2019 Semester 1
COMP20007 Design of Algorithms
School of Computing and Information Systems
The University of Melbourne

C Programming Refresher

Knowledge of C is a pre-requisite for this subject so this document isn't meant as an all-encompassing introduction to C, rather a refresher for those who may not have worked in C for a semester or two.

Students who completed *COMP10002 Foundations of Computing* should have seen dynamic memory allocation using `malloc`.

Also required will be the use of header files (`.h`), a brief introduction is provided here.

If any of this is new we strongly recommend *Programming, Problem Solving and Abstraction with C* by *A. Moffat* as a good reference for the C programming required for successful completion of this subject.

1 Data Types

The integer data types in C are `int`, `char`, `short` and `long`. `char`, `short` and `long` are at least 1, 2 and 4 bytes respectively. The size of an `int` is platform dependent but is usually 2 or 4 bytes.

There are corresponding `unsigned` types for each of these integer data types for non-negative numbers. For example an `int` may be able to store the integers in the range -32768 to 32767, in which case an `unsigned int` can store integers from 0 to 65535.

For floating point numbers C has `float` and `double`.

A `char` can store a single ASCII character. Strings in C are arrays of `char`'s terminated by a null byte, for instance `"comp20007"` is stored as the array of characters `['c','o','m','p','2','0','0','0','7','\0']`.

Here's an example of some variable declarations of different data types:

```c
/* Integer data types */
int my_negative_number = -20007;
unsigned int my_unsigned_number = 20007;

/* Floating point numbers */
float my_float = 0.123;
double my_double = 0.123;

/* Declaring a char using a number or a character in single quotes */
char my_char = 100;
char my_other_char = 'A';

/* We can declare string literals like this: */
char my_str[5] = {'c', 'o', 'm', 'p', '\0'};
```

```
/* But we'd usually do it like this: */
char my_str[] = "comp";
```

Out of the box C doesn't have a boolean type, and integers can be used. Non-zero values are interpreted as true while 0 is interpreted as false. For example:

```
int my_true_bool = 1;
int my_false_bool = 0;

if (my_true_bool) {
  printf("Algorithms are fun!\n");
}

if (my_false_bool) {
  printf("Algorithms are not fun\n");
}
```

will print "Algorithms are fun!".

If you'd prefer (and you're using C99) the `stdbool.h` header provides the `bool` data type and the values `true` and `false`.

2 Function Syntax

When we define a function in C we must declare the type of the value returned (if there is one) and the types of each of the function arguments. To return a value from a function we use the `return` keyword. An example of a function which takes an integer and returns an integer (used as a bool) is given.

```
int is_fun(int subject_code) {
  if (subject_code == 20007) {
    return 1;
  } else {
    return 0;
  }
}
```

To declare a function which does not return a value we use the `void` keyword:

```
void returns_nothing(int subject_code) {
  if (subject_code == 20007) {
    printf("Fun!\n");
  }
}
```

If a function takes no arguments the parentheses after the function name can be left empty:

```
void takes_no_arguments() {
  printf("Fun!\n");
}
```

In C you must declare a function before it's used. To get around having to clever about the order of your function implementations it's good practice to provide function prototype declarations towards the top of the file before the function implementations. An example is provided.

```c
/* Function Prototype */
int my_function(int n);

int main(int argc, char **argv) {
  int n = 100;
  int result = my_function(n);
  printf("%d\n", result);
  return 0;
}

/* Function Implementation */
int my_function(int n) {
  return n * n;
}
```

3 Main Function

When a C program is run from the command line the `main` function is executed. The `main` function takes the number of provided arguments (`argc`) and an array of argument strings (`argv` – for "argument vector"). The return value is used to indicate success or failure of the program. A return value of zero indicates success and a non-zero return value indicates failure.

Note that it's standard for the type of `argv` to be `char **`, as it is a pointer to an array of strings.

The example below prints the number of arguments and what those arguments 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;
}
```

If this is compiled into a program called `myprogram` then an example of this program's execution might be:

```
$ ./myprogram
Number of arguments: 1
./myprogram
$ ./myprogram arg1 arg2
Number of arguments: 3
./myprogram
arg1
arg2
```

4 Compilation

The command I use to compile a C program `hello.c` is

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

- `gcc` is the name of the C compiler we're using.
- `-Wall` stands for "Warnings All" and it turns on the highest level compiler warnings. In general having more compiler warnings is good since catching an error at compile time is better than having a hard to catch runtime error.
- `-pedantic` turns on another set of compiler errors.
- `-o hello` tells the compiler that the output program should be called `hello`.
- `hello.c` is the C source code file.

A debugging tip is that if you use the `-g` flag you'll be able to access the source code/variable names/function names *etc.* from inside debuggers such as `gdb` and `lldb`.

5 Library Functions

You can import standard library header files using the `#include` preprocessor directive[1], for instance:

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

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

Some of the common header files which are included are,

- stdio.h – contains functions related to input/output such as `printf` and `scanf`.
- stdlib.h – contains the definition of `NULL`, and subjects related to memory allocation such as `malloc` and `free`.
- assert.h – contains the definition of `assert`, which is useful to confirm statements which should be true at certain points in your program. One common place `assert` is used is after memory allocation, to ensure that the memory has indeed been allocated.
- math.h – contains maths functions such as `sin`, `cos`, `log`, `sqrt`, `ceil`, `floor` *etc.*

6 Pointers

In C we're able to explicitly access positions in memory where data is stored. This is achieved using *pointers*. While `int` and `float` define the type of a variable which contains and `int` or a `float` respectively, we can have have types which hold memory addresses to integers and floats by using an asterisk. For example `int *my_ptr` will contain the address of an int. To get a pointer to a pointer we can use two asterisks: a variable of type `int **` will contain an address of an address to an integer.

If we have a variable `foo`, we can get the pointer to that variable using `&foo`. If we have a pointer `bar` we can access the data stored at that memory address using `*foo`.

[1] Preprocessor directives are the keywords that start with the # symbol, such as `#define` and `#include`. These are evaluated before any compilation is done, for instance wherever the preprocessor sees something that has been `#define`d it will essentially copy paste the definition into the code. Similarly with `#include`s the preprocessor can be thought of as copy pasting the function definitions from the included header file.

I like to think of `&foo` translating to "address of `foo`", and `*bar` translating to "data stored at `bar`".

Since a pointer type knows which data type it is pointing to, it also knows how large that data is. This allows us to do pointer arithmetic. For example if `int *my_ptr` is a pointer to the start of an array of integers we can jump forward the size of an `int` in memory by using `my_ptr + 1`. More on this in the next section.

7 Arrays

If we know at compilation time how large an array is (*i.e.,* it's size is a constant) we can create one like so: `int my_array[100];` This will create an array which has room for 100 integers. To access an element, you can use the `my_array[7]` syntax. Similarly to set the value of an element we can do `my_array[7] = 200;`

Arrays in C are just pointers to the first element of the array (!!!), hence the following are equivalent:

$$\texttt{my\_array[10]} \iff \texttt{*(my\_array + 10)}$$
$$\texttt{\&my\_array[10]} \iff \texttt{my\_array + 10}$$

You can also explicitly define the elements of an array like so: `int arr[] = {2, 0, 0, 0, 7};`

To make my life easier when I'm dealing with arrays I always use the pointer notation for the data types (*i.e.*, in function definitions *etc.*), so I would use

```
int get_length(int *array) {
  /* ... */
  return length;
}
```

instead of

```
int get_length(int array[]) {
  /* ... */
  return length;
}
```

8 Structs

In C we use *structs* to encapsulate multiple pieces of data. For instance – let's say we want to represent a student's name, student ID and mark in some subject – rather than having to keep 3 or 4 variables around we can declare a struct which encapsulates this information like so:

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

When defining this struct we've created a struct called `student` which can be referred to with `struct student`. To make our lives easier we typedef this to `Student`, so "`Student`" is just an alias for "`struct student`".

Another pattern for declaring structs you might see is having the typedef and struct in the same statement like so:

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

The idea here is that we avoid giving an intermediate name to the struct. I prefer to use the first of the two options as it means you can refer to the struct within it's own definition. For instance if we're defining a node (in a linked list, or graph for example) we can do the following:

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

To access the fields in a struct we can use the dot notation, or if we have a pointer to the struct we can use the arrow notation, like so:

```
Student matthew;
matthew.student_number = 123456; /* Dot notation */

/* We'll cover malloc in the next section */
Student *james = malloc(sizeof(*james));
assert(james);

james->student_number = 654321; /* Arrow notation */

free(james);
james = NULL;
```

Note that `foo.bar` and `(&foo)->bar` are synonymous, as are `foo->bar` and `(*foo).bar`.

9 Dynamic Memory Allocation

When a variable is declared inside a function in C it is usually stored on what's called the *stack*. A function's local variables and function parameters exist in a *stack frame* specific to the function, and this stack frame lives only as long as the function is running. Once the function returns or ends the memory associated with the local variables and function parameters are de-allocated.

Additionally, the size of these variables is required to be known at compile time. For instance the compiler knows exactly how much memory is needed to represent the following:

- `int my_int;`
- `float *my_float_ptr;`
- `MyStruct my_struct;`
- `int my_const_size_array[100];`

Alternatively we can, as programmers, ask for a specific amount of memory using `malloc`. This memory is allocated on what is called the *heap* and rather than living and dying with the function, it will exist until we explicitly `free` the memory.

This memory is allocated at runtime, which means that there's a chance that it could fail. An example of a situation where this might arise would be if the program has already allocated the total amount of memory the operating system has reserved for it. One way to ensure that the result of `malloc` has succeeded is to follow it with an `assert` to assert that the pointer is not null.

For example we could allocate memory for an `int` like so:

```
/* The return value of malloc is a (void *) which we cast to an (int *) */
/* This sizeof(*my_int) will do the same as sizeof(int) */
int *my_int = malloc(sizeof(*my_int));

/* Assert that the pointer is not null, i.e., the malloc succeeded */
assert(my_int);

/* ... Do something with my_int ... */

/* Free the memory now that we're done */
free(my_int);

/* Set my_int = NULL so that we don't inadvertently access freed memory */
my_int = NULL;
```

Since arrays are just pointers to the first element in an array we can use `malloc` to allocate space for a variable sized array. For n items we can do this by simply allocating enough space for n items next to each other:

```
int n = 20007;
double *array = malloc(sizeof(*array) * n);
assert(array);

/* ... */

free(array);
array = NULL;
```

As we saw above this is also usually how we create a struct.

10 Header Files

To split up our C code into multiple files we can group related code into what are called *modules*. For instance we could create a module for linked lists and a module for queues.

A module consists of a header file `module.h` and a file containing implementations, `module.c`. `module.h` should contain information about how to use the module, that is function prototypes and type definitions. These functions should be implemented in the `module.c` file.

To access the definitions from `module.h` use the preprocessor directive `#include "module.h"` in any file using the definitions (including the `module.c` file).

C doesn't let us declare something more than once, so it's often good practice to use if guards to prevent a `.h` file from being included more than once. We do this by defining a macro per header file, and only declaring anything if it hasn't been defined yet.

For example if we wanted to write a module which provided hello world functionality we might create the following files.

`hello.h`:

```c
/* The import guard. ifndef runs if the symbol has not been defined. */
#ifndef HELLO_H
#define HELLO_H

/* Prints "Hello, {name}!" on it's own line. */
void hello(char *name);

#endif
```

`hello.c`:

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

/* Prints "Hello, {name}!" on it's own line. */
void hello(char *name) {
  printf("Hello, %s!\n", name);
}
```

`main.c`:

```c
#include "hello.h"

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

To compile a program which uses multiple `.c` files we can use a command like:

`gcc -o <executable_name> <list of .c files>`

So for the above example this would be `gcc -o main main.c hello.c`.

11 Makefiles

To automate the compilation of C programs (among other things) we can use the unix utility `make`, which will keep track of changes across various files and only compile what needs to be recompiled when something changes.

An example `Makefile` which provides detailed usage instructions is provided on the LMS.