `q` ==> string 6
Enter input string or `q` ==> string 7
Size BEFORE calling realloc: 6   Realloc to 12
Enter input string or `q` ==> string 8
Enter input string or `q` ==> q
0       string 1

1       string 2

2       string 3

3       string 4

4       string 5

5       string 6

6       string 7

7       string 8

==18424==
==18424== HEAP SUMMARY:
==18424==     in use at exit: 0 bytes in 0 blocks
==18424==   total heap usage: 14 allocs, 14 frees, 8,540 bytes allocated
==18424==
==18424== All heap blocks were freed -- no leaks are possible
```

(continues on next page)

(continued from previous page)

```
==18424==
==18424== For counts of detected and suppressed errors, rerun with: -v
==18424== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.30.3 Exercise 3

Write a program `inter_dynamic_03.c` to prompt for the bounds of an integer array. The entered number **must be an even number**.

Initialize the elements from 0 to num_elems/2 - 1 (first half of the array) with the number `10` and the remaining elements (second half) with the number `20`.

Do not use loops to initialize the array. Use a loop to print the final array.

Print the elements after initialization.

The output should resemble:

```
==20331== Memcheck, a memory error detector
==20331== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20331== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==20331== Command: ./i3
==20331==
==20331== error calling PR_SET_PTRACER, vgdb might block
Enter number of array elements - must be an EVEN number ==> ert
Could not fetch num of elements from input 'ert'
Enter number of array elements - must be an EVEN number ==> 35
35 not EVEN. Try again
Enter number of array elements - must be an EVEN number ==> 20
Num entered = 20
First 10 elements:
10  10  10  10  10  10  10  10  10  10
Second 10 elements:
20  20  20  20  20  20  20  20  20  20
==20331==
==20331== HEAP SUMMARY:
==20331==     in use at exit: 0 bytes in 0 blocks
==20331==   total heap usage: 3 allocs, 3 frees, 8,212 bytes allocated
==20331==
==20331== All heap blocks were freed -- no leaks are possible
==20331==
==20331== For counts of detected and suppressed errors, rerun with: -v
==20331== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Run the program with valgrind; there should be no leaks or memory errors.

1.30.4 Exercise 4

Write a program `inter_dynamic_04.c` that copies the array created in exercise 2 above into a new array *in reverse order*.

The program should:

- Perform the steps in exercise 2 above to create the *source array* (copy the program).

Change the copy of the program from exercise 2 to:

- After fetching string inputs, copy the elements of the first array into the second array **using strdup**;such that, the first element of the second array **contains the data** in the last element of the first array.
- When done, print the elements of the both arrays.

Run the program to *force reallocation* by entering more than three strings.

Run the program with valgrind; there should be no leaks or memory errors.

Include *checks for successful allocation* after every statement that *dynamically allocates memory*.

The output should resemble:

```
==21664== Memcheck, a memory error detector
==21664== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==21664== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==21664== Command: ./i4
==21664==
==21664== error calling PR_SET_PTRACER, vgdb might block
Enter input string or `q` ==> s0
Enter input string or `q` ==> s1
Enter input string or `q` ==> s2
Enter input string or `q` ==> s3
Size BEFORE calling realloc: 3   Realloc to 6
Enter input string or `q` ==> s4
Enter input string or `q` ==> q
Input strings:
0       s0

1       s1

2       s2

3       s3

4       s4


Input strings in reverse order:
0       s4

1       s3

2       s2

3       s1

4       s0

==21664==
```

(continues on next page)

(continued from previous page)

```
==21664== HEAP SUMMARY:
==21664==     in use at exit: 0 bytes in 0 blocks
==21664==   total heap usage: 16 allocs, 16 frees, 8,444 bytes allocated
==21664==
==21664== All heap blocks were freed -- no leaks are possible
==21664==
==21664== For counts of detected and suppressed errors, rerun with: -v
==21664== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.31 Objectives

Gain a better understanding of previously covered concepts of the C programming language through the completion of practical exercises.

1.32 Exercises

:!exercise:

1.32.1 Exercise 1

Write a program, `lab01_ex1.c` that creates 5 instances of a `struct` and allows the user to print or change one of the fields in the struct's instance.

The struct and helpful comments are provided in the startup file `starter_code/lab_01/lab01_ex1.c`.

When run with `valgrind`, the output should resemble:

```
==3132== Memcheck, a memory error detector
==3132== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3132== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3132== Command: ./l1
==3132==
==3132== error calling PR_SET_PTRACER, vgdb might block
Employee info for Employee ID 0
Name: Emp0      Grade: 7        Dept ID: 1      Salary: 1674390.38

Employee info for Employee ID 1
Name: Emp1      Grade: 19       Dept ID: 8      Salary: 2422524.38

Employee info for Employee ID 2
Name: Emp2      Grade: 10       Dept ID: 12     Salary: 677445.75

Employee info for Employee ID 3
Name: Emp3      Grade: 13       Dept ID: 2      Salary: 2504426.62

Employee info for Employee ID 4
Name: Emp4      Grade: 18       Dept ID: 4      Salary: 2964000.38

Enter 'C'/'c' to change emp salary, 'P'/'p' to print employee data or 'Q'/'q' to quit␣
 ↪==> g
Invalid option g
```

(continues on next page)

(continued from previous page)

```
Enter 'C'/'c' to change emp salary, 'P'/'p' to print employee data or 'Q'/'q' to quit␣
 ↪==> P
Enter an employee number between 1 and 5 ==> 66
Invalid emp num entered: 66
Enter an employee number between 1 and 5 ==> 2
Employee info for Employee ID 1
Name: Emp1      Grade: 19       Dept ID: 8      Salary: 2422524.38

Enter 'C'/'c' to change emp salary, 'P'/'p' to print employee data or 'Q'/'q' to quit␣
 ↪==> c
Enter an employee number between 1 and 5 ==> 4

Employee Emp3 - old salary = 2504426.625000
Employee Emp3 - new salary = 2817479.953125

Enter 'C'/'c' to change emp salary, 'P'/'p' to print employee data or 'Q'/'q' to quit␣
 ↪==> q
==3132==
==3132== HEAP SUMMARY:
==3132==     in use at exit: 0 bytes in 0 blocks
==3132==   total heap usage: 7 allocs, 7 frees, 8,352 bytes allocated
==3132==
==3132== All heap blocks were freed -- no leaks are possible
==3132==
==3132== For counts of detected and suppressed errors, rerun with: -v
==3132== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.32.2 Exercise 2

Write a program, `lab02_ex2.c` that computes the min, max and average of numbers in an array of integers.

Write a function that accepts an array of integers as a parameter plus any other parameters deemed necessary.

The function must make the computed min, max and average *known to the calling function*.

Given the array initialization:

```
int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

The output should be:

```
Min = 1   Max = 10   Average = 5.500000
```

The above results are printed from the `main` function based on the computations done in a function.

1.32.3 Exercise 3

Write a program `lab01_ex3.c` that provides an implementation of a *stack of structures*.

Starter code for this exercise is in the file `starter_code/lab01/lab01_ex3.c`. The starter code has **most of the stack implementation already coded**, including the *pop*, *push*, and *peek* operations, as well as the *stack structure definition* and some code to assign values to the structure elements.

The starter code lacks implementation of several key features:

- Code required to *create the stack*
- Code required to *reallocate the stack when full*
- Code required to *free dynamically allocated memory objects*

The program pushes 5 struct instances onto the stack, prints the stack capacity and current number of elements, *pops three items*, and prints the stack capacity and current number of elements.

Do not change any code in the `main` function. This code has calls to create the stack, pop and push items, and prints stack information.

Run with *valgrind; program should produce no leaks or memory errors. The output should resemble:

```
==27067== Memcheck, a memory error detector
==27067== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27067== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==27067== Command: ./l3
==27067==
==27067== error calling PR_SET_PTRACER, vgdb might block
Person name: Person X           Age: 50
pushed to stack

Person name: Person XX          Age: 51
pushed to stack

Stack filled - attempting to reallocate array to 4 elements
Person name: Person XXX         Age: 52
pushed to stack

Person name: Person XXXX                Age: 53
pushed to stack

Stack filled - attempting to reallocate array to 8 elements
Person name: Person XXXXX               Age: 54
pushed to stack


Stack has capacity: 8
Currently, stack has 5 elements

Person name: Person X           Age: 50
Person name: Person XX          Age: 51
Person name: Person XXX         Age: 52
Person name: Person XXXX                Age: 53
Person name: Person XXXXX               Age: 54
Item # 0 popped from stack
Person name: Person XXXXX               Age: 54
Item # 1 popped from stack
Person name: Person XXXX                Age: 53
```

(continues on next page)

(continued from previous page)

```
Stack has capacity: 8
Currently, stack has 3 elements

Person name: Person X           Age: 50
Person name: Person XX          Age: 51
Person name: Person XXX         Age: 52
==27067==
==27067== HEAP SUMMARY:
==27067==     in use at exit: 0 bytes in 0 blocks
==27067==   total heap usage: 15 allocs, 15 frees, 4,516 bytes allocated
==27067==
==27067== All heap blocks were freed -- no leaks are possible
==27067==
==27067== For counts of detected and suppressed errors, rerun with: -v
==27067== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.32.4 Exercise 4

Write a program, `lab01_ex4.c` with a `main` function that:

- Calls a function that accepts an integer argument representing an array size and dynamically allocates *and returns* a string array to `main`.
- Calls another function that accepts the previously returned string array as an argument, *randomly generates strings containing only alphanumeric characters between 10 and 20 characters*, and loads the array.
- Calls one or more functions that determines the *longest, shortest, min and max string* and prints them.

Run with *valgrind; program should produce no leaks or memory errors.

The output should resemble:

```
==32760== Memcheck, a memory error detector
==32760== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==32760== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==32760== Command: ./l4
==32760==
==32760== error calling PR_SET_PTRACER, vgdb might block
String # 0      'WlrBbmQBhCDarzO'
String # 1      'KkYHIDdqSCDXrJmOWF'
String # 2      'xsjyBldbEFSArCBynE'
String # 3      'dyGgxxpkLoRellNM'
String # 4      'apqfWkHOPkMCoQHnWnk'
String # 5      'EwHsqmGbbuqCLJJiV'
String # 6      'wMdkqtBxIXMVTRRbl'
String # 7      'pTnsNFWZqfj'
String # 8      'AfAdrrWsofsBcnuV'
String # 9      'HFfbsAQxWpQCAc'
String # 10     'HcHzvFrkMLNozjKpq'
String # 11     'xRjxKITzYxAcb'
String # 12     'hKicQCoENdt'
String # 13     'MfGDWDwFc'
String # 14     'pXiqVKuYtDLcgDe'
String # 15     'HTacIOHOR'
String # 16     'tqKVwcsGSp'
String # 17     'oqmsBOAgUwnnyQX'
```

(continues on next page)

(continued from previous page)

```
String # 18     'zlGdGwPbtrwblNsaDeu'
String # 19     'UuMoqcDRUbeTo'

Shortest string: 'MfGDWDwFc'    Longest string: 'zlGdGwPbtrwblNsaDeu'

Min string: 'AfAdrrWsofsBcnuV'  Max string: 'zlGdGwPbtrwblNsaDeu'
==32760==
==32760== HEAP SUMMARY:
==32760==     in use at exit: 0 bytes in 0 blocks
==32760==   total heap usage: 42 allocs, 42 frees, 5,056 bytes allocated
==32760==
==32760== All heap blocks were freed -- no leaks are possible
==32760==
==32760== For counts of detected and suppressed errors, rerun with: -v
==32760== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.32.5 Exercise 5

Write a program, `lab02_ex5.c` that:

- Accepts a *single word* from the console or "quit" to quit the input process.
- Alerts the user if more than a single word was entered.
- Converts the input word to lower case.
- Check that the word *starts with an alpha character* and alert user if word entered is invalid.
- Copies the lower case version of the input word to an *array of words*.

After the user enters `quit`, the program:

- Print the words entered as shown in the output below:

If the user does not enter any words starting with a letter, output lists *None Entered* for that letter.

Print out the letter and the associated array elements:

```
Enter a single word or 'quit' to quit input ==> this is invalid
Could not parse 'this is invalid' into a single word
Enter a single word or 'quit' to quit input ==> #and
'#and' does not start with an alpha character
Enter a single word or 'quit' to quit input ==> this
Enter a single word or 'quit' to quit input ==> is
Enter a single word or 'quit' to quit input ==> valid
Enter a single word or 'quit' to quit input ==> and
Enter a single word or 'quit' to quit input ==> so
Enter a single word or 'quit' to quit input ==> is
Enter a single word or 'quit' to quit input ==> that
Enter a single word or 'quit' to quit input ==> And
Enter a single word or 'quit' to quit input ==> COnVeRT
Enter a single word or 'quit' to quit input ==> words
Enter a single word or 'quit' to quit input ==> whatever
Enter a single word or 'quit' to quit input ==> quit

'a' words: 'and' 'and'
'b' words: None Entered
'c' words: 'convert'
```

(continues on next page)

(continued from previous page)

```
'd' words: None Entered
'e' words: None Entered
'f' words: None Entered
'g' words: None Entered
'h' words: None Entered
'i' words: 'is' 'is'
'j' words: None Entered
'k' words: None Entered
'l' words: None Entered
'm' words: None Entered
'n' words: None Entered
'o' words: None Entered
'p' words: None Entered
'q' words: None Entered
'r' words: None Entered
's' words: 'so'
't' words: 'this' 'that'
'u' words: None Entered
'v' words: 'valid'
'w' words: 'words' 'whatever'
'x' words: None Entered
'y' words: None Entered
'z' words: None Entered
```

This chapter provides a review of the first five lessons in preparation for the upcoming written exam.

- Compilation units
- Basic pointer usage
- Intermediate pointer usage
- Basic dynamic memory usage
- Intermediate dynamic memory usage

1.33 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`.

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.c.

```
int main(void)
{
        printf("The result is %d\n", calculate());
}
```

library.c.

```
int calculate(void)
{
        return 8 * 6 + 7 / 5 - 3 + 0 * 9;
}
```

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

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

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* to 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 += -Wall -Wextra -Wpedantic

program: program.o library.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
cc -Wall -Wextra -Wpedantic   -c -o program.o program.c
program.c: In function ‘main’:
program.c:5:31: warning: implicit declaration of function ‘calculate’ [-Wimplicit-
 ↪function-declaration]
    5 |  printf("The result is %d\n", calculate());
      |                               ^~~~~~~~~
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`. 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
The result is 46
```

- Header Files

Solving the compiler warning (*implict declaration*) requires a different strategy. The declaration of the function `calculate` must be made explicit.

This is done by creating a *header file* for the given compilation unit.

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

```
#include <stdio.h>

#include "library.h"

int main(void)
{
        printf("The result is %d\n", calculate());
}
```

These delimiters tell the preprocessor where to look for headers. In C, these headers **do not** have any executable code, **only** declarations.

`library.h`: A single file that holds declarations for a compilation unit.

```
#ifndef LIBRARY_H
#define LIBRARY_H

int calculate(void);

#endif
```

A header file should hold declarations of all functions that the module makes available to its users. 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.

- `#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.

- `#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.

- External Linkage

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

`library.h`: Wrong; creates storage for `global_state`.

```
#ifndef LIBRARY_H
#define LIBRARY_H

char global_state;

int calculate(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.

Multiple storages created for a global variable.

```
cc   program.o library.o other.o  -o program
/usr/bin/ld: library.o:(.data+0x0): multiple definition of `global_state'; program.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.h`: Using `extern` does not create storage for `global_state`.

```
#ifndef LIBRARY_H
#define LIBRARY_H

extern char global_state;

int calculate(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.

- Internal Linkage

Just as symbols can be `extern` and link externally, accessible 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. A `static` variable is only initialized once. `static` variables inside functions retain their values between calls.

But, `static` also marks the linkage for a symbol as internal, and will not be handled by the linker.

`library.c`.

```
static int secret_function(void)
{
        return 8;
}

int calculate(void)
```

(continues on next page)

(continued from previous page)

```
{
        return secret_function() * 6 + 7 / 5 - 3 + 0 * 9;
}
```

1.34 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 `&`.

Get the address of a piece of data with `&`.

```
int main(void)
{
        int value = 13;
        printf("value = %d, &value = %p\n", value, &value);
}
```

On most systems, this will print something similar to:

```
value = 13, &value = 0x7ffc30f879d4
```

The `&` operator can only work on variables stored in memory. It will not work on literal values, return values, or ``register``s.

WRONG: Cannot take address of literal value.

```
int main(void)
{
        //ERROR: '17' is not a value in memory
        printf("%p\n", &17);
}
```

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

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 it.

- Pointer Declarations

Pointers are addresses. Addresses are 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 *px;
```

The human interpretation of this is "``px`` is a pointer to ``int``".

Once created, a pointer variable may hold an address.

```
int main(void)
{
        int value = 13;
        int *px;
        px = &value;

        printf("%d\n", *px); // Prints "13"
}
```

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

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

Pointers may store the address of any instance of their type.

```
int main(void)
{
        double pi=3.14, e=2.72, ln10=2.30;

        double *current;

        current = &pi;
        printf("%lf\n", *current); // Prints "3.14"
        current = &ln10;
        printf("%lf\n", *current); // Prints "2.30"
        current = &e;
        printf("%lf\n", *current); // Prints "2.72"
}
```

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

Pointers may be initialized.

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

(continues on next page)

(continued from previous page)

```
        int value = 19;
        int *px = &value;

        printf("%d\n", *px); // Prints "19"
}
```

Pointer variables also allow modification of the data being pointed at.

Pointers may be used to modify what they point at.

```
int main(void)
{
        int value = 19;
        int *px = &value;

        printf("%d\n", value); // Prints "19"
        *px = 23;
        printf("%d\n", value); // Prints "23"
}
```

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

Multiple pointers may point to the same data.

```
int main(void)
{
        int value = 29;
        int *px = &value;
        int *py = &value;

        printf("%d : %d\n", *px, value); // Prints "29 : 29"
        *py = 31;
        printf("%d : %d\n", value, *px); // Prints "31 : 31"
}
```

Since pointers are also values (the address), they can also be compared. Ordering pointers is not very useful, but comparing for equality is. Use the ==/!= operators to compare pointers for equality/inequality.

- 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.

null.c.

```
```

- 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.

Pointer arguments can manipulate data outside the function's scope.

```
void square(int *num)
{
        if (!num) {
                return;
        }

        // This changes the data pointed at by num (squares the number, in this case)
        *num *= *num;
}

int main(void)
{
        int favorite_number = 37;

        square(&favorite_number);
        printf("%d\n", favorite_number); // Prints "1369"
}
```

- 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.

```
// Calculates the positive and negative root of input
// pos and neg are output parameters
void two_roots(double input, double *pos, double *neg)
{
        double root = sqrt(input);
        if (pos) {
                *pos = root;
        }
        if (neg) {
                *neg = -root;
        }
}
```

- `struct` Pointers

When it comes to ``struct``s, pointers have a lot of value.

```
struct mrap {
        int mass_kg;
        int width_cm;
        int height_cm;
        int length_cm;
        int max_speed_kph;
        int id_number;
};

struct tank {
        int mass_kg;
        int width_cm;
        int height_cm;
        int length_hull_cm;
        int length_fwd_gun_cm;
        int max_speed_kph;
        int id_number;
```

(continues on next page)

(continued from previous page)

```
};
```

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.

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

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

```
void tank_print(struct tank *t);

int main(void)
{
        struct tank abrams = { ... };
        // The address of 'abrams' is copied to 't': t = &abrams;
        tank_print(&abrams);
}
```

- \-> Operator

Working with ``struct``s presents a problem: member access.

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 tank_print(struct tank *t)
{
        printf("%dcm\n", t->height_cm);
}
```

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 tank_commander {
        char *name;
        struct tank *tank;
};

void tank_commander_print(struct tank_commander *tc)
{
        printf("%dcm\n", tc->tank->height_cm);
}
```

1.35 Intermediate Pointers

- Pointer Return Values

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

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 of some data. 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!

WRONG: Returning a pointer to a function-local object.

```
struct tank *tank_create(...)
{
        struct tank t = {...};

        //ERROR: t's lifetime is over when the function returns.
        return &t;
}

int main(void)
{
        struct tank *abrams = tank_create(...);
}
```

Storage for data returned from a function needs to last longer than the function call, **guaranteed**. There are three ways to do this:

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

Pointer lower On the Call Stack Than the Caller:

1. Only a few instances are needed; AND
2. The object does not need to live longer than the current 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 tank_print(struct tank *t);

int main(void)
{
        // 'abrams' is on the stack frame for 'main'
        struct tank abrams = {...};
        tank_print(&abrams);
}

// 't' is a pointer to data inside the 'main' function
void tank_print(struct tank *t)
{
        printf(...);
```

(continues on next page)

(continued from previous page)

```
        // At the end of the function, t is reclaimed, but not its contents
}
```

There are numerous times when this model is used. However, 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

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

Recall that the Data section of the Most Important Picture, which holds `static` data, is fixed in size.

Since static data is alive the entire program, a pointer to `static` data will continue to be valid.

Returning a pointer to the Data section.

```
struct db_connection *db_connect(...)
{
        static db_connection dbh = {...};
        ...
        return &dbh;
}
```

The downside of this approach is that there can only ever be one instance of the data.

Heap Storage Duration

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. 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 *a);

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

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.

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). Use the `sizeof` compile-time operator:

Arrays decay to pointers when passed as arguments.

```
void array_print_size(int a[5])
{
      // sizeof returns the size of the pointer, not the array
        printf("Size in array_print_size: %zu\n", sizeof(a)); // Prints "8"
}

int main(void)
{
        int fibonacci[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};

        array_print_size(fibonacci); // Prints "8", sizeof(&fibonacci)
        printf("Size in main: %zu\n", sizeof(fibonacci)); // Prints "36"
}
```

- Pointer Arithmetic

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.

```
void array_print_first_two(int *a)
{
        printf("First element: %d\n", a[0]);
        printf("Second element: %d\n", a[1]);

        printf("a[0] == *a → %d", a[0] == *a); // Prints "1"
}

int main(void)
{
        int fibonacci[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
        array_print_first_two(fibonacci);
}
```

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])`.

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.

Sample output from `array_print(fibonacci, 5)`.

```
a + 0 → 0x7ffffffc00
a + 1 → 0x7ffffffc04
a + 2 → 0x7ffffffc08
```

(continues on next page)

(continued from previous page)

```
a + 3 → 0x7fffffffc0c
a + 4 → 0x7fffffffc10
```

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

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

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 interleave_string(const char *s, char sep)
{
        if (!s) {
                return;
        }

        // While `*s` is not the NUL byte
        while (*s) {
                printf("%c%c", *s, sep);

                // Move `s` forward one step in the character array
                ++s;
        }
}
```

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.

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 tank``s. Due to array decay, this type becomes `struct tank **`.

```
void tank_foreach_print(struct tank **tanks, size_t sz);

int main(void)
{
        struct tank abrams = { ... };
        struct tank patton = { ... };
        struct tank *platoon[] = { &abrams, &patton, ... };

        tank_foreach_print(platoon, sizeof(platoon) / sizeof(platoon[0]));
}
```

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.

- `void *`

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 = "US Army";

        void *value = name;

        char *other_name = value;
}
```

The `NULL` pointer is actually a `void *`. The `NULL` pointer may be assigned to a pointer of *any type*.

- 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; quite the opposite, actually.

A common use of casting pointers is to add or remove `const` qualifiers.

- Function Pointers

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
        printf("a == &a → %d", main == &main); // Prints '1'
}
```

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

Declaring a pointer to a function.

```
int *actual_function(int, int);
int (*pointer_to_function)(int, 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.

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.

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.

```
// Takes a function as a parameter, and prints each element in the array
void array_print(void **a, size_t sz, void (*print_func)(void *))
{
        for (size_t n=0; n < sz; ++n) {
                print_func(a[n]);
        }
}

void mrap_print(void *arg)
{
        struct mrap *m = arg;
        ...
}

void tank_print(void *arg)
{
        struct tank *t = arg;
        ...
}

int main(void)
{
        // Prints the 'platoon' array using 'tank_print'
        array_print(platoon, platoon_sz, tank_print);

        // Prints the 'column' array using 'mrap_print'
        array_print(column, column_sz, mrap_print);
}
```

This is much simpler than having to write a custom `tank_array_print` function and a `mrap_array_print` function, plus whatever other types are needed.

Note that both `mrap_print` and `tank_print` had signatures that took in `void *` rather than `struct tank *`. 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.

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

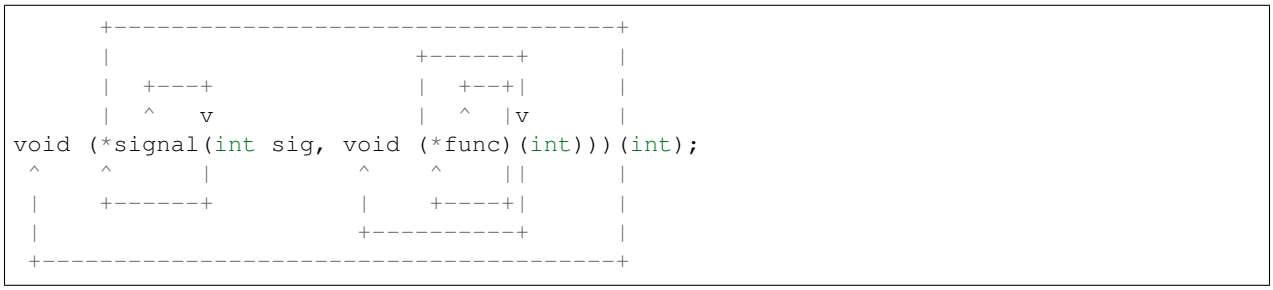
1. Begin with the unknown identifier.
2. Move in a clockwise spiral from the identifier, reading off each piece of syntax.
3. Resolve the entirety of a parenthetical before moving outside of it.

Parsing `int *alfa[3].`

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

1. `alfa` is…
2. an array of 3…
3. pointers to…
4. `int`

Parsing `void (*signal(int sig, void (*func)(int)))(int).`



1. `signal` is…
2. a function that takes…
    1. `sig`, which is…
    2. an `int`…
    3. and `func`, which is…
    4. a pointer to…
    5. a function that takes…
        1. an `int`
        2. returning `void`
3. returning a pointer to…
4. a function that takes…
    1. an `int`
    2. returning `void`

1.36 Basic Dynamic Memory

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

But, they are 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`

```
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 enough storage for a filename, plus a terminating NUL
        char *filename = malloc(NAME_MAX + 1);
        // malloc() returns NULL on failure
        if (!filename) {
                perror("Could not allocate enough memory");
                return 1;
        }

        // Get filename, do stuff.....

        // Returns the memory to the Heap to be reused
        free(filename);
}
```

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

When the program calls `malloc()`, the function checks for and returns an available piece of Heap memory.

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.

```
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*.

When such a function issuing `malloc` calls without `free` calls 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`. There is a zero-tolerance policy throughout this course against memory leaks.

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.

WRONG: Allocation size and type do not agree.

```
struct tank *tank_create(...)
{
        //ERROR: sizeof(struct tank) != sizeof(struct mrap)
        struct tank *t = malloc(sizeof(struct mrap));

        ...
        return t;
}
```

To avoid this, the size expression passed to `malloc` should *always* use `sizeof(\*_object_)`. Be careful to remember the `*`, otherwise just a pointer's worth of bytes are allocated!

Preferred: Allocation size based on destination.

```
struct tank *tank_create(...)
{
        struct tank *t = malloc(sizeof(*t));

        ...
        return t;
}
```

Do not cast the pointer returned from `malloc` to the type specified in the RHS of the `malloc` expression.

- 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.

The memory accessible by the pointer returned by `malloc` does not (and should not) be initialized if it 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`.

- 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.

```
int main(void)
{
        size_t array_size = 10;

        double *pies = malloc(array_size * sizeof(*pies));
        if (!pies) {
                perror("Unable to create list of pies");
                return 1;
        }

        for (size_t = n; n < array_size; ++n) {
                pies[n] = 3.141592653;
        }

        ...
}
```

As with all arrays, the number of elements in the array should accompany the array wherever it goes. C does not have bounded arrays.

- 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*.

The programmer must ensure that references to dynamically allocated memory (`malloc`) is accessible **where the memory needs to be reclaimed** (`free`).

- Valgrind

Valgrind is a dynamic analysis tool that is useful to developers. It is very easy to forget 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`.

```
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.

WRONG: Leaks memory.

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

        //ERROR: Never freed
        double *values = malloc(10 * sizeof(*values));

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

Valgrind output of leaked-memory program.

```
$ valgrind ./hello
Hello World!
Goodbye World!
==25093==
==25093== HEAP SUMMARY:
==25093==     in use at exit: 80 bytes in 1 blocks
==25093==   total heap usage: 2 allocs, 1 frees, 1,104 bytes allocated
```

(continues on next page)

(continued from previous page)

```
==25093==
==25093== LEAK SUMMARY:
==25093==    definitely lost: 80 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
Hello World!
Goodbye World!
==25240==
==25240== HEAP SUMMARY:
==25240==     in use at exit: 80 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 hello)
==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)
```

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
Hello World!
Goodbye World!
==25240==
==25240== HEAP SUMMARY:
==25240==     in use at exit: 80 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 (hello.c:9)
==25240==
==25240== LEAK SUMMARY:
```

(continues on next page)

(continued from previous page)

```
==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)
```

- `valgrind` and memory Overrun Detection

Even further, Valgrind will detect when parts of the Heap that were *not* allocated are accessed.

WRONG: Leaks memory AND overruns array.

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

        //ERROR: Never freed
        double *values = malloc(10 * sizeof(*values));

        //ERROR: writes past end of array
        values[10] = 3.14;

        puts("Goodbye World!");
}

$ valgrind  --leak-check=full  ./hello
Hello World!
==27379== Invalid write of size 8
==27379==    at 0x10919F: main (hello.c:10)
==27379==  Address 0x4a544d0 is 0 bytes after a block of size 80 alloc'd
==27379==    at 0x483A7F3: malloc (in /usr/lib/...)
==27379==    by 0x10918A: main (hello.c:9)
==27379==
Goodbye World!
==27379==
==27379== HEAP SUMMARY:
==27379==     in use at exit: 80 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 (hello.c:9)
==27379==
==27379== LEAK SUMMARY:
==27379==    definitely lost: 80 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 8`" indicates that unallocated memory in the Heap was written to, a violation of memory safety. Valgrind shows the line number where the write happened.

Fixing errors and warnings from Valgrind should be done rigorously, in the same way that fixing compiler errors and warnings is done. 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.

- `valgrind` and uninitialized Value Detection

One important warning that Valgrind may emit is "`uninitialised values`" (sic).

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

  double *values = malloc(10 * sizeof(*values));

  if (values[3] > 3.14) {
    puts("Bigger than Pi!");
  }

  puts("Goodbye World!");
  free(values);
```

This kind of warning is emitted when the uninitialized 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
Hello World!
==28810== Conditional jump or move depends on uninitialised value(s)
==28810==    at 0x1091C3: main (hello.c:11)
==28810==
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)
```

1.37 Intermediate Dynamic Memory

- `memset` and `calloc`

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

Use `memset` to initialize a block of allocated memory to some initial value.

Use `memset` to fill memory with 01010101 bit pattern.

```
int main(void)
{
        size_t array_len = 1024;
        unsigned char *bytes = malloc(array_len);
```

(continues on next page)

(continued from previous page)

```
        if (!bytes) {
                perror("Unable to allocate byte array");
                return 1;
        }
        memset(bytes, 0x55, array_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` allocates memory like `malloc` and initializes the memory to `0` (0, `'\0'`, or `NULL` depending on the data type of the target pointer) Also like `malloc`, any such allocated memory must be ``free``d by the end of the program.

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

Only use `calloc` when a block of memory is needed AND the memory needs to be all zeroes.

- A Truly Dynamic Array example

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

```
        size_t array_capacity = 8;
        double *values = malloc(array_capacity * sizeof(*values));
```

It is important to track which elements of this array are valid. Note that this is different from the total capacity available: the array might only be used for two values, but has space for eight.

```
        size_t array_size = 0;
        values[0] = 3.14;
        ++array_size;
        values[1] = 2.78;
        ++array_size;
```

But, each of these variables (`values`, `array_capacity`, and `array_size`) 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 *values;
        size_t size;
        size_t capacity;
};
```

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

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.

```
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.

Preferred: Use of `realloc`.

```
struct array {
        double *values;
        size_t size;
        size_t capacity;
};

// Returns true if the value is appended, false otherwise
bool array_append(struct array *a, double new_value)
{
        if (!a) {
                return false;
        }

        if (a->size == a->capacity) {
                // Store result in temporary to test for success
                double *tmp = realloc(a->values, 2 * a->capacity);
                if (!tmp) {
                        return false;
                }

                // Overwrite old pointer, and update available capacity
                a->values = tmp;
                a->capacity *= 2;
        }

        a->values[a->size++] = value;
        return true;
}
```

It is *generally* a good idea to grow memory by doubling the current amount for a given allocation to limit the number of `realloc` calls needed as the program executes.

- Arrays of pointers

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 wordlist {
        char **words;
        size_t size;
        size_t capacity;
};

struct wordlist *wordlist_create(void)
{
        struct wordlist *wl = malloc(sizeof(*wl));
        if (!wl) {
```

(continues on next page)

(continued from previous page)

```
                return NULL;
        }

        // 8 seems like a reasonable starting capacity
        wl->capacity = 8;
        wl->size = 0;

        wl->words = malloc(wl->capacity * sizeof(*wl->words));
        if (!wl->words) {
                // Remember to free any partial allocations
                free(wl);
                return NULL;
        }

        return wl;
}
```

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.

- `strdup`

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

```
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.

Store duplicates of strings in container.

```
bool wordlist_append(struct wordlist *wl, const char *s)
{
        if (!wl) {
                return false;
        }

        if (wl->size == wl->capacity) {
                char **tmp = realloc(wl->values, 2 * wl->capacity);
                if (!tmp) {
                        return false;
                }

                wl->values = tmp;
                wl->capacity *= 2;
        }

        wl->values[wl->size] = strdup(s);
        // Confirm that `strdup` succeeded
        if (!wl->values[wl->size]) {
```

(continues on next page)

(continued from previous page)

```
                return false;
        }

        wl->size++;
        return true;
}
```

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 wordlist_destroy(struct wordlist *wl)
{
        if (!wl) {
                return;
        }

        for (size_t n=0; n < wl->size; ++n) {
                free(wl->words[n]);
        }

        free(wl->words);
        free(wl);
}
```

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

Todo: Module I - Validate files and correct image references.

CHAPTER

TWO

MODULE J - C PROGRAMMING II - BLOCK 2

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 mean 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.

2.1 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.

```
programname: programname.o atl.o yyg.o

...

testprogname: testprogname.o atl.o yyg.o

# programname is freshly built when tests are run
check: programname testname
        ./testname
```

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.

```
check: test/test_all
        ./$^

# All unit test code
test/test_all: test/test_all.o test/test_atl.o test/test_yyg.o
```

(continues on next page)

(continued from previous page)

```
# Modules under test
test/test_all: atl.o yyg.o
# Extra Library for Check
test/test_all: LDLIBS += -lcheck
```

2.2 Check Organization

With Check, there are five key pieces to writing tests:

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).

2.2.1 Suite Runner

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

`test_all.c`.

```
extern Suite *test_yyg_suite(void);
extern Suite *test_atl_suite(void);

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

        srunner_add_suite(sr, test_yyg_suite());
        srunner_add_suite(sr, test_atl_suite());
        srunner_add_suite(sr, ...);
        ...

        srunner_run_all(sr, CK_NORMAL);
        int failed = srunner_ntests_failed(sr);
        srunner_free(sr);

        return !!failed;
}
```

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.

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!

2.2.2 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_yyg.c`.

```
Suite *test_yyg_suite(void)
{
        Suite *s = suite_create("YYG");
        TFun *curr = NULL;

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

        TCase *tc_input = tcase_create("input");
        ...
        suite_add_tcase(s, tc_input);

        ...

        return s;
}
```

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`.

2.2.3 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 core_tests[] = {
        test_yyg_create,
        test_yyg_put_down,
        test_yyg_flip,
        test_yyg_reverse,
        NULL
};

Suite *test_yyg_suite(void)
{
        Suite *s = suite_create("YYG");
        TFun *curr = NULL;

        curr = core_tests;
        while (*curr) {
                tcase_add_test(tc_core, *curr++);
        }
        suite_add_tcase(s, tc_core);

        ...

        return s;
}
```

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_atl.c`.

```
static airport *atl;

static void in_setup(void)
{
        atl = airport_create("ATL");
}

static void in_teardown(void)
{
        airport_destroy(atl);
}

Suite *test_atl_suite(void)
{
```

(continues on next page)

(continued from previous page)

```
        Suite *s = suite_create("ATL");
        TFun *curr = NULL;

        TCase *tc_input = tcase_create("input");
        tcase_add_checked_fixture(tc_input, in_setup, in_teardown);
        curr = input_tests;
        while (*curr) {
                tcase_add_test(tc_input, *curr++);
        }
        suite_add_tcase(s, tc_input);

        return s;
}
```

2.2.4 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)
{
        ck_assert_str_eq(atl->name, "Atlanta");
}
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 square_data[][2] = {
        {0, 0},
        {1, 1},
        {2, 4},
        {-2, 4},
        {9, 81}
};
enum { SQUARE_DATA_SZ = sizeof(square_data)/sizeof(square_data[0]) };

START_TEST(test_square)
{
        ck_assert_int_eq(square(square_data[_i][0]), square_data[_i][1]);
}
END_TEST

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

(continues on next page)

(continued from previous page)

```
        tcase_add_loop_test(tc_core, test_square, 0, SQUARE_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 complex_function_test {
        int a;
        int b;
        const char *s;
        double expected;
        double tolerance;
};

struct complex_function_test cft_data[] {
        { 3, 4, "5", 0, 0.001 },
        { 18, 76, "45", 3.44e10, 10.0 },
        ...
};

START_TEST(complex_test)
{
        struct complex_function_test *item = &cft_data[_i];

        ck_assert_double_eq_tol(
                        complex_function(item->a, item->b, item->s),
                        item->expected,
                        item->tolerance);
}
END_TEST
```

2.2.5 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.

2.3 `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_airport(airport *a)
{
        //DANGER: This line may be omitted by the preprocessor
        assert(a != NULL);

        //ERROR: The decrement will not happen in a release build
        assert(a->refs-- != 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.

2.4 Exercises

:!exercise:

The exercises for this lesson involve writing a *Suite Runner* that runs a *Test Suite* referencing one or more *Test Cases* that reference several *Unit Tests* containing *one assertion*.

The Unit Tests should test if each method produces a correct value and reports on incorrect values.

The starter code folder contains a folder for each lab that contains:

- A `src` folder
  - The `src` folder contains the functions to be tested; there is no `main` function in the file in the `src` folder.
- A `tests` folder
  - The `tests` folder is where the student will code the *Test Suite* referencing one or more *Test Cases* that reference several *Unit Tests* containing *one assertion*.
- A `makefile`
  - This makefile should not be changed.

The starter code folders atr `check01`, `check02`, `check03`, and `check04` corresponding to the four exercises included in this lesson.

The *exercise descriptions* show the expected output from successful coding of the test components.

2.4.1 Exercise 1

There are *three* functions to be tested.

The expected output is:

```
$ make all
gcc -c -Wall  src/*.c
gcc -c -Wall   tests/*.c
gcc check01.o check_check01.o -lcheck -lm -lpthread -lrt -lsubunit  -o check_check01_
 ↪tests

$ make test
./check_check01_tests
Running suite(s): arith_ops
50%: Checks: 6, Failures: 3, Errors: 0
tests/check_check01.c:12:F:add_fail:add_fail:0: Assertion 'add_two_numbers(16, 16) ==␣
 ↪0' failed: add_two_numbers(16, 16) == 32, 0 == 0
tests/check_check01.c:21:F:sub_fail:sub_fail:0: Assertion 'subtract_two_numbers(2, 2)␣
 ↪== 10' failed: subtract_two_numbers(2, 2) == 0, 10 == 10
tests/check_check01.c:29:F:mul_fail:mul_fail:0: Assertion 'multiply_two_numbers(1, 1)␣
 ↪== 10' failed: multiply_two_numbers(1, 1) == 1, 10 == 10
Makefile:21: recipe for target 'test' failed
make: *** [test] Error 1
```

Note there are six test cases; three passed, three failed.

2.4.2 Exercise 2

There are *three* functions to be tested.

For this lab, create an *array of test functions* and add the test in a loop.

The expected output is:

```
$ make all
gcc -c -Wall   tests/*.c
gcc check02.o check_check02.o -lcheck -lm -lpthread -lrt -lsubunit  -o check_check02_
 ↪tests

$ make test
./check_check02_tests
Running suite(s): str_ops
50%: Checks: 6, Failures: 3, Errors: 0
tests/check_check02.c:21:F:str_ops:core_tests[f_i]:0: Assertion 'upper == "abCDe"'␣
 ↪failed: upper == "ABCDEFGH", "abCDe" == "abCDe"
tests/check_check02.c:33:F:str_ops:core_tests[f_i]:0: Assertion 'the_smallest_
 ↪char(string) == 'a'' failed: the_smallest_char(string) == 70, 'a' == 97
tests/check_check02.c:45:F:str_ops:core_tests[f_i]:0: Assertion 'string == "abcde"'␣
 ↪failed: string == "hGFedcba", "abcde" == "abcde"
Makefile:21: recipe for target 'test' failed
make: *** [test] Error 1
```

2.4.3 Exercise 3

There is *one* function to be tested.

When the following triples of data are tested:

```
1, 2, 3
3, 4, 5
6, 8, 10
13, 14, 15
```

The expected output is:

```
$ make all
gcc -c -Wall   tests/*.c
gcc check03.o check_check03.o -lcheck -lm -lpthread -lrt -lsubunit  -o check_check03_
 ↪tests

$ make test
./check_check03_tests
Running suite(s): check03_ops
50%: Checks: 4, Failures: 2, Errors: 0
tests/check_check03.c:17:F:right_triangle:triangle_test:0: Assertion 'right_
 ↪triangle(triangle_sides[_i]) == 1' failed: right_triangle(triangle_sides[_i]) == 0,␣
 ↪1 == 1
tests/check_check03.c:17:F:right_triangle:triangle_test:3: Assertion 'right_
 ↪triangle(triangle_sides[_i]) == 1' failed: right_triangle(triangle_sides[_i]) == 0,␣
 ↪1 == 1
Makefile:21: recipe for target 'test' failed
make: *** [test] Error 1
```

Code the triplets in an appropriate structure and use a *looping construct* to add the test case function.

2.4.4 Exercise 4

Copy the work done in the previous exercise and change the test program to use a *structure* that holds the data with the expected result. With the expected result from the function coded in the structure, the test cases *should all pass*.

The output using the same triples of data in exercise 3 above should yield the following output:

```
./check_check04_tests
Running suite(s): check04_ops
100%: Checks: 4, Failures: 0, Errors: 0
```

With an understanding of pointers and unit tests, more attention can now be given to designing programs in C. 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.

2.5 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 `centimeters`.

```
typedef  unsigned int  centimeters;
```

In the above example, there is now a new type called `centimeters`. New variables can now be created with this type.

```
centimeters tub_length = 15;
```

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  centimeters;
typedef  unsigned int  liters;

{
        centimeters tub_length = 15;
        liters tub_volume = 80;

        //Misleading: these different units combine with no compiler warning
        unsigned int misleading = tub_volume * tub_length;
}
```

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 `centimeters` and `liters` should not be multiplied 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  char **  word_array;
```

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

2.6 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 valuelist *values = valuelist_create();

        //BAD; valuelist functions would start malfunctioning
        values->size = 43;
```

(continues on next page)

(continued from previous page)

```
        ...
}
```

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 dynamic array of words. These fields are tightly coupled and generally should *only* be manipulated by code that understands their relationship.

```
struct wordlist_ {
        char **words;
        size_t size;
        size_t capacity;
};
```

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

```
typedef struct wordlist_ wordlist;
```

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`.

`wordlist.h`.

```
#ifndef WORDLIST_H
#define WORDLIST_H

#include <stdbool.h>


typedef struct wordlist_ wordlist;


wordlist *wordlist_create(void);

bool wordlist_append(wordlist *wl, const char *s);

void wordlist_print(const wordlist *wl);

void wordlist_destroy(wordlist *wl);

#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)
{
        wordlist *words = wordlist_create();
        if (!words) {
```

(continues on next page)

(continued from previous page)

```
                perror("Could not create word list");
                return 1;
        }

        // Unpreferred; the return value should be checked
        wordlist_append(words, "Able");
        wordlist_append(words, "Baker");
        wordlist_append(words, "Charlie");

        wordlist_print(words);

        wordlist_destroy(words);
}
```

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

wordlist.c.

```
#include "wordlist.h"

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

struct wordlist_ {
        char **words;
        size_t size;
        size_t capacity;
};

wordlist *wordlist_create(void)
{
        wordlist *wl = malloc(sizeof(*wl));
        if (!wl) {
                return NULL;
        }
        wl->size = 0;
        ...
}

bool wordlist_append(wordlist *wl, const char *s)
{
        ...
}

void wordlist_print(const wordlist *wl)
{
        ...
}

void wordlist_destroy(wordlist *wl)
{
        ...
}
```

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.).

2.7 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.

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 BALLYNAHINC_H

#define BALLYNAHINC_H

...

#endif
```

2.7.1 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.

2.7.2 ``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.

2.7.3 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 { WL_SUCCESS, WL_OUT_OF_MEMORY, WL_NO_PERMISSION };

enum { WL_DEFAULT_CAPACITY=8 };
```

2.7.4 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.

2.7.5 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 repared 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`.

2.7.6 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.

2.7.7 `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.

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

```
// This variable is only visible inside the compilation unit
static size_t number_of_extant_wordlists;

// This function would only be used internally to the compilation unit
static bool check_valid(wordlist *wl)
{
        return wl && wl->words;
}
```

2.8 Exercises

:!exercise:

2.8.1 Exercise 1

Build out the `wordlist` type as a separate module. It should support the following methods.

```
wordlist *wordlist_create(void);

// Returns the number of valid entries in the wordlist
size_t wordlist_size(const wordlist *wl);

// Fetches the word at the specified index
// Returns NULL on failure
const char *wordlist_get(wordlist *wl, size_t idx);

// Adds a word to the end of the wordlist
// Returns false on failure
bool wordlist_append(wordlist *wl, const char *s);

// Adds a word to the front of the wordlist
// Returns false on failure
```

(continues on next page)

(continued from previous page)

```
bool wordlist_prepend(wordlist *wl, const char *s);

// Print the list
void wordlist_print(wordlist *wl);

void wordlist_destroy(wordlist *wl);
```

The above code is in the starter code file `starter_code/design/wordlist_methods.txt`.

Use the techniques described in this chapter (opaque types, etc.) to build the module.

The solution will include an *interface file*, an *implementation file*, and another program that uses the module.

Stub the implementation of the above methods in the implementation file. Each method should print a string describing what it does when called. The output shown below shows example strings the stubbed methods may print. The student will *complete the implementation* in a later exercise. Here, the emphasis is on creating files that allow the wordlist to be used *as an opaque type*.

The `main` function is provided in the starter file `starter_code/design/use_wordlist.c`. The student will need to *copy this file* and complete the file by including the needed header files.

When executed, the output should be:

```
'wordlist_create' called - Wordlist created with size = 0 and capacity = 8
'wordlist_append' called - Word Word1 appended
'wordlist_append' called - Word word2 appended
'wordlist_append' called - Word word3 appended
'wordlist_print' called - Wordlist printed
'wordlist_prepend' called - Word Word100 prepended
'wordlist_get' called - Word at 1 returned
The second word in wordlist is : WordFromList
'wordlist_print' called - Wordlist printed
'wordlist_destroy' called - Wordlist destroyed
```

The file `use_wordlist` in the starter code contains the following:

```
        // Attempting to access fields within opaque type wordlist
        // should cause a compiler error:
        // printf("Capacity = %d\n", a_wl -> capacity);
```

Uncomment out the `printf` line and build the program. The compiler should complain as follows:

```
use_wordlist.c: In function ‘main’:
use_wordlist.c:11:33: error: dereferencing pointer to incomplete type ‘wordlist {aka␣
 ↪struct wordlist_}’
  printf("Capacity = %d\n", a_wl -> capacity);
```

This shows the wordlist type *is opaque and encapsulates its fields*.

2.8.2 Exercise 2

Examine the `stack` data structure in the file `stack.c` found in the starter code folder `starter_code/design`.

Build out the `stack` type as a separate module. It should support the following methods.

```
stack* create_stack(int capacity)

int isFull(stack* a_stack)

void reallocate_stack(stack * a_stack)

void push(stack* a_stack, int item)

int pop(stack* a_stack)

int peek(stack* a_stack)

void free_stack_memory(stack * a_stack)

void print_stack_info(stack * a_stack)
```

These method descriptions are in the starter code file `starter_code/design/stack_methods.txt`.

Use the techniques described in this chapter (opaque types, etc.) to build the module.

The file `use_stack.c` in the starter code folder `starter_code/design` contains most of the `main` function used to call stack functions.

The output should be:

```
0 pushed to stack
1 pushed to stack
2 pushed to stack
3 pushed to stack
Stack filled - attempting to reallocate array to 8 elements
4 pushed to stack
5 pushed to stack
6 pushed to stack
7 pushed to stack
Stack filled - attempting to reallocate array to 16 elements
8 pushed to stack
9 pushed to stack

Stack has capacity: 16
Currently, stack has 10 elements

9 popped from stack
8 popped from stack

Stack has capacity: 16
Currently, stack has 8 elements

Freeing data used for the stack...
```

As with the previous exercise, the `main` function has a `printf` that attempts to *directly access a field of the stack struct*. Remove the comment and rebuild. The compiler should complain:

```
use_stack.c: In function ‘main’:
use_stack.c:9:42: error: dereferencing pointer to incomplete type ‘stack {aka struct␣
↪stack}’
  printf("Stack capacity = %d\n", a_stack -> capacity);
```

This shows the wordlist type *is opaque and encapsulates its fields*.

2.8.3 Exercise 3

Examine the program `design_03.c`, paying attention to the functions that use the `emp_info` struct.

Examine the file `data.txt`; the second column is the *employee number*.

Run the program first, supplying valid and invalid employee numbers.

A sample run is as follows:

```
Enter an employee ID or 0 to quit ==> 1000
Employee ID: 1000       Name: Employee_0         Dept ID: 21     Salary: 170844.
↪920000   Hire Date:12/22/2007

Enter an employee ID or 0 to quit ==> 1200
Employee ID: 1200       Name: Employee_200       Dept ID: 11     Salary: 59696.200000␣
↪   Hire Date:4/24/1998

Enter an employee ID or 0 to quit ==> 5000
No employee for ID number 5000 found


Enter an employee ID or 0 to quit ==> 1400
Employee ID: 1400       Name: Employee_400       Dept ID: 34     Salary: 960385.
↪070000   Hire Date:11/21/2017

Enter an employee ID or 0 to quit ==> 0
```

Code an **interface** that allows the functions that access the struct and a *separate implementation* containing those functions.

There will be changes required to the program *to use pointers* where struct instances are used to recode the program to use an opaque type.

Build and run the program **with valgrind** using the same inputs as above. The results should be identical with no memory leaks or other memory errors.

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

2.9 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.

The program `bitwise.c` shows these operations in use:

```
#include <stdio.h>

int main(void)
```

(continues on next page)

(continued from previous page)

```
{
   unsigned int a = 60; /* 60 = 0011 1100 */
   unsigned int b = 13; /* 13 = 0000 1101 */

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

   /* 61 = 0011 1101 */
   printf("BIT OR : %d | %d is %d\n", a, b, a | b );

   /* 49 = 0011 0001 */
   printf("BIT EXCLUSIVE OR: %d ^ %d is %d\n", a, b, a ^ b);

   /* 61 = 1100 0011 */
   printf("COMPLIMENT: ~%d is %d\n", a, ~a );

   /* 240 = 1111 0000 */
   printf("LEFT SHIFT 2: %d << 2 is %d\n", a, a << 2 );

   /* 15 = 0000 1111 */
   printf("RIGHT SHIFT 2: %d >> 2 is %d\n", a,a >> 2 );
}
```

which produces this output:

```
BIT AND: 60 & 13 is 12
BIT OR : 60 | 13 is 61
BIT EXCLUSIVE OR: 60 ^ 13 is 49
COMPLIMENT: ~60 is -61
LEFT SHIFT 2: 60 << 2 is 240
RIGHT SHIFT 2: 60 >> 2 is 15
```

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*.

The program `compoundbit.c` provides an example:

```
#include <stdio.h>

int main(void) {

   unsigned int a = 60; /* 60 = 0011 1100 */
   unsigned int b = 13; /* 13 = 0000 1101 */

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

   /* a = 12 = 0000 1100 */
   printf("COMPOUND LEFT SHIFT 2 BITS: %d =<<2 is ", a );
   printf("%d\n", a <<= 2);
   printf("Value of a is now %d\n", a);
   /* a is now 48 = 0011 0000 */
```

(continues on next page)

(continued from previous page)

```
   /* Watch out for the unexpected! */
   /* Avoid using compound assignments in printf statements! */
   printf("\nCOMPOUND LEFT SHIFT 2 BITS: %d =<<2 is %d\n", b, b <<= 2 );
}
```

which prints:

```
COMPOUND BIT AND: 12 & 13 is 12
Value of a is now 12
COMPOUND LEFT SHIFT 2 BITS: 12 =<<2 is 48
Value of a is now 48

COMPOUND LEFT SHIFT 2 BITS: 52 =<<2 is 52
```

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

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: 52 =<<2 is 52`. The above applies to *any compound operator* - not just the bit operators.

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

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

The program `twos_comp.c` shows this:

```
#include <stdio.h>

int main(void)
{
   int a = 60;
   int b = -13;

   printf("COMPLIMENT + 1 NEGATES NUMBER: ~%d is %d\n", a, ~a + 1);
   printf("COMPLIMENT + 1 NEGATES NUMBER: ~%d is %d\n", b, ~b + 1);
}
```

which prints:

```
COMPLIMENT + 1 NEGATES NUMBER: ~60 is -60
COMPLIMENT + 1 NEGATES NUMBER: ~-13 is 13
```

The above program prints the *twos compliment* of the numbers `60` and `-13`.

2.10 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, so 5 bits
    // are sufficient
    unsigned int day : 5;

    // month has value between 1 and 12, so 4 bits
    // are sufficient
    unsigned int month : 4;

        // Structs can contain non-bitfield members as well as bitfields
    unsigned int year;
};
```

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

```
#include <stdio.h>
// Using bitfields
struct date {
    unsigned int day;
    unsigned int month;
    unsigned int year;
};

struct date_with_bit_fields {
        // day has value between 1 and 31, so 5 bits
    // are sufficient
    unsigned int day : 5;
    // month has value between 1 and 12, so 4 bits
    // are sufficient
    unsigned int month : 4;
        // Structs can contain non-bitfield members as well as bitfields
    unsigned int year;
};

int main(void)
{
    struct date dt = { 31, 12, 2019 };
    printf("Size of date struct with int fields %lu bytes\n", sizeof(dt));
    printf("Date created from is date struct with int fields %d/%d/%d\n\n",
                dt.month, dt.day, dt.year);

        struct date_with_bit_fields dtbf = { 31, 12, 2019 };
    printf("Size of date struct with bit fields %lu bytes\n", sizeof(dtbf));
    printf("Date created from is date struct with bit fields %d/%d/%d\n",
                dtbf.month, dtbf.day, dtbf.year);
}
```

which prints:

```
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 12/31/2019
```

- 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.

The program `bitfields3.c`, not shown here, has the above assignment statement:

```
struct date_with_bit_fields dtbf = { 40, 12, 2019 }; // 40 too big!!!
```

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

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

`cc` complains:

```
bitfield3.c: In function ‘main’:
bitfield3.c:16:39: warning: large integer implicitly truncated to unsigned type [-
↪Woverflow]
  struct date_with_bit_fields dtbf = { 40, 12, 2019 }; // 310 too big!!!
```

but the program executes:

```
Size of date struct with bit fields 8 bytes
Date created from is date struct with bit fields 12/8/2019
```

`40` in binary is `101000`. 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 `bitfields2.c`, not shown here, has the above struct `date_with_bit_fields`, coded with `int` members and used in the above program. When executed, `bitfields2.c` 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_alignment {
    unsigned int day : 5;
        // Force alignment of next bitfield on four-byte boundary
        unsigned int : 0;
    unsigned int month : 4;
    unsigned int year;
};

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

int main(void)
{
    struct force_alignment dtfa = { 31, 12, 2019 };
    printf("Size of date struct with alignment %lu bytes\n", sizeof(dtfa));

        struct date_with_bit_fields dtbf = { 31, 12, 2019 };
    printf("Size of date struct with bit fields %lu bytes\n", sizeof(dtbf));

}
```

which prints:

```
Size of date struct with alignment 12 bytes
Size of date struct with bit fields 8 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

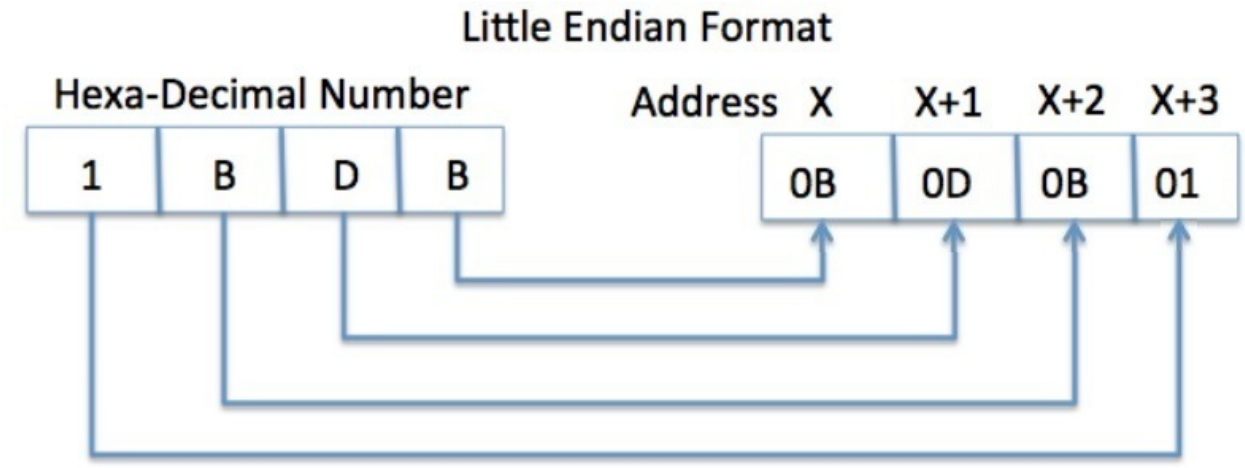
No pointers to bitfields: Since bitfields are not guaranteed to be aligned on a byte, there cannot be a pointer containing the address of a bitfield. Pointers to structs containing bitfields is permissable.

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.

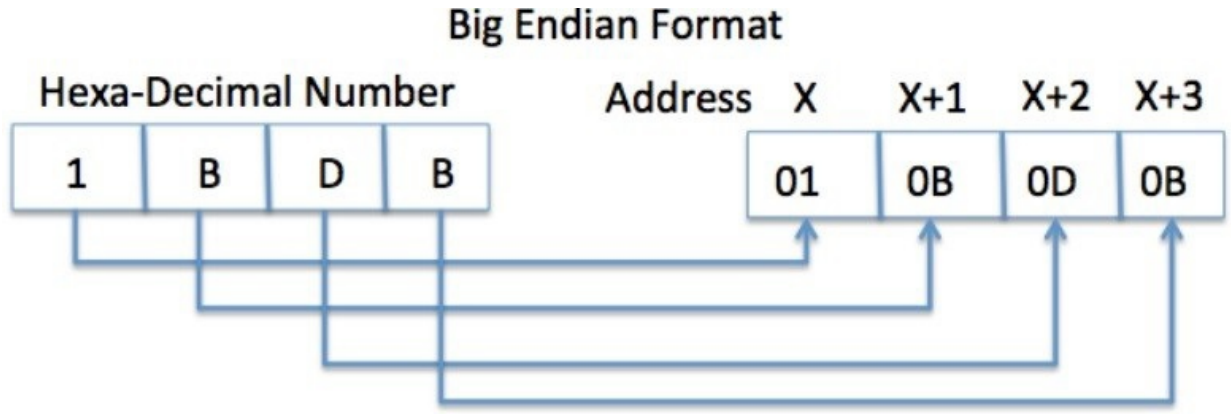
2.11 Endianness

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*.



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



The following program `little_or_big.c` outputs the byte order of the host platform:

```
#include <stdio.h>
// Big endian or little endian?
int main(void)
{
        unsigned int i = 1;
        char *c = (char*)&i;
        if (*c)
                printf("Little endian\n");
        else
                printf("Big endian\n");
}
```

which prints:

```
Little endian
```

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:

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, `endian_functions.c`, shows a little-endian byte order integer converted to a network (big endian) byte order integer, then back:

```
#include <stdio.h>
#include <inttypes.h>
#include <arpa/inet.h>
//include <netinet/in.h> could also be used here on some systems
int main(void)
{
        uint16_t short_num = 0x1234;

        printf("Original short           - 0x%x\n", short_num);
    printf("Network short from htons - 0x%x\n", htons(short_num));
    printf("Host short from ntohs    - 0x%x\n", ntohs(short_num));
        printf("Back to original short   - 0x%x\n\n", htons(ntohs(short_num)));

    uint32_t long_num = 0x12345678;

    printf("Original long           - 0x%x\n", long_num);
    printf("Network long from htonl - 0x%x\n", htonl(long_num));
    printf("Host long from ntohl    - 0x%x\n", ntohl(long_num));
        printf("Back to original long   - 0x%x\n", htonl(ntohl(long_num)));
}
```

The output follows:

```
Original short           - 0x1234
Network short from htons - 0x3412
Host short from ntohs    - 0x3412
Back to original short   - 0x1234

Original long           - 0x12345678
Network long from htonl - 0x78563412
Host long from ntohl    - 0x78563412
Back to original long   - 0x12345678
```

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

The lines below write the same number:

```
printf("Network short from htons - 0x%x\n", htons(short_num));
printf("Host short from ntohs    - 0x%x\n", ntohs(short_num));


Network short from htons - 0x3412
Host short from ntohs    - 0x3412
```

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**.

2.12 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.

The program `bin-io-array.c` shows using `fwrite` to create a binary file and `fread` to read the created file and print its contents:

```
#include <stdio.h>

enum { SIZE = 5 };
int main(void)
{
        double a[SIZE] = {1.,2.,3.,4.,5.};
        FILE *fp;
        size_t num_read_or_written;
        if ((fp = fopen("test.bin", "wb")) == NULL)
        {
                perror("Could not open 'test.bin'");
                exit(EXIT_FAILURE);

        } else
        if ((num_read_or_written = fwrite(a, sizeof(*a), SIZE, fp) != SIZE))
        {
                printf("Write incomplete: Only wrote %zu; should have written %d\n",
                       num_read_or_written, SIZE);
                perror("Failure on write to 'test.bin'");
                // Close the file here - clean up before exit!!
                fclose(fp);
                exit(EXIT_FAILURE);
        }
        // If program gets here, open and write successful; close file
        fclose(fp);

    double b[SIZE];
        if ((fp = fopen("test.bin", "rb")) == NULL)
        {
                perror("Could not open 'test.bin'");
                exit(EXIT_FAILURE);

        } else
        if ((num_read_or_written = fread(b, sizeof(*b), SIZE, fp) != SIZE))
        {
                printf("Read incomplete: Only read %zu; should have read %d\n",
                       num_read_or_written, SIZE);
                perror("Failure on read from 'test.bin'");
                // Close the file here - clean up before exit!!
                fclose(fp);
                exit(EXIT_FAILURE);
        }
        fclose(fp);

        // If program gets here, print array and close file
        for(int n = 0; n < SIZE; ++n)
                printf("%f ", b[n]);
        putchar('\n') ;
}
```

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

```
1.000000 2.000000 3.000000 4.000000 5.000000
```

The statement:

```
fwrite(a, sizeof(*a), SIZE, fp);
```

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.

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

enum { MAX_NAME_SIZE = 20, NUM_EES = 5 };

typedef struct employee {
        char * name;
        // emp_id from 0 to 5
        unsigned int emp_id: 3;
        // grade from 0 to 31
        unsigned int grade: 5;
        // dept_id from 0 to 14
        unsigned int dept_id: 4;
        // Salary just a double
        double salary;
} employee;
// Return a random positive integer not exceeding upper_b
unsigned int random_number_from(int upper_b)
{
        return rand() % (upper_b + 1);
}
// Create an employee struct - use struct pointer as output parm
void create_employee(employee * an_emp)
{
        static unsigned int emp_id = 0;
        // Let's make this easy - get employee name from here
        char * emp_names[MAX_NAME_SIZE] = {"Emp0", "Emp1", "Emp2", "Emp3", "Emp4"};
        // Load 'em up
```

(continues on next page)

(continued from previous page)

```
        an_emp -> name = emp_names[emp_id];
        an_emp -> emp_id = emp_id++;
        an_emp -> grade = random_number_from(31);
        an_emp -> dept_id = random_number_from(14);
        an_emp -> salary = 100.125 * random_number_from(30000);
}

void print_emp(employee * an_emp)
{
        printf("Employee info for Employee ID %u\n", an_emp -> emp_id);
        printf("Name: %s\tGrade: %u\tDept ID: %u\tSalary: %.2f\n\n",
               an_emp -> name, an_emp -> grade, an_emp -> dept_id, an_emp -> salary);
}
// Was the file open successful?
void file_opened(FILE ** fp, char * file_name, char * file_mode)
{
        if ((*fp = fopen(file_name, file_mode)) == NULL)
        {
                char err_desc[100];
                sprintf(err_desc, "Could not open %s in %s mode", file_name, file_
 ↪mode);
                perror(err_desc);
                exit(EXIT_FAILURE);
        }
}
// Write file - check for correct number of characters
void file_written(FILE * fp, employee * an_emp)
{
        size_t num_read_or_written;
        if ((num_read_or_written = fwrite(an_emp, sizeof(*an_emp), 1, fp) != 1))
        {
                printf("Write incomplete: Only wrote %zu; should have written %d\n",
                       num_read_or_written, 1);
                perror("Failure on write");
                // Close the file here - clean up before exit!!
                fclose(fp);
                exit(EXIT_FAILURE);
        }
}
// Write file - check for correct number of characters
void file_read(FILE * fp, employee * an_emp)
{
        size_t num_read_or_written;
        if ((num_read_or_written = fread(an_emp, sizeof(*an_emp), 1, fp) != 1))
        {
                printf("Read incomplete: Only read %zu; should have read %d\n",
                       num_read_or_written, 1);
                perror("Failure on read");
                // Close the file here - clean up before exit!!
                fclose(fp);
                exit(EXIT_FAILURE);
        }
}
int main(void)
{
    FILE *fp ;
```

(continues on next page)

(continued from previous page)

```
        file_opened(&fp, "emp_data.bin", "wb");
        for (int e_id = 0; e_id < NUM_EES; e_id++)
        {
                employee an_emp;
                create_employee(&an_emp);
                print_emp(&an_emp);
                file_written(fp, &an_emp);
        }
    fclose(fp);
        printf("\nEE data written to file... Read back in and print\n");
        // Open file for read
    file_opened(&fp, "emp_data.bin", "rb");
        // Read one at a time

        for (int e_id = 0; e_id < NUM_EES; e_id++)
        {
                employee an_emp;
                file_read(fp, &an_emp);
                print_emp(&an_emp);
        }
    fclose(fp);
}
```

The output is:

```
Employee info for Employee ID 0
Name: Emp0      Grade: 7        Dept ID: 1      Salary: 1674390.38

Employee info for Employee ID 1
Name: Emp1      Grade: 19       Dept ID: 8      Salary: 2422524.38

Employee info for Employee ID 2
Name: Emp2      Grade: 10       Dept ID: 12     Salary: 677445.75

Employee info for Employee ID 3
Name: Emp3      Grade: 13       Dept ID: 2      Salary: 2504426.62

Employee info for Employee ID 4
Name: Emp4      Grade: 18       Dept ID: 4      Salary: 2964000.38


EE data written to file... Read back in and print
Employee info for Employee ID 0
Name: Emp0      Grade: 7        Dept ID: 1      Salary: 1674390.38

Employee info for Employee ID 1
Name: Emp1      Grade: 19       Dept ID: 8      Salary: 2422524.38

Employee info for Employee ID 2
Name: Emp2      Grade: 10       Dept ID: 12     Salary: 677445.75

Employee info for Employee ID 3
Name: Emp3      Grade: 13       Dept ID: 2      Salary: 2504426.62

Employee info for Employee ID 4
Name: Emp4      Grade: 18       Dept ID: 4      Salary: 2964000.38
```

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 uses functions to verify successful file open, read and writes.

Of interest is the function definition for the *file open* function, `file_opened`:

```
void file_opened(FILE ** fp, char * file_name, char * file_mode)
```

The `fopen` function *modifies the file pointer*. Since this file pointer is used for subsequent writes and reads, the modified value *must be available to the calling function* (`main`, in this case).

The call to `file_opened` is:

```
file_opened(&fp, "emp_data.bin", "wb");
```

The first argument is a *pointer to a FILE pointer*, which will allow the `file_opened` function to modify the file pointer.

Both the read and write functions `file_read` and `file_write` should **close the file** before exiting the program.

2.13 Exercises

:!exercise:

2.13.1 Exercise 1

Write a program, `gen_ulong_randoms.c` that generates and prints 20 *unsigned long* random numbers.

Print out the maximum value of the `rand` function found in `stdlib.h`.

The output should resemble:

```
Max random number generated by rand() function: 2147483647

7749363893351949254
7222815480849057907
8408462745175416063
3091884191388096748
2562019500164152525
4403210617922443179
3364542905362882299
8782769017716072774
5863405773976003266
1306053050111174648
150346236956174824
1265737889039205261
1445109530774087002
1197105577171186275
9213452462461015967
4730966302945445786
5650605098630667570
5880381756353009591
4552499520046621784
2697991130065918298
```

2.13.2 Exercise 2

Write a program, `bin_rep_dec_num.c` that generates a *bit string* that represents the binary value (digits) of a number, fetched by user console input in response to a prompt.

The program should have a function that *creates the bit string* and returns that string to the calling function. Using this string as an argument, the program calls a function that prints the bits *in groups of 4, 8 and 16*.

Sample output follows:

```
Enter an integer number :125698

Binary value of 125698 is = 00000000000000011110101100000010
Grouped by 4 bits
0000 0000 0000 0001 1110 1011 0000 0010
Grouped by 8 bits
00000000 00000001 11101011 00000010
Grouped by 16 bits
0000000000000001 1110101100000010

Enter an integer number :-120

Binary value of -120 is = 11111111111111111111111110001000
Grouped by 4 bits
1111 1111 1111 1111 1111 1111 1000 1000
Grouped by 8 bits
11111111 11111111 11111111 10001000
Grouped by 16 bits
1111111111111111 1111111110001000
```

There is no requirement to *check for a valid input*.

Run `valgrind` to ensure there are no memory leaks.

2.13.3 Exercise 3

Copy the program in the startup folder `startup_code/bit_manipulation/change_structs.c`, which contains several ``struct``s, and *create new structs* that may contain the same data but are *smaller in size*. The startup file has the ``struct``s coded and comments on the allowable values for the `struct` fields.

Output should be:

```
Size original address struct: 40
Size new address struct:      24

Size original employee struct: 80
Size new employee struct:      48
```

2.13.4 Exercise 4

Write a program, `flip_bits.c` that prints the number of bits that must flip from `0` to `1` or `1` to `0` to change one number into another.

Sample output follows:

```
Enter the first number ==> 10
Enter the second number ==> 254
Must flip 5 bits to change 10 to 254
```

2.14 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.

2.15 `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 int_and_remainder = div(100, 11);
        printf("Integer part: %d\tRemainder part: %d\n",
              int_and_remainder.quot, int_and_remainder.rem);
}
```

which prints:

```
Integer part: 9 Remainder part: 1
```

- 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`:

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

void use_strtod(char * parse_me)
{
        char *ptr_to_unparsed_string;
        double parsed_val = strtod(parse_me, &ptr_to_unparsed_string);
        printf("The number(double) is %lf\n", parsed_val);
        printf("String part is '%s'\n\n", ptr_to_unparsed_string);
}

int main (void)
{
        use_strtod("2.71828 and string data");

        use_strtod("2.718281828");
```

(continues on next page)

(continued from previous page)

```
        use_strtod("No match 2.718281828");

    // Repeatedly call strtod to parse all numbers in string
        char * nums_in_str = "100.5  2.718 3.14159 and more";
        double parsed_val;
        char * unparsed;
        while( (parsed_val = strtod(nums_in_str, &unparsed)) != 0)
        {
                printf("The number(double) is %lf\n\n", parsed_val);
                // Reassign nums_in_str and continue
                nums_in_str = unparsed;
        }
}
```

which produces:

```
The number(double) is 2.718280
String part is ' and string data'

The number(double) is 2.718282
String part is ''

The number(double) is 0.000000
String part is 'No match 2.718281828'

The number(double) is 100.500000

The number(double) is 2.718000

The number(double) is 3.141590
```

Note by reassigning the unparsed portion of the string `unparsed` to `nums_in_str`, 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:

```
/* strtol example */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
        char numbers[] = "2001 60c0c0 -110111010011010010000 0x6fffff";
        char * unparsed;

        long li1;
        long li2;
        int  li3;
        int  li4;

        li1 = strtol (numbers, &unparsed, 10);
        li2 = strtol (unparsed, &unparsed, 16);
        li3 = strtol (unparsed, &unparsed, 2);
        li4 = strtol (unparsed, &unparsed, 0);

        printf ("The decimal equivalents are: %ld, %ld, %d and %d.\n", li1, li2, li3,␣
 ↪li4);
}
```

which produces:

```
The decimal equivalents are: 2001, 6340800, -3624224 and 7340031.
```

If there were non-blank alpha characters between the numbers, the parsing would stop.

2.16 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:

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

int comp_strings (const void * a, const void * b ) {
    const char *pa = *(const char**)a;
    const char *pb = *(const char**)b;

    return strcmp(pa,pb);
}

int comp_nums(const void *a, const void *b)
{
        return *(int *)a - *(int *)b;
}

int main() {
    // Sort an array of strings:
    const char *string_arr[] = {"a","orange","apple","mobile","car"};

    size_t string_len = sizeof(string_arr) / sizeof(char *);
    qsort(string_arr, string_len, sizeof(char *), comp_strings);

        printf("Strings\n");
    for (int i=0; i < string_len; ++i)
        printf("%d: %s\n", i, string_arr[i]);

        // Sort an array of numbers:
        int nums[10] = {0,3,4,2,3,5,6,8,7,1};

        // Sort an array of integers
        size_t nums_len = sizeof(nums) / sizeof(nums[0]);
        qsort(nums, nums_len, sizeof(nums[0]), comp_nums);

        printf("\nNumbers\n");
    for (int i=0; i < nums_len; ++i)
        printf("%d: %d\n", i, nums[i]);

}
```

The output follows:

```
Strings
0: a
1: apple
2: car
3: mobile
4: orange

Numbers
0: 0
1: 1
2: 2
3: 3
```

(continues on next page)

(continued from previous page)

```
4: 3
5: 4
6: 5
7: 6
8: 7
9: 8
```

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.

2.17 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.

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

int comp_strings (const void * a, const void * b ) {
    const char *pa = *(const char**)a;
```

(continues on next page)

(continued from previous page)

```
    const char *pb = *(const char**)b;

    return strcmp(pa,pb);
}

int comp_nums(const void *a, const void *b)
{
        return *(int *)a - *(int *)b;
}

int main() {
    // Search an array of strings:
    const char *string_arr[] = {"a","orange","apple","mobile","car"};
        const char * find_me = "apple";
        const char * not_in_array = "banana";
        char ** found;

    size_t string_len = sizeof(string_arr) / sizeof(char *);
        // MUST SORT FIRST!!!
        qsort(string_arr, string_len, sizeof(char *), comp_strings);

    found = (char **)bsearch(&find_me, string_arr, string_len, sizeof(char *), comp_
↪strings);
        // Check if found
        if (found != NULL) {
                printf("Key found!!! %s\n",*found);
        } else {
                printf("Key not found!!!");
        }

    found = (char **)bsearch(&not_in_array, string_arr, string_len, sizeof(char *),␣
↪comp_strings);
        // Check if found - should not be found
        if (found != NULL) {
                printf("Key found!!! %s\n", *found);
        } else {
                printf("Key not found!!!\n");
        }
        puts("\n");

        // Search an array of numbers:
        int nums[10] = {0,3,4,2,3,5,6,8,7,1};
        int * found_int;
        int find_me_num = 5;
        int not_there = 100;
        size_t nums_len = sizeof(nums) / sizeof(nums[0]);
        // MUST SORT FIRST!!!
        qsort(nums, nums_len, sizeof(nums[0]), comp_nums);

    found_int = (int *)bsearch(&find_me_num, nums, nums_len, sizeof(nums[0]), comp_
↪nums);

        if (found_int != NULL) {
                printf("Key found!!! %d\n", *found_int);
        } else {
                printf("Key not found!!!\n");
        }
```

(continues on next page)

(continued from previous page)

```
    found_int = (int *)bsearch(&not_there, nums, nums_len, sizeof(nums[0]), comp_
↪nums);
        // Check if found - should not be found
        if (found_int != NULL) {
                printf("Key found!!! %d\n", *found_int);
        } else {
                printf("Key not found!!!\n");
        }
}
```

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)
```

2.18 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.

2.19 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`:

```
// 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)
        {
                switch(opt)
                {
                        case 'i':
                                printf("Processing option: %c\n", opt);
                                break;
                        case 'r':
                                printf("Processing option: %c\n", opt);
                                break;
                        case 'x':
                                printf("Processing option: %c\n", opt);
                                break;
                        case '?':
                                printf("unknown option: %c\n", optopt);
                                break;
                }
        }

        // 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
Processing option: i
./get: invalid option -- 'f'
unknown option: f
Processing option: x

$ ./get -i -f -x 'not an option' -i
Processing option: i
./get: invalid option -- 'f'
unknown option: f
Processing option: x
Processing option: i
extra arguments: not an option

$ ./get


$ ./get -irfrx
Processing option: i
Processing option: r
./get: invalid option -- 'f'
unknown option: f
Processing option: r
Processing option: x

$ ./get -xri
Processing option: x
Processing option: r
Processing option: i

$ ./get no good -xri here -t
Processing option: x
Processing option: r
Processing option: i
./get: invalid option -- 't'
unknown option: t
extra arguments: no
extra arguments: good
extra arguments: here
```

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
Processing option: x
unknown option: t
Processing option: 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 — *t*) 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.

```
// 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)
        {
                switch(opt)
                {
                        case 'i':
                        case 'l':
                        case 'r':
                                printf("Processing option: %c\n", opt);
                                break;
                        case 'f':
                                printf("Processing option f - passed filename: %s\n",␣
 ↪optarg);
                                break;
                        case '?':
                                printf("unknown option: %c\n", optopt);
                                break;
                }
        }

        // 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
Processing option: i
```

(continues on next page)

(continued from previous page)

```
Processing option f - passed filename: filename.txt
Processing option: r

$ ./get2 -f -r -i
Processing option f - passed filename: -r
Processing option: i

$ ./get2 -f "for opt f" -d -f "other f option val"
Processing option f - passed filename: for opt f
./get2: invalid option -- 'd'
unknown option: d
Processing option f - passed 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 in the program run with these command line arguments:

```
$ ./get2 -f -r -i
Processing option f - passed filename: -r
Processing option: i
```

2.20 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.

2.21 ..., 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 add_em_up (int count,...)
{
  va_list ap;                    /* ap represents the arguments passed to program */
  int i, sum;

  va_start(ap, count);          /* Initialize the argument list. */

  sum = 0;
  for (i = 0; i < count; i++)
    sum += va_arg (ap, int);     /* Get the next argument value. */

  va_end(ap);                   /* Clean up. */
  return sum;
}

int main (void)
{
  /* This call prints 16. */
  printf("%d\n", add_em_up (3, 5, 5, 6));

  /* This call prints 55. */
  printf("%d\n", add_em_up (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
}
```

which outputs:

```
16
55
```

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 two line of the above program with:

```
printf ("%d\n", add_em_up (3, 5, "A string", 6));
printf ("%d\n", add_em_up (10, 1, 2, "Not an int", 4, 5, 6, 7, 8, 9, 10));
```

and running the program prints:

```
278923519
278923573
```

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.

```
// Shows that the first argument to variadic function need not
// represent an element count
#include<stdio.h>
#include<stdarg.h>

void printargument(int required_but_just_the_first_arg, ...)
{
        va_list arg_list;
        int my_arg;
        // MUST code required_but_just_the_first_arg as the second arg to 'va_start'
        va_start(arg_list, required_but_just_the_first_arg);

        int arg_count = 0;
        //Print until zero
        for (my_arg = required_but_just_the_first_arg;
            my_arg != 0;
                my_arg = va_arg(arg_list, int))

                printf("arg index = %d\tPassed arg = %d\n", ++arg_count, my_arg);

        va_end(arg_list);
}

int main(void)
{
        printargument(5,10,15,0);
}
```

The output:

```
arg index = 1   Passed arg = 5
arg index = 2   Passed arg = 10
arg index = 3   Passed arg = 15
```

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.

2.22 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 process_buncha_stuff(void)
{
        double small_num = 1e-10;
        double sum = 0;
        long iters = 1000000000;
        for (long idx = 0; idx < iters; idx++)
                sum += small_num;

        printf("Iterating over %ld times: Sum = %f\n", iters,sum);
```

(continues on next page)

(continued from previous page)

```
}

int main(void)
{
        clock_t start = clock();
    // Time this function
        process_buncha_stuff( );
        clock_t end = clock() - start;
        double elapsed_CPU_time = (double)end / CLOCKS_PER_SEC;

        printf("Processing took %f seconds\n", elapsed_CPU_time);
}
```

which outputs:

```
Iterating over 1000000000 times: Sum = 0.100000
Processing took 3.031250 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.

2.23 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 seconds;

   seconds = time(NULL);
   // time(&seconds) also works
   printf("Hours since January 1, 1970 = %ld\n", seconds/3600);
}
```

which prints:

```
Hours since January 1, 1970 = 444412
```

- 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:

```
// Example of localtime function
#include <stdio.h>
#include <time.h>

int main (void)
{
   time_t rawtime;
   struct tm *info;

   time(&rawtime);
   info = localtime(&rawtime );


   printf("Local date from localtime function:\t");
   printf("%d/%2d, %04d\n", (info->tm_mon) + 1, info->tm_mday, info->tm_year + 1900);

   printf("Local time from localtime function:\t");
   printf("%2d:%02d:%02d\n", info->tm_hour, info->tm_min, info -> tm_sec);
}
```

which outputs:

```
Local date from localtime function:     9/12, 2020
Local time from localtime function:      0:03:39
```

The time reported is 12:03 AM

Note the offsets for `info -> tm_mon` and `info -> 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.

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

enum {BST = 1, CCT = 8};

int main (void)
{
   time_t rawtime;
   struct tm *info;

   time(&rawtime);
   /* Get GMT time */
   info = gmtime(&rawtime );

   printf("Current world clock:\n");
   printf("London : %2d:%02d\n", (info->tm_hour+BST)%24, info->tm_min);
   printf("China  : %2d:%02d\n", (info->tm_hour+CCT)%24, info->tm_min);
}
```

which outputs:

```
Current world clock:
London :  6:11
China  : 13:11
```

Adding hours to `info->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 process_buncha_stuff(void)
{
        double small_num = 1e-10;
        double sum = 0;
        long iters = 1000000000;
        for (long idx = 0; idx < iters; idx++)
                sum += small_num;

        printf("Iterating over %ld times: Sum = %f\n", iters,sum);
}

int main(void)
{
        struct timespec start, end;
        // Critical section, the part being measured
        timespec_get(&start, TIME_UTC);
        process_buncha_stuff();
        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;
        }
        // time_t defined as a long but may be different on other platforms
        printf("%ld.%09ld\n", (long)seconds, nanoseconds);
}
```

which prints:

```
Iterating over 1000000000 times: Sum = 0.100000
3.086458900
```

2.24 Exercises

:!exercise:

2.24.1 Exercise 1

Code a program named `my_print.c` that has a `my_print` function that prints characters, strings or integers.

The function, called from `main`, accepts a *format string* containing data types along with data to be printed.

When the `my_print` function is called from `main` as follows:

```
my_printf("sdc", "hi there", 25, 'x');
```

The output is:

```
string hi there
int 25
char x
```

2.24.2 Exercise 2

Code a program named `get_file_info.c` that accepts a *directory name* and an operation from the command line.

The `d` option specifies a directory and is required. The `l` option specifies the number of entries from the directory to process the `f` option, *if present*, will only process **files**.

```
./get_file_info -d /usr/bin -l 10 -f
```

produces the following table:

```
File Name  File?   Dir?    File Size
```

```
          2to3-2.7  Y       N       96
                NF  Y       N       963
     VGAuthService  Y       N       129248
                 [  Y       N       51384
        aa-enabled  Y       N       22696
           aa-exec  Y       N       22696
           aclocal  Y       N       36792
      aclocal-1.15  Y       N       36792
       acpi_listen  Y       N       14608
add-apt-repository  Y       N       7258
```

This run:

```
./get_file_info -d /usr/bin -l 5
```

produces the following table:

```
File Name  File?   Dir?    File Size
```

```
     2to3-2.7  Y       N       96
           NF  Y       N       963
VGAuthService  Y       N       129248
          X11  N       Y       0
            [  Y       N       51384
```

Report on any invalid or extra arguments.

Feel free to use code written in the C_I module that had a lab that produces the same table.

2.24.3 Exercise 3

Code a program named 'process_nums.c' that accepts command-line arguments that are **any arithmetic type**.

The program computes the *sum, average, min and max* of the numbers entered. Skip any arguments not of an arithmetic type, printing a message.

This run:

```
./process_nums 25 100.5 200.5 "not a num" 25
```

produces the following output:

```
'not a num' is skipped - non arithmetic type

Sum = 351
Min = 25
Max = 200.5
Avg = 87.75
```

2.24.4 Exercise 4

Code a program named `sort_strings_by_length` that sorts an array of strings by the length of each string element.

For the string array:

```
{"banana bread", "kiwi", "Mighty Thor", "x", "apple","car"}
```

The resultant array could be:

```
{"x", "car", "kiwi", "apple", "Mighty Thor", "banana bread"}
```

Use the above string array in the program if you like.

2.24.5 Exercise 5

Code a program `clock_versus_cpu.c` which times a process and reports on *elapsed, calendat time* versus *CPU time*.

The program will start the clock to measure CPU time *after the user enters a number*. The program will start the clock to measure wall/calendar time *before the user enters a number*.

Here is a sample execution:

```
Wall Clock started. Enter a number to start the CPU clock ==> 5
Iterating over 1000000000 times: Sum = 0.100000
CPU Processing took 3.062500 seconds
Wall/Calendar time took 10.612262600 seconds
```

The number entered at the prompt does not matter.

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.

2.25 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.

2.26 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{nbsp}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;
};
```

2.27 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.

2.28 `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.

2.28.1 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{nbsp}seconds`, followed by `calls`. Each column may help in identifying bottlenecks. The two `self` columns, `self{nbsp}seconds` and `self{nbsp}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?

2.28.2 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`.

2.29 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/wechlin/x/maze/drich/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/wechlin/x/maze/drich/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)
```

(continues on next page)

(continued from previous page)

```
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.

2.30 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.

2.30.1 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.

2.31 Exercises

:!exercise:

2.31.1 Exercise 1

Is the code below faster or slower than `strchr`?

```
char *find_char(const char *s, int c)
{
        if (!s) {
                return NULL;
        }

        while (*s && *s != c) {
                ++s;
        }

        return *s ? (char *)s : NULL;
}
```

Write a program that times the execution of the above function and `strchr` when executed *10, 100 and 1,000 times*. Code the program to run these functions for three scenarios:

- The letter searched for is at the *end of the string*.
- The letter searched for is at the *start of the string*.
- The letter searched for is *not in the string*.

Time the executions when applied to a string of *random lowercase letters* of sizes *10,000,000 and 20*.

The time spent should be printed in **nanoseconds**.

The output should resemble:

```
Character 'Z' at end of string: time in nanos.
Length big string: 10000000     Length small string: 20
# times called           strchr(big str)  find_char(big str)     strchr(20 char) ␣
 ↪find_char(20 char)

10                                5744600           217328800                3500     ␣
 ↪          1400
100                              57584400          2185021800                1200     ␣
 ↪          6400
1000                            575282200         21836110400                4600     ␣
 ↪         38900

Character 'Z' at start of string: time in nanos.
Length big string: 10000000     Length small string: 20
# times called           strchr(big str)  find_char(big str)     strchr(20 char) ␣
 ↪find_char(20 char)

10                                    400                500                 500     ␣
 ↪           600
100                                   700                800                 700     ␣
 ↪           900
1000                                 2900               4100                3000     ␣
 ↪          4200
```

(continues on next page)

(continued from previous page)

```
Character 'X' not in string: time in nanos.
Length big string: 10000000     Length small string: 20
# times called           strchr(big str)  find_char(big str)     strchr(20 char) ␣
↪find_char(20 char)

10                                5603300           217193100                  600     ␣
↪          1500
100                              57750100          2170159800                  800     ␣
↪          4800
1000                            575625200         21757038300                 5700     ␣
↪         49300
```

2.31.2 Exercise 2

The program `profiling_02.c` in the startup folder `startup/profiling` executes two functions `500,000,000` times. This program could be improved by changing the code in the functions to increase performance.

The thrust of this exercise is to use some tools and determine if these tools provide any useful information *for this program* to provide hints on how the program may be changed to improve performance.

The student will:

- Code a missing implementation of a function
- Build the program to be processed by `gprof`
- Run the program *standalone* to get the timing outputs and to create the `gmon.out` file required for `gprof`
- Run `gprof` and examine the `gprof` results (may use *brief* (`-b`) option on `gprof`)
- Run `valgrind` with required options to run `callgrind_annotate`
- Run `callgrind_allocate1`
- Examine the output from the `callgrind_annotate` program, looking for *frequently executing lines*
- Change the source code to increase performance
- Once performance improvements are noticed, repeat the steps from *Run `gprof` and examine results* and note changes in the output files produced by the tools

The changes allowed are:

- Changing the parameters passed to functions
- Changing the contents of any code in the loop bodies for the functions `find_max_min_array` and `add_the_evens`

```
for (int i = 0; i < NUM_ITERS; ++i)
{
        // Loop body here may be changed
}
```

The changes *not allowed* are:

- Changing the value of the constant `NUM_ITERS`
- Changing the data in *any array* in `main`

The remainder of the exercise description is hints and commands to implement the above steps:

—

The program has code to *time the functions execution*.

Copy this program and note the program is missing a function implementation:

```
long get_nanos(void)
{
   // Write code to return the number of nanoseconds
   // between the function return and the epoch
}
```

Build the program with the implementation to create an output file suitable for input to the `gprof` profiler.

Run the program *standalone*; the output should resemble:

```
Sum of evens is 5722
Time spent in add_the_evens: Nanos: 401660800   Seconds: 0.401661
Sum of evens is 66
Time spent in add_the_evens: Nanos: 366264300   Seconds: 0.366264
Sum of evens is 26
Time spent in add_the_evens: Nanos: 361907200   Seconds: 0.361907
Sum of evens is 792
Time spent in add_the_evens: Nanos: 359195300   Seconds: 0.359195

Min = 5 Max = 5654
Time spent in find_max_min_array: Nanos: 451100200      Seconds: 0.451100
Min = 10        Max = 56
Time spent in find_max_min_array: Nanos: 450478800      Seconds: 0.450479
Min = 12        Max = 17
Time spent in find_max_min_array: Nanos: 454498800      Seconds: 0.454499
Min = 101       Max = 560
Time spent in find_max_min_array: Nanos: 454738300      Seconds: 0.454738


Elapsed nanos All functions - 3308039200        Seconds: 3.308039
```

The program runs quicker *without the instrumentation flag* `-pg` in the compile and link step. The student is invited to build the program *without the* `pg` *flag* to note the difference.

Generate a *profile* with the `gprof` tool. Redirect the `gprof` output to a text file for browsing. Does the profile identify any useful information that helps in improving performance?

Run the program with `valgrind`, using the plugin `callgrind`. The program takes a while - best to execute the program as a *background task*; append the command with a space and `&`. Redirect the output for browsing.

Note, the timings saved in the redirected output from `valgrind` are **much slower** than running the program standalone.

The shell prints:

```
[1] 19518
==19518== Callgrind, a call-graph generating cache profiler
==19518== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==19518== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==19518== Command: ./pro2
==19518==
==19518== For interactive control, run 'callgrind_control -h'.
==19518== error calling PR_SET_PTRACER, vgdb might block
$
```

19518 is the PID of the valgrind program running in the background.

When valgrind finishes executing in the background, the student will see:

```
$ ==19518==
==19518== Events    : Ir
==19518== Collected : 32950224711
==19518==
==19518== I   refs:      32,950,224,711

[1]+  Done                    valgrind --tool=callgrind ./pro2 > pro2valg.data
```

After completion there should be a file:

```
callgrind.out.19518
```

in the working directory.

Run the callgrind_annotate tool using the above file as a background task redirecting the output with the following command:

```
callgrind_annotate --auto=yes callgrind.out.19518 > callgrind.out
```

This command executes quickly; the saved data in the file callgrind.out is readily available.

Look over callgrind.out and note frequently executing instructions:

—

Make changes to the program and run the program to see if changed resulted in better performance. After noting improved performance, repeat the steps to run gprof, valgrind, and callgrind_annotate to generate **new output** files. Compare the outputs from the previous run with those of the current run and note any changes.

The output from the functions should be the same. Any changes should not affect the function's workings.

The sums produced by add_the_evens and find_max_min_array must be the same for all runs.

Here is an example of the timings from an improved version:

```
Sum of evens is 5722
Time spent in add_the_evens: Nanos: 114190700   Seconds: 0.114191
Sum of evens is 66
Time spent in add_the_evens: Nanos: 106370100   Seconds: 0.106370
Sum of evens is 26
Time spent in add_the_evens: Nanos: 105761800   Seconds: 0.105762
Sum of evens is 792
Time spent in add_the_evens: Nanos: 105069000   Seconds: 0.105069

Min = 5 Max = 5654
Time spent in find_max_min_array: Nanos: 427621300      Seconds: 0.427621
Min = 10        Max = 56
Time spent in find_max_min_array: Nanos: 430068300      Seconds: 0.430068
Min = 12        Max = 17
Time spent in find_max_min_array: Nanos: 431497400      Seconds: 0.431497
Min = 101       Max = 560
Time spent in find_max_min_array: Nanos: 431223700      Seconds: 0.431224


Elapsed nanos All functions - 2154424100        Seconds: 2.154424
```

2.31.3 Exercise 3

The program `profiling_03.c`, located in the startup folder `startup_code/profiling` contains an implementation of a function, `find_UC_charsOrig`, that saves and prints the upper case letters of its string argument. Assume the function `find_UC_charsOrig` is coded to accept *only alpha characters and a space* in the string argument.

The program in the starter code folder contains hints and stubs for various functions and structures. Make a copy of this program and make the changes stated in the hints and comments.

When completed, the program should contain several implementations of the function `find_UC_charsOrig` that have improved performance than the original.

The starter code program has stubs for *three* revisions to the original, slow version. Code one revision, run the program and note if your changes improved performance. Make a *small change* to the original in the revisions.

Below are a sample execution with one revision:

```
***Original slow function***
Upper case characters of 'hE llo Th ere' are 'ET'
Time spent in find_UC_charsOrig: Nanos: 853766900       Seconds: 0.853767
Upper case characters of 'Th is Is a Str INg' are 'TISIN'
Time spent in find_UC_charsOrig: Nanos: 1538448600      Seconds: 1.538449
Upper case characters of 'And AN Ot her one' are 'AANO'
Time spent in find_UC_charsOrig: Nanos: 1418378600      Seconds: 1.418379
Upper case characters of 'o nE MOre' are 'EMO'
Time spent in find_UC_charsOrig: Nanos: 541801400       Seconds: 0.541801

Elapsed nanos for this version - 4355680400     Seconds: 4.355680

***Changed this to that, moved a line, etc.***
Upper case characters of 'hE llo Th ere' are 'ET'
Time spent in find_UC_charsR1: Nanos: 329551600 Seconds: 0.329552
Upper case characters of 'Th is Is a Str INg' are 'TISIN'
Time spent in find_UC_charsR1: Nanos: 475279400 Seconds: 0.475279
Upper case characters of 'And AN Ot her one' are 'AANO'
Time spent in find_UC_charsR1: Nanos: 438123700 Seconds: 0.438124
Upper case characters of 'o nE MOre' are 'EMO'
Time spent in find_UC_charsR1: Nanos: 244994900 Seconds: 0.244995

Elapsed nanos for this version - 1490056000     Seconds: 1.490056
```

Note the performance improvement.

The description for the revision:

```
Changed this to that, moved a line, etc
```

should be more meaningful.

Add another version and run the revised program:

```
***Original slow function***
Upper case characters of 'hE llo Th ere' are 'ET'
Time spent in find_UC_charsOrig: Nanos: 878564000       Seconds: 0.878564
Upper case characters of 'Th is Is a Str INg' are 'TISIN'
Time spent in find_UC_charsOrig: Nanos: 1528078200      Seconds: 1.528078
Upper case characters of 'And AN Ot her one' are 'AANO'
Time spent in find_UC_charsOrig: Nanos: 1431795700      Seconds: 1.431796
Upper case characters of 'o nE MOre' are 'EMO'
```

(continues on next page)

(continued from previous page)

```
Time spent in find_UC_charsOrig: Nanos: 533938900       Seconds: 0.533939

Elapsed nanos for this version - 4375129600     Seconds: 4.375130

***Changed this to that, moved a line, etc.***
Upper case characters of 'hE llo Th ere' are 'ET'
Time spent in find_UC_charsR1: Nanos: 331800500 Seconds: 0.331800
Upper case characters of 'Th is Is a Str INg' are 'TISIN'
Time spent in find_UC_charsR1: Nanos: 470418600 Seconds: 0.470419
Upper case characters of 'And AN Ot her one' are 'AANO'
Time spent in find_UC_charsR1: Nanos: 439304700 Seconds: 0.439305
Upper case characters of 'o nE MOre' are 'EMO'
Time spent in find_UC_charsR1: Nanos: 245245300 Seconds: 0.245245

Elapsed nanos for this version - 1488878900     Seconds: 1.488879

***R1 changes changed this to that***
Upper case characters of 'hE llo Th ere' are 'ET'
Time spent in find_UC_charsR2: Nanos: 361803700 Seconds: 0.361804
Upper case characters of 'Th is Is a Str INg' are 'TISIN'
Time spent in find_UC_charsR2: Nanos: 517189000 Seconds: 0.517189
Upper case characters of 'And AN Ot her one' are 'AANO'
Time spent in find_UC_charsR2: Nanos: 476540500 Seconds: 0.476540
Upper case characters of 'o nE MOre' are 'EMO'
Time spent in find_UC_charsR2: Nanos: 265144700 Seconds: 0.265145

Elapsed nanos for this version - 1623300500     Seconds: 1.623301
```

Apparently, the second revision did not improve performance of the first revision, but is a substantial improvement over the original.

Add another revision and execute, or change the second version and execute.

After three revisions are coded, build the program for `gprof` use, run `valgrind` in preparation for running the `callgrind_annotate` program and run `callgrind_allocate`.

As with exercise 2 above, the `valgrind` execution may take some time; running `valgrind` as a background task is a good idea.

When done, examine the outputs of `gprof` and `callgrind_annotate`. The outputs should have statistics on *all four versions*. See if the contents of the outputs help explain why some versions execute faster than others.

2.32 Objectives

Gain a better understanding of previously covered concepts of the C programming language through the completion of practical exercises.

2.33 Exercises

:!exercise:

2.33.1 Exercise 1

Code a program named `my_var_print.c` that has a `my_print` function that prints characters, strings or integers.

The function, called from `main`, accepts a *format string* containing data types along with data to be printed.

When the `my_print` function is called from `main` as follows:

```
my_printf("%s%dc", 10, "hi there", 5, 25, 'x');
```

The output is:

```
int 10
string 'hi there'
int 5
int 25
char x
```

2.33.2 Exercise 2

Write a program, `time_search_methods.c` that times *using sequential access versus sorting first* to the min, second smallest, max and second largest of an array of *unsigned long integers*.

The program should accept command line arguments that specify the *size* of the array **as a positive power of 10**.

Check if the *size* parameter does not generate a number that exceeds the largest `unsigned long` value. Print appropriate messages to show the argument's invalid value and reason the value is invalid. Exit on invalid argument values.

Sample output follows:

```
$ ./time_search_methods -s
usage:  time -s pwr_of_10

$ ./time_search_methods -s this
Could not parse power of 10 from 'this'

$ ./time_search_methods -s 5this
Could not parse power of 10 from '5this'

$ ./time_search_methods -s 55555
Entered value results in array size 280047383447157399926560074016125412255029264384.
 ↪000000, which exceeds 18446744073709551615

./time_search_methods -s 8
```

(continues on next page)

(continued from previous page)

```
Creating array with 100000000 elements
Finding min and max of the array via sequential access
From iteration:
Min = 159741148168
Max = 9223371878576622376
Second smallest = 220973299333
Second largest = 9223371737107624445
Min/Max via sequential access took 0.531250 seconds

Finding min and max of the array via qsort
From qsort:
Min = 159741148168
Max = 9223371878576622376
Second smallest = 220973299333
Second largest = 9223371737107624445
Qsort access took 25.593750 seconds
```

Your timings may differ.

Once the arguments are validated, define the array and *load it with the correct number of* ***random*** `unsigned long` *integers*. You will need a function to **generate random unsigned longs**. Feel free to use the function written in *exercise 1* in the *Bit manipulation* lesson.

You may need to *link the math library* into your program. The linker option `-lm` will do this.

2.33.3 Exercise 3

The program `dec_ops.c` in the startup folder `starter_code/lab02` has several expressions that perform arithmetic using the usual arithmetic operations.

Make a copy (`dec_versus_bit_ops.c`) and run the program. The output should resemble:

```
2 mod 8    = 2
2 times 33 = 66
2 is a power of 2

15 mod 8    = 7
15 times 33 = 495
15 is not a power of 2

4 mod 8    = 4
4 times 33 = 132
4 is a power of 2

99 mod 8    = 3
99 times 33 = 3267
99 is not a power of 2

101 mod 8    = 5
101 times 33 = 3333
101 is not a power of 2

256 mod 8    = 0
256 times 33 = 8448
256 is a power of 2
```

Change `dec_versus_bit_ops.c` by adding functions that use *bit operations and, if needed, a single arithmetic operation* that produce the same results as the arithmetic operations.

Run the changed program to ensure that the changed program reproduces the output of the original.

2.33.4 Exercise 4

Write a program, `time_bit_dec_ops.c` to measure the performance of the two programs used in the previous exercise (`dec_versus_bit_ops.c` and `dec_ops.c`).

The program should produce a table showing the *cpu time* taken by repeated iterations of the called functions for 100, 1,000, 10,000, 100,000 and 1,000,000 times.

The following should serve as a template for program output:

```
# times called                          mod 8      mod 8 bit        mult_33    mult_
 ↪33_bit          pwr_2      pwr_2_bit

1000                                 0.000000       0.000000       0.000000       0.
 ↪000000       0.000000       0.000000
100000                               0.000000       0.000000       0.000000       0.
 ↪000000       0.000000       0.000000
10000000                             0.015625       0.031250       0.031250       0.
 ↪015625       0.078125       0.031250
1000000000                           2.359375       2.343750       2.421875       2.
 ↪500000       7.578125       2.484375
```

Use **function pointers** to pass a function to another function that executes the passed function pointer the specified number of times.

2.33.5 Exercise 5

The program `linked_list.c` in `starter_code/lab02.c` contains an implementation of a *singly linked list* and a `main` function that manipulates the list. When executed, the output is:

```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10)  ]

Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items:
[  ]

Restored List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10)  ]

Element found: (4,1)
List after deleting an item:
[ (6,56) (5,40) (3,30) (2,20) (1,10)  ]

Element not found.
List after sorting the data:
```

(continues on next page)

(continued from previous page)

```
[ (1,10) (2,20) (3,30) (5,40) (6,56)  ]
List after reversing the data:
[ (6,56) (5,40) (3,30) (2,20) (1,10)  ]
```

Build the type defined in the program as a *separate module*. Create another program, `use_linked_list.c` that has the same `main` function in the program `linked_list.c` to call the functions that operate on the list.

The output from executing `use_linked_list.c` should be the same as above.

The student will need to make changes to code in the `main` method and `linked_list.c` to **encapsulate the `node` data structure**.

2.33.6 Exercise 6

Write unit tests using `check` to test the three functions in the program `dec_ops.c`; these are the functions used in *exercise 3* above.

In particular, write a *Suite Runner* that runs a *Test Suite* referencing one or more *Test Cases* that reference several *Unit Tests* containing *one assertion*.

The Unit Tests should test if each method produces a correct value and reports on incorrect values.

The starter code folder has a folder `dec_opstest` that contains:

- A `src` folder
  - The `src` folder contains the functions in `dec_ops.c` to be tested; there is no `main` function in `dec_ops.c` in the `src` folder.
- A `tests` folder
  - The `tests` folder is where the student will code the *Test Suite* referencing one or more *Test Cases* that reference several *Unit Tests* containing *one assertion*.
- A `makefile`
  - This `makefile` is similar to the ones used in the lesson on *Unit testing*.

The output should resemble:

```
$ make all
gcc -c -Wall  src/*.c
gcc -c -Wall   tests/*.c
gcc dec_ops.o check_dec_ops.o -lcheck -lm -lpthread -lrt -lsubunit  -o check_dec_ops_
 ↪tests

$ make test
./check_dec_ops_tests
Running suite(s): dec_ops
50%: Checks: 6, Failures: 3, Errors: 0
tests/check_dec_ops.c:11:F:mod_8_fail:mod_8_fail:0: Assertion 'mod_8(7) == 0' failed:␣
 ↪mod_8(7) == 7, 0 == 0
tests/check_dec_ops.c:20:F:mult_33_fail:mult_33_fail:0: Assertion 'mult_33(3) == 70'␣
 ↪failed: mult_33(3) == 99, 70 == 70
tests/check_dec_ops.c:28:F:pwr_2_fail:pwr_2_fail:0: Assertion 'pwr_2(60) == 1' failed:
 ↪ pwr_2(60) == 0, 1 == 1
Makefile:29: recipe for target 'test' failed
make: *** [test] Error 1
```

Note there are *six* tests; *three* passed and *three* failed.

This chapter provides a review of the previous five lessons in preparation for the upcoming written exam.

- Unit testing
- C program design
- Bit manipulation
- Library functions
- Profiling C programs

2.34 Unit testing

In C, unit testing is generally more difficult than in other languages.

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.

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.

```
check: test/test_all
        ./$^

# All unit test code
test/test_all: test/test_all.o test/test_atl.o test/test_yyg.o
# Modules under test
test/test_all: atl.o yyg.o
# Extra Library for Check
test/test_all: LDLIBS += -lcheck
```

- Check Organization

With Check, there are five key pieces to writing tests:

Suite Runner: Runs *Test Suites*; usually one Suite Runner.*Test Suites*: High-level grouping of *Test Cases*; usually one per compilation unit.*Test Cases*: Low-level grouping of Unit Tests; get added to a Test Case.*Unit Tests*: Series of *Assertions* that the system under test is working correctly.*Assertions*: Simple true-or-false statements; if the assertion fails, that indicates a problem with the system under test.

- Suite Runner

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

`test_all.c`.

```
extern Suite *test_yyg_suite(void);
extern Suite *test_atl_suite(void);
```

(continues on next page)

(continued from previous page)

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

        srunner_add_suite(sr, test_yyg_suite());
        srunner_add_suite(sr, test_atl_suite());
        // Add other test suites if needed
        ...

        srunner_run_all(sr, CK_NORMAL);
        int failed = srunner_ntests_failed(sr);
        srunner_free(sr);

        return !!failed;
}
```

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 second argument to `srunner_run_all` configures to no output, error-only output, or all tests. Most of the time, the `CK_NORMAL` level is appropriate.

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

The test program should return an appropriate value from `main` as to whether the tests succeeded or not. Check the number of tests that failed with `srunner_ntests_failed`: 0 is success and should return that from `main`, any other number is failure.

The unusual construct `!!failed` has an important *side effect*; `!!failed` coerces the value of `failed` to be either `0` or `1`.

- Test Suites

A Test Suite should collate and gather all the Test Cases for that compilation unit. a Test Suite is generally coded in separate compilation units.

`test_yyg.c`.

```
Suite *test_yyg_suite(void)
{
        Suite *s = suite_create("YYG");
        TFun *curr = NULL;

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

        // More 'tcase_create' and `suite_add_tcase` calls follow for all cases added␣
 ↪to the suite

        return s;
}
```

The main export of a compilation unit's worth of tests needs to be its Test Suite. Code a function (`test_yyg.c` in this example) that builds and returns the Test Suite. Create and add Test Cases to the Test Suite with `suite_add_tcase`.

- Test Cases

Each Test Case has a name that it is created with, using the `tcase_create` function.

A Test Case is composed of multiple Unit Tests, individual functions of type `TFun{nbsp}*`. 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 *core_tests[] = {
        test_yyg_create,
        test_yyg_put_down,
        // Other test functions
        NULL
};

Suite *test_yyg_suite(void)
{
        Suite *s = suite_create("YYG");
        TFun *curr = NULL;

        curr = core_tests;
        while (*curr) {
                tcase_add_test(tc_core, *curr++);
        }
        suite_add_tcase(s, tc_core);

        ...

        return s;
}
```

- 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_atl.c.

```
static airport *atl;

static void in_setup(void)
{
        atl = airport_create("ATL");
}

static void in_teardown(void)
{
        airport_destroy(atl);
}

Suite *test_atl_suite(void)
{
        Suite *s = suite_create("ATL");
        TFun *curr = NULL;

        TCase *tc_input = tcase_create("input");
        tcase_add_checked_fixture(tc_input, in_setup, in_teardown); // Register␣
 ↪fixture functions
        curr = input_tests;
        while (*curr) {
```

(continues on next page)

(continued from previous page)

```
                tcase_add_test(tc_input, *curr++);
        }
        suite_add_tcase(s, tc_input);

        return s;
}
```

- Unit Test

A Unit Test is the set of steps it takes to ascertain correct functionality. The Unit Test is set up and ended with a set of **Check macros**. `START_TEST()` takes an argument that will be the name of the function.

```
START_TEST(test_name_constructor)
{
        ck_assert_str_eq(atl->name, "Atlanta");
}
END_TEST
```

Every Unit Test will have a number of Check Assertions inside its function body.

- Test Loops

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 square_data[][2] = {
        {0, 0},
        {1, 1},
        {2, 4},
        {-2, 4},
        {9, 81}
};
enum { SQUARE_DATA_SZ = sizeof(square_data)/sizeof(square_data[0]) };

START_TEST(test_square)
{
        ck_assert_int_eq(square(square_data[_i][0]), square_data[_i][1]);
}
END_TEST

int main(void)
{
        ...
        tcase_add_loop_test(tc_core, test_square, 0, SQUARE_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`.

For more complex tests, it can be useful to define a `struct` that tracks the data members.

- 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`, `ck_assert_uint_eq` Confirms that two signed/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()`

`assert` tests 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.

The programmer must ensure that the calls to `assert` **do not produce side effects**.

When the symbol `NDEBUG` ("`not debugging`") is set, all `assert` lines are stripped out.

2.35 C Program Design

With an understanding of pointers and unit tests, more attention can now be given to designing programs in C. 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.

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

The compiler treats the new variable as the original type declaration. 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.

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.

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, known as "`opaque types`".

Consider a dynamic array of words. These fields are tightly coupled and generally should *only* be manipulated by code that understands their relationship.

```
struct wordlist_ {
        char **words;
        size_t size;
        size_t capacity;
};
```

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

```
typedef struct wordlist_ wordlist;
```

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; 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`. Function declarations will only use pointers to objects of the new `typedef`.

`wordlist.h`.

```
#ifndef WORDLIST_H
#define WORDLIST_H

#include <stdbool.h>

typedef struct wordlist_ wordlist;

wordlist *wordlist_create(void);
bool wordlist_append(wordlist *wl, const char *s);
void wordlist_print(const wordlist *wl);
void wordlist_destroy(wordlist *wl);
#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.

```
int main(void)
{
        wordlist *words = wordlist_create();
        if (!words) {
                perror("Could not create word list");
                return 1;
        }

        // Check if append was successful
        if (!wordlist_append(words, "Able"))    {// Do something }
        if (!wordlist_append(words, "Baker"))   {// Do something }
        if (!wordlist_append(words, "Charlie")) {// Do something }

        wordlist_print(words);

        wordlist_destroy(words);
}
```

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

`wordlist.c`.

```
#include "wordlist.h"

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

struct wordlist_ {
        char **words;
        size_t size;
        size_t capacity;
};

wordlist *wordlist_create(void)
```

(continues on next page)

(continued from previous page)

```
{
        wordlist *wl = malloc(sizeof(*wl));
        if (!wl) {
                return NULL;
        }
        wl->size = 0;
        ...
}

bool wordlist_append(wordlist *wl, const char *s)
...

void wordlist_print(const wordlist *wl)
...

void wordlist_destroy(wordlist *wl)
...
```

The implementation file should always have the corresponding header as the very first `#include`, before any others.

- APIs

Designing APIs can be challenging. In C, extra care is needed to produce a useful API.

Identify any opaque types, ``struct``s, constants, and 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.

- 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.

Methods should begin with a consistent name prefix; this means `wordlist_append` instead of `append_wordlist`.

- ``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`. 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 { WL_SUCCESS, WL_OUT_OF_MEMORY, WL_NO_PERMISSION };

enum { WL_DEFAULT_CAPACITY=8 };
```

- 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.

- Interface vs. Implementation

Keep the interface of a module, its header file, as small as possible.

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.

Find any functions in the public interface that should be made private, such as helper functions. Code these methods in the implementation file only, and made `static`.

- Documentation

The public interface of a module, its header file, is the place for public documentation.

This *definitely* is not to say that comments can be left out of the implementation file. Comments in the *implementation file* are for the authors and maintainers of the code. Comments in the *interface file* are for users of the module.

2.35.1 `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`. The linker will only use that function locally, in the current compilation unit.

This prevents name conflicts at link time.

2.36 Bit manipulation

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

- Bitwise operators

C contains *bitwise operators* specifically designed to operate on the individual bits os a number.

- Bitwise operators in compound assignment statements

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

The expressions:

```
a &= b              a = a & b
a = a << 2          a =<<2
```

are equivalent

Worthy of note is that *adding one to the compliment of a number negates the number*, known as *two's compliment*.

- Bitfields

C allows for *bitfields*, which are fields that may occupy less than a byte of storage. 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, so 5 bits
    // are sufficient
    unsigned int day : 5;

    // month has value between 1 and 12, so 4 bits
    // are sufficient
    unsigned int month : 4;

        // Structs can contain non-bitfield members as well as bitfields
    unsigned int year;
};
```

Bitfields must be large enough to hold assigned values. Also, using **signed int types** rather than `unsigned` may cause issues.

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.

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.*

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

Most platforms are *little endian*.

When performing I/O on the same platform the byte order is not that important, but once more than one machine (or the network) is involved, the program may need to convert to and from big- and little-endian formats.

There are no builtin functions that convert big to little or vice-versa for **all data types**.

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

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

These functions are available in the `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`.

- 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.

`struct` instances may be written/read in binary. The following statements provide a template for the write/read operations:

```
// fp is a file pointer, an_emp is an instance of a struct
num_written = fwrite(an_emp, sizeof(*an_emp), 1, fp);

num_read = fread(an_emp, sizeof(*an_emp), 1, fp);
```

The programmer may check the value of `num_written` and `num_read` to verify the write/read operation was successful.

2.37 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`.

- 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 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
```

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.

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.

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.

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`

`getopt` allows for more sophisticated command line parsing than using functions in the `string.h` header.

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.

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`.

Iterate over the command line arguments, comparing the options to those coded for `getopt` processing with a statement similar to:

```
while((opt = getopt(argc, argv, "irx")) != -1)
```

This call to `getopt` searches the command line arguments for `-i -r -x` in any order.

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 accessible.

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.

Set the `opterr` state variable `0` to *suppress diagnostics from* `getopt`.

For invalid options, `getopt` prints:

```
./myprogram: invalid option -- 't'
```

By assigning `0` to the state variable `opterr`, a program can handle unknown options in its own manner.

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.

`getopt` emits a diagnostic if an option requiring a parameter is not supplied one.

This call to `getopt` expects to have a value immediately following the `f` option:

```
while((opt = getopt(argc, argv, "irf:x")) != -1)
```

- `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.

- `...`, `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, ...);
```

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:

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

int add_em_up (int count,...)
{
  va_list ap;                    /* ap represents the arguments passed to program */
  int i, sum;

  va_start(ap, count);          /* Initialize the argument list. */

  sum = 0;
  for (i = 0; i < count; i++)
    sum += va_arg (ap, int);     /* Get the next argument value. */
```

(continues on next page)

(continued from previous page)

```
  va_end(ap);                  /* Clean up. */
  return sum;
}

int main (void)
{
  /* This call prints 16. */
  printf("%d\n", add_em_up (3, 5, 5, 6));

  /* This call prints 55. */
  printf("%d\n", add_em_up (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
}
```

which outputs:

```
16
55
```

Type information for arguments is *not passed*.

There is *no type information* passed with the varying arguments. The variadic function must know the types of the arguments.

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`.

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);
        //

        {
```

(continues on next page)

(continued from previous page)

```
                // 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).

`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).

`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             */
};
```

`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 */
};
```

- `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)
```

- `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.

Fetch a time using the `time` function. Pass the value returned by `time` to `localtime`.

```
time_t rawtime;
struct tm *info;

time(&rawtime);
info = localtime(&rawtime);
```

Access the date and time fields by dereferencing the `info` pointer.

- `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.

Fetch a time using the `time` function. Pass the value returned by `time` to `gmtime`.

```
time_t rawtime;
struct tm *info;

time(&rawtime);
/* Get GMT time */
info = gmtime(&rawtime);
```

- `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*.

```
struct timespec start;
// Critical section, the part being measured
timespec_get(&start, TIME_UTC);
```

Access the seconds and nanoseconds from the `struct timespec` object.

2.38 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 user's perception of program speed is important. 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.

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;         // Time in seconds
        long nanoseconds = end.tv_nsec - start.tv_nsec;     // Time in nanos
        if (nanoseconds < 0) {
                seconds -= 1;
                nanoseconds += 1000000000;
        }

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

`struct{nbsp}timespec` is a data structure designed to represent an interval.

```
struct timespec {
        time_t tv_sec;      // number of seconds since Jan 1, 1970
        long tv_nsec;       // number of nanoseconds for the current second
};
```

- 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 by compiling and linking a *profiling build* of a program or library.

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.

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

- `gprof`

When a program with profiling symbols is run, it will generate a file `gmon.out`, the *call graph profile*, in the current working directory of the program. It tracks when functions are called, and how much time is taken inside the function.

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, which is omitted in output examples.

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

In the sample above, the vast majority of the time is spent in `path_find`. 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]
```

(continues on next page)

(continued from previous page)

```
[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.

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. The `callgrind` tool (a plugin for `valgrind`) generates a call graph similar to `gprof`. While slightly slower than `gprof`, `callgrind` tends to be more accurate in its measurements than `gprof`.

```
$ 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.

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

--------------------------------------------------------------------------------
Ir          file:function
--------------------------------------------------------------------------------
18,421,283  maze.c:path_find [/home/wechlin/x/maze/drich/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/wechlin/x/maze/drich/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, which helps trach down bottlenecks from library programs. 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 instruction counts, 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; some assignments took 3 instructions, some take 2 instructions.

Consult the documentation for more information on additional `callgrind` functionality, such as L1 and L2 cache reads and writes.

- Optimization

There are **three major rules** when it comes to optimization.

1. Don't Optimize.
2. (For experts only!) Don't Optimize Yet.
3. Measure (profile) before optimizing.

Taking these in reverse, the first step is to measure what a program is doing by *profiling the program*.

Look at the total time spent in a function and the number of times functions are called.

The best optimization is *running less code*.

When it comes to optimizing, there is a hierarchy of how effective (in cost/time) certain categories of optimization happen to be.

Buy faster hardwareOrganize program data efficientlyUse a better algorithmEngage in micro-optimizations

- 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 the these optimizations is, to some extent, sacrificing readability and clarity for speed.

Loop unrolling is expanding some of the loop within the loop and decreasing the conditional expression(s) that govern loop execution.

Loop Rolled Up.

```
for (size_t n=0; n < 100; ++n) {
        execute_function(n);
}
```

In this first case, the conditional is evaluated 101 times. The first 100 times, the conditional is true and the loop body is executed. On the 101st time, the conditional is false and the loop exits.

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);
}
```

`switch` over multiple `if` statements because `switch` statements evaluate the condition *once* for each execution, whereas a series of multiple `if` statements may evaluate the conditional *several times* for each execution.

Preincrement over Postincrement, because postincrement operations may require 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 new value.

The general rule would be to always prefer preincrement, and resort to postincrement only if needed by the expression.

Todo: Module I - Validate files and correct image references.
