COMP20007 Design of Algorithms

Week 3 Workshop

Tutorial

0. Sums Give closed form expressions for the following sums.

(a) $\sum_{i=1}^{n} 1$ (b) $\sum_{i=1}^{n} i$ (c) $\sum_{i=1}^{n}(2i+3)$

(d) $\sum_{i=0}^{n-1}\sum_{j=0}^{i} 1$ (e) $\sum_{i=1}^{n}\sum_{j=1}^{m} ij$ (f) $\sum_{k=0}^{n} x^k$

1. Complexity classes For each of the following pairs of functions, $f(n)$ and $g(n)$ determine whether $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$ or both (*i.e.*, $f(n) \in \Theta(g(n))$).

(a) $f(n) = \frac{1}{2}n^2$ and $g(n) = 3n$

(b) $f(n) = n^2 + n$ and $g(n) = 3n^2 + \log n$

(c) $f(n) = n \log n$ and $g(n) = \frac{n}{4}\sqrt{n}$

(d) $f(n) = \log(10n)$ and $g(n) = \log(n^2)$

(e) $f(n) = (\log n)^2$ and $g(n) = \log(n^2)$

(f) $f(n) = \log_{10} n$ and $g(n) = \ln n$

(g) $f(n) = 2^n$ and $g(n) = 3^n$

(h) $f(n) = n!$ and $g(n) = n^n$

2. Sequential search adapted from *Levitin* [2nd Ed.] 2.2.1. Use O, Ω and/or Θ to make the strongest possible claim about the runtime complexity of sequential search in,

(a) general (*i.e.*, all possible inputs)

(b) the best case

(c) the worst case

(d) the average case

3. Solving recurrence relations Solve the following recurrence relations, assuming $T(1) = 1$.

(a) $T(n) = T(n-1) + 4$ (b) $T(n) = T(n-1) + n$ (c) $T(n) = 2T(n-1) + 1$

4. k-Merge adapted from *DPV* 2.19. Consider a modified sorting problem where the goal is to sort k lists of n sorted elements into one list of kn sorted elements.

One approach is to merge the first two lists, then merge the third with those, and so on until all k lists have been combined. What is the time complexity of this algorithm? Can you design a faster algorithm using a divide-and-conquer approach?

5. Mergesort complexity (optional) Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

Construct a recurrence relation to describe the runtime of mergesort sorting n elements. Explain where each term in the recurrence relation comes from.

This kind of recurrence is difficult to solve by expansion. We haven't seen the master theorem yet, but you can look it up and use it to solve this recurrence relation and find the runtime complexity of mergesort if you finish these questions early.

Computer Lab

Modules and Multi-file C Programs. Up until this point, we have primarily written C programs using a single file containing all of the structs and functions for our program to run, as well as the main function which serves as the entry point to our program.

As our programs become more complex we will want to separate our code into smaller components containing related functionality. This helps us organise our code, as well as making code re-use easier.

1. Modules A module is made up of two files, a **header file** and a **C file**. Download `racecar.h` and `racecar.c` from the LMS, which makes up the racecar module.

A header file declares the functions and types which the module implements. It doesn't contain the definitions of any functions, just the prototypes.

Take a look at `racecar.h` and list the types and functions which make up the racecar module.

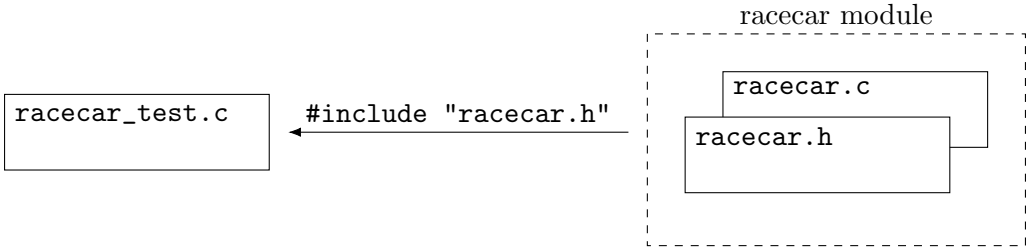
2. Using Modules Download the file `racecar_test.c` from the LMS.

The racecar module doesn't define a main function, and so on its own it is not a program that we can run. The `racecar_test.c` file defines a main function which creates some `Racecar` structs and prints out some information about them to test the racecar module.

The `racecar_test.c` file knows about the functions from the racecar module because it *includes the module*, which is achieved by the following line:

```
#include "racecar.h"
```

Note that we use double quotes (*e.g.,* `#include "..."`) for modules we have created, and angle brackets (*e.g.,* `#include <...>`) for C standard library modules.



By including the `.h` file, the C pre-processor essentially copy-pastes the content of `racecar.h` into `racecar_test.c` before compiling it, meaning it knows about any `typedef`s and function prototypes.

Note that multiple files can include the same module which can result in the same function prototype or type being declared more than once, which will be an error in C. To avoid erros when a module is included more than once we use **include guards**, which allows us to check whether the file has been included yet, and only declare the types/functions if it has not been. This is done using the following pre-processor directives:

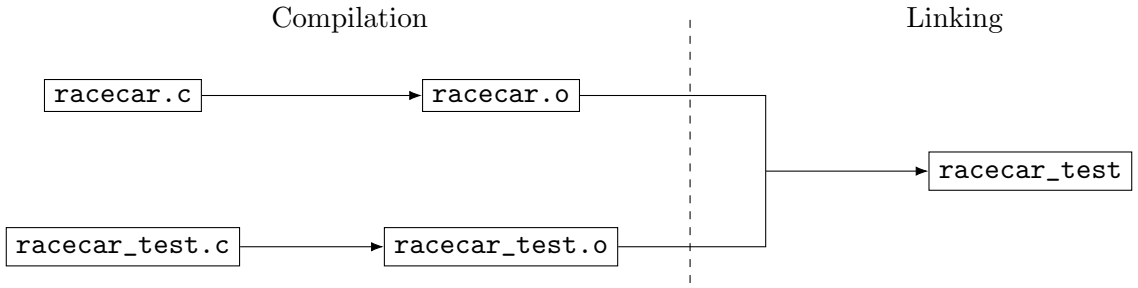
```
#ifndef RACECAR_H
#define RACECAR_H

... .h file contents go here ...

#endif
```

Compiling a multi-file C program involves two steps: compiling and linking.

First, each `.c` file has to be compiled into an object file (`.o`) and then the object files are linked to produce the program (*i.e.,* the executable file). In this example this looks like:



- To compile a file `file.c` into its object file `file.o` run:
  `$ gcc -Wall -c file.c`

- To link the object files (`file1.o`, ..., `fileN.o`) into an executable called `program` run:
  `$ gcc -Wall -o program file1.o ... fileN.o`

We can also do the compilation and linking in a single command, however this means re-compiling all `.c` files even if they haven't changed, so it's often faster to just re-compile the `.c` files which have changed and link everything again. To do this in one command you would run:
`$ gcc -Wall -o program file1.c ... fileN.c`

Compile and link the program to test the racecar module. Call this program `racecar_test`. Run this program to confirm that you've compiled it successfully. What does it output?

3. Compilation using a Makefile `make` is a tool for automatically compiling and linking multi-file C programs. It keeps track of which files have changed, so that only files that have changed (or had their dependencies changed) since the previous compilation will ever be recompiled.

To make use of `make` you must provide instructions for how your program should be compiled in a file called `Makefile` or `makefile`. Note that this file should not have a file extention, it won't work if your file is named `Makefile.txt` for example.

A `Makefile` is made up of *rules*. A rule has the following structure:

```
target: dependency1 .. dependencyN
  instruction1
  ...
  instructionM
```

Note: the indentation of the instuctions component of a rule must be a single tab, using spaces to indent will result in an error.

The `target` is the name of the file that should be created by a rule (*e.g.,* an ȯ file, or an executable file), the dependencies are the files that must already exist before the target can be created and the instructions are just command line instructions used to construct the target.

For example, the following rules compile and link a program named main which uses a module called foo.

```
main: main.o foo.o
  gcc -Wall -o main main.o foo.o

main.o: main.c foo.h
  gcc -Wall -c main.c

foo.o: foo.c foo.h
  gcc -Wall -c foo.c
```

Note that `main.c` and `foo.c` would both include `foo.h`, and so it must be a dependency for each of these rules.

To run this `Makefile` we would run `make <target>` (*e.g.,* `make main`) or just `make`, which will make the first target by default.

Now, create your own `Makefile` to compile the `racecar_test` program from *Question 2.*

4. Which parts of the module can we access? In `racecar.h` we have the following typedef:

```
typedef struct racecar Racecar;
```

In `racecar.c` we have the struct's definition:

```
struct racecar {
  char *driver;
  char *team;
  double *laps;
  size_t n_laps;
  size_t laps_capacity;
};
```

So any C file which includes `racecar.h` will know about a type called `Racecar` and know that it is a struct. But will it know *what information the struct contains*?

In the `racecar_test.c` file, try to print out the number of laps in the `renault` variable using `printf("n_laps: %d\n", renault->n_laps)`. Does this work? What error do you get?

It turns out that `racecar_test.c` doesn't know what information is in the struct, nor how to access it. This is potentially desirable, as we have abstracted the implementation details of the `Racecar` type away from those who use it. Rather, to interact with this struct you should use the functions which `racecar.h` provides.

Write a new function in `racecar.c` called `num_laps()` which returns the number of laps a particular `Racecar` has done. Use this function in `racecar_test.c` and re-compile your program.

Notice as well that the function `print_error` is defined within `racecar.c`, but the prototype is not present in `racecar.h`. Try calling this function from `racecar_test.c`. Does it work? What error do you get?

5. Linked List Module We'll use linked lists for many algorithms and data structures which are covered in this class, and it would be nice if we had some C code to use whenever we require a linked list in our programs.

In this question you should write a linked list module, comprised of `list.c` and `list.h`.

There will be some decisions you must make when you design your modules, for instance will your linked list be singly- or doubly-linked? Will you have a single node structure, or a linked list structure which contains more information about your list (*e.g.,* the head, tail and size of your list)?

Create functionality to insert and delete at the front and end of your list, find out how many elements are in the linked list and to free the linked list.

Once you've created your linked list module, create a main program to test your linked list.