C Review

Table of Contents

Data types

Integer

- `int`: 2 or 4 bytes (platform dependent)
- `char`: 1 byte
- `short`: 2 bytes
- `long`: 4 bytes
- corresponding `unsigned` types for non-negative numbers

- e.g. `int` may store -32768 to 32767
  - `unsigned int` stores integers from 0 to 65535

Floating point numbers

- `float`
- `double`

`char` s and strings

- `char` stores a single ASCII character
- Strings: arrays of chars terminated by a null byte (`'\0'`)
  - e.g. "Hello world!" is stored as the array of characters: ['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!', '\0']'

Boolean values

- no built-in boolean type, integers can be used
- non-zero values: true
- 0: false
- C99 with `stdbool.h` provides `bool` data type with `true` and `false`

Function declarations

- place function prototype declarations at top of file as good practice so you don't need to worry about ordering of functions in file

```
// prototype (at top of file)
return_type function_name(arg_type arg_name);

// function implementation
return_type function_name(arg_type arg_name) {
    return ret_value;
}
```

`main` Function

- when a C program is run from command line, `main` function is executed
- `argc`: argument counter; number of arguments supplied
- `argv`: argument vector; array of argument strings
- return value: indicates success (0) or failure (non-zero) of program

Program to print the number of arguments and what they are:

```c
int main(int argc, char **argv) {
    int i;

    printf("Number of arguments: %d\n", argc);
    for (i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Compilation

To compile hello.c

```
$ gcc -Wall -pedantic -o hello hello.c
```

- `-Wall`: warnings all; highest level compiler warnings turned on
- `-pedantic`: enables another set of compiler errors
- `-o <file_name>`: output program should be called `<file_name>`
- `<source>.c`: source file
- for debugging, compile with `-g` to access source code/variable names/function names from inside debuggers e.g. `gdb`, `lldb`

Preprocessor directives

- keywords that start with # e.g. `#define, #include`
- these are evaluated prior to compilation by the preprocessor, which effectively copy and pastes the definition/included function definition into the code

Library functions

Standard library header files imported using `#include` preprocessor directive

```c
#include <assert.h> // contains assert, frequently used to verify malloc
#include <math.h>   // math functions e.g. cos, sin, log, sqrt, ceil, floor
#include <stdio.h>  // input/output e.g. printf, scanf
#include <stdlib.h> // contains NULL, memory allocation e.g. malloc, free

int main(int argc, char **argv) {
    /* ... */
    return 0;
}
```

Pointers

- pointers are memory addresses
- we can have types which hold memory addresses to integers and floats using an asterisk
- `int *my_ptr`: contains address of an int
- `int **`: pointer to a pointer; address of an address to an integer
- `&foo`: memory address/pointer to `foo`; "address of foo"
- `*bar`: access data stored at pointer `bar`; "data stored at bar"
- pointer arithmetic: pointer type knows which data type it points to, and therefore knows the size. If `int *my_ptr` is a pointer to the start of an array of integers, you can jump forward the size of an `int` with `my_ptr+1`

Arrays

- creating a static array: `int my_array[100];` to create an array with room for 100 integers
- `my_array[7]` to access the 8th element of the array
- arrays in C are simply pointers to the first element of the array, so:
  - my_array[10]` ⟺ `*(my_array + 10)
  - &my_array[10]` ⟺ `my_array + 10
- explicit definition of static array: `int arr[] = {1, 2, 3, 4, 5};`
- tip: always use pointer notation for data types (in function definitions etc.) i.e.

```c
// preferred
int get_length(int *array) {
    /* ... */
```

```c
    return length;
}
// not recommended
int get_length(int array[]) {
    /* ... */
    return length;
}
```

Structs

- encapsulate multiple pieces of data e.g. student record

```c
typedef struct student Student;
struct student {
    char *first_name;
    char *last_name;
    int id;
    float mark;
}
```

- here we created a struct `student` which can be referred to with `struct student`
- syntactic sugar: `typedef` this to `Student`, such that `Student` is an alias for `struct student`
- an alternative that avoids the intermediate name is:

```c
typedef struct {
    char *first_name;
    char *last_name;
    int id;
    float mark;
} Student;
```

- this doesn't allow you to reference the struct within the definition e.g. nodes for a linked list/-graph:

```c
typedef struct node Node;
struct node {
    int data;
    Node *next;
}
```

Accessing fields

```c
Student matthew;
// dot notation
```

```
matthew.student_number = 123456;

Student *james = malloc(sizeof(*james));
assert(james);
// arrow notation
james->student = 654321;
free(james);
james = NULL;
```

- foo.bar` $\iff$ `(&foo)->bar
- foo->bar` $\iff$ `(*foo).bar

Dynamic Memory Allocation

- variables declared inside a function are usually stored on the *stack*
- function's local variables and function parameters exist in a *stack frame* specific to the function
  - stack frame only lasts as long as the function is running
  - once the function returns the local variables/function parameters are de-allocated
  - size of variables needs to be known at compile time
- `malloc` requests specific amount of memory on the *heap* which exists until we explicitly *`free`* it
- memory allocated at runtime, and may fail e.g. program already has used full allowance of memory OS has reserved for it
- use `assert` to check the pointer is not NULL i.e. has been successfully allocated
- `malloc` returns a void pointer

```
void *malloc(size_t size)    // size: size of memory block [bytes]
```

Example: allocating memory for an int

```
int *my_int = malloc(sizeof(*my_int)); // cast to (int *)
assert(my_int);      // check pointer is not null, i.e. malloc succeeded
/* do stuff */
free(my_int);        // free the memory
my_int = NULL;       // ensure that we don't inadvertently access freed memory
```

Variable-sized array

- arrays are pointers to first element in the array, so you can use `malloc` to allocate a variable sized array. For `n` items you can allocate a block with enough space for `n` adjacent items:

```
int n = 10000;
double *array = malloc(sizeof(*array) * n);
/* magic happens here */
free(array);
array = NULL;
```

Header Files

- *modules* are used to separate out code into related groups. Consists of:
  - `module.h`: consists of a header file, containing:
    * info on how to use the module,
    * function prototypes
    * type definitions
  - `module.c`: file containing implementations
- `#include "module.h"` is then used to access the definitions

Import guards

- C doesn't allow you to declare things more than once
- good practice: use *if guards* to prevent a `.h` file being included more than once
- define a macro per header file, and only declare anything if it hasn't been defined yet

e.g. to write a hello world module

`hello.h`:

```
// import guard
#ifndef HELLO_H
#define HELLO_H

// print "hello, {name}!" on a line
void hello(char *name);
#endif
```

`hello.c`:

```c
#include <stdio.h>
#include "hello.h"

// print "hello, {name}!" on a line
void hello(char *name) {
  printf("Hello, %s!\n", name);
}
```

main.c

````
#include "hello.h"

int main(int argc, char **argv) {
    char *name = "Barney";
    hello(name);
    return 0;
}

To compile a program with multiple `.c` files:
```console
$ gcc -o <executable name> <list of .c files>
````

For this example

```
$ gcc -o main main.c hello.c
```

Makefiles

`make` keeps track of changes across various files, only compiles what needs to be recompiled when something changes - example Makefile for compiling C programs

```makefile
# # # # # # #
# Sample Makefile for compiling a simple multi-module C program
#
# created for COMP20007 Design of Algorithms 2017
# by Matt Farrugia <matt.farrugia@unimelb.edu.au>
#

# Welcome to this sample Makefile. If you're new to make and makefiles,
    have a
# read through with the comments and follow their instructions.


# VARIABLES - change the values here to match your project setup

# specifying the C Compiler and Compiler Flags for make to use
CC     = gcc
CFLAGS = -Wall
```
```

```

# exe name and a list of object files that make up the program
EXE     = main-2
OBJ     = main-2.o list.o stack.o queue.o


# RULES - these tell make when and how to recompile parts of the project

# the first rule runs by default when you run 'make' ('make rule' for others)
# in our case, we probably want to build the whole project by default, so we
# make our first rule have the executable as its target
#   |
#   v
$(EXE): $(OBJ) # <-- the target is followed by a list of prerequisites
	$(CC) $(CFLAGS) -o $(EXE) $(OBJ)
# ^
# and a TAB character, then a shell command (or possibly multiple, 1 line each)
# (it's very important to use a TAB here because that's what make is expecting)

# the way it works is: if any of the prerequisites are missing or need to be
# recompiled, make will sort that out and then run the shell command to refresh
# this target too

# so our first rule says that the executable depends on all of the object files,
# and if any of the object files need to be updated (or created), we should do
# that and then link the executable using the command given


# okay here's another rule, this time to help make create object files
list.o: list.c list.h
	$(CC) $(CFLAGS) -c list.c

# this time the target is list.o. its prerequisites are list.c and list.h, and
# the command (its 'recipe') is the command for compiling (but not linking)
# a .c file

# list.c and list.h don't get their own rules, so make will just check if the
# files of those names have been updated since list.o was last modified, and
```

```
# re-run the command if they have been changed.


# actually, we don't need to provide all that detail! make knows how to compile
# .c files into .o files, and it also knows that .o files depend on their .c
# files. so, it assumes these rules implicitly (unless we overwrite them as
# above).

# so for the rest of the rules, we can just focus on the prerequisites!
# for example stack.o needs to be rebuilt if our list module changes, and
# also if stack.h changes (stack.c is an assumed prerequisite, but not stack.h)
stack.o: stack.h list.h

# note: we only depend on list.h, not also list.c. if something changes inside
# list.c, but list.h remains the same, then stack.o doesn't need to be rebuilt,
# because the way that list.o and stack.o are to be linked together will remain
# the same (as per list.h)

# likewise, queue.o depends on queue.h and the list module
queue.o: queue.h list.h

# so in the future we could save a lot of space and just write these rules:
# $(EXE): $(OBJ)
#    $(CC) $(CFLAGS) -o $(EXE) $(OBJ)
# list.o: list.h
# stack.o: stack.h list.h
# queue.o: queue.h list.h



# finally, this last rule is a common convention, and a real nice-to-have
# it's a special target that doesn't represent a file (a 'phony' target) and
# just serves as an easy way to clean up the directory by removing all .o files
# and the executable, for a fresh start

# it can be accessed by specifying this target directly: 'make clean'
clean:
    rm -f $(OBJ) $(EXE)
```