UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

COMPUTER PROGRAMMING SKILLS AND CONCEPTS

Monday 13th August 2012

09:30 to 12:30

Convener: J Bradfield
External Examiner: A Preece

INSTRUCTIONS TO CANDIDATES

1. **Answer all questions.**
2. **Consult the separate printed sheet of instructions for details of how to get the files for the exam and submit your answers.**
3. **Sections A and B each account for half the marks.**
4. **Questions in section A are all worth 10 marks, but are not necessarily of equal difficulty. You are advised to answer them in order.**
5. **The two questions in section B are of approximately equal difficulty.**

Section A

This section contains five short questions worth 10 marks each, in which you are asked to implement either one function of a program, or a short complete program. **Little credit will be given for incomplete solutions.**

1. You are given a file `imperial.c` containing a skeleton program. Complete the `main` function to produce a program that converts *grams* (Metric system) into *pounds* and *ounces* (the Imperial system). To do the conversion, use the facts that *1 pound equals 454 grams* and *a pound is subdivided into 16 ounces*.

   Ensure that in computing the number of ounces, you should round *to the closest number* – ie any fraction equal to or more than 0.5 should add 1 to the number of ounces; otherwise it should be discarded.

   Here are some example results for testing:

   ```
   Enter number of grams: 42
   This equals 0 pounds and 1 ounces.

   Enter number of grams: 43
   This equals 0 pounds and 2 ounces.

   Enter number of grams: 900
   This equals 2 pounds and 0 ounces.
   ```

   You do **not** need to handle erroneous input.

   Note: If you use the function
   `double round(double x)`
   from `math.h`, you must remember to compile with the math library:
   `gcc imperial.c -lm`
   You may alternatively implement rounding yourself. [*10 marks*]

   When you are ready, submit the file `imperial.c`

2. You are given the file `power.c`, which contains the following function prototype:

   ```
   double power(double x, int n)
   ```

   Your task is to implement this function so that it accepts a real number `x`, and an integer `n`, and returns the value x^n.

   You cannot use `pow` or any other functions from the `math.h` library.

   You must give the solution for negative integers `n` as well as positive. You may assume x is non-zero. (x^{-n} is defined to be $1/x^n$, and $x^0 = 1$.)

   Here are two example results

   ```
   2.000000 to the power of 5 is 32.000000.

   4.000000 to the power of -3 is 0.015625.
   ```

   [*10 marks*]

   When you are ready, submit the file `power.c`

3. Academic staff at the University of Edinburgh have the titles Lecturer, Senior Lecturer, Reader, or Professor. In processions at events such as graduations, the order of precedence is defined thus: "the Professors in order of appointment as professor, then the Readers in order of appointment as reader, then the Lecturers and Senior Lecturers in order of appointment as lecturer". That is, a professor appointed in 2005 precedes one appointed in 2006, and all professors precede any reader, and so on; but a lecturer appointed in 2005 precedes a senior lecturer who joined the university in 2006.

   In the file `procession.c`, you will find enums and structs representing the rank and appointment date of members of staff. Complete the code for the function

   ```
   int precedes(staff s1, staff s2)
   ```

   so that it returns `1` if `s1` precedes `s2`, `-1` if `s2` precedes `s1`, and `0` if they have the same precedence. [*10 marks*]

   When you are ready, submit the file `procession.c`

4. In the template file `rotate.c`, you can see the function prototype:

   ```
   void rotate(int nums[], int n)
   ```

   together with some `main` code to allow the user to enter numbers from standard input, which are then saved in an array local to `main`.

   Implement the `rotate` function so that it rotates the first `n` items of the array `nums` by one position to the right. [*10 marks*]

   When you are ready, submit the file `rotate.c`

5. The perceived brightness p (in suitable units) to the human eye of light of a given wavelength λ (measured in nanometres) can be crudely approximated by $p = p_b + p_g + p_r$, where
p_b is $1 - \frac{|\lambda - 445|}{100}$ for $345 < \lambda < 545$ and zero otherwise;
p_g is $1 - \frac{|\lambda - 535|}{150}$ for $385 < \lambda < 685$ and zero otherwise;
p_r is $1 - \frac{|\lambda - 575|}{125}$ for $450 < \lambda < 700$ and zero otherwise.
($|x|$ is the absolute value of x, i.e. x if $x > 0$, and $-x$ if $x < 0$. The `math.h` function `fabs()`, of type `double fabs(double x)`, gives the absolute value of a floating point value.)

   Write a program `brightness.c` which reads in a wavelength and prints out the perceived brightness to three decimal places, as in the following examples:

```
Enter wavelength: 400
Brightness: 0.650

Enter wavelength: 535
Brightness: 1.780

Enter wavelength: 600
Brightness: 1.367
```

   You do **not** need to handle erroneous input.

   Note: Remember that when using functions from `math.h`, you must compile with the math library:
`gcc brightness.c -lm`
as well as have `#include <math.h>` in your program. [*10 marks*]

   When you are ready, submit the file `brightness.c`

Section B

This section contains two longer questions worth 25 marks each. The questions have several parts. Each part of the question is marked independently of any errors in previous parts.

1. In this question, we consider the problem of maintaining a database of half-marathon runners. We will support certain operations: finding the runner with the fastest time, adding a new runner to the database, using the results of a recent half-marathon to update a runner's entry, etc.

   The starting point for this question is the file `running.c`, which contains a suite of type declarations, as well as the function prototypes for this question (and the code for some helper functions). The first type declaration is a structured type `runtime_t` to store running results in hours, minutes and seconds. `runner_t` is another structured type containing the fields `name`, `runid` (unique id of the runner, guaranteed to be a positive integer), and two fields `pb` (personal best) and `recent` (most recent) of type `runtime_t`. The structured type `runclub_t` consists of an array of size `MAXRUNNERS` of type `runner_t`, together with a field `total` which we will use to maintain a count of the current number of runners in the running club.

```
typedef struct {
  int hr;
  int min;
  int sec;
} runtime_t;

typedef struct {
  char name[30];
  int runid;
  runtime_t pb;
  runtime_t recent;
} runner_t;

typedef struct {
  int total;
  runner_t runners[MAXRUNNERS];
} runclub_t;

typedef struct {
  char name[30];
  int runid;
  runtime_t result;
} result_t;
```

There is one extra structured type called `result_t` which is similar to `runner_t` except that it only contains one field for a running time. `result_t` is intended to be used when we are giving updates of recent races, or adding a new runner to the database.

In the file `running.c`, we have declared one global variable of type `runclub_t` named `slowAC`, and this will be the database which *all* our functions refer to. We assume that if there are k runners in the database, that `slowAC.total` will have the value k, and the runners will be stored at indices $0, \ldots, k-1$ of the `slowAC.runners` array.

(a) Your first task is to implement a function which takes a single parameter of type `runtime_t`, and computes the minutes-per-mile pace represented by this half-marathon time. You should use the fact that a half-marathon is 13.1 miles. The function prototype that you must complete is:

```
double minpermile(runtime_t time)
```

[*5 marks*]

(b) The second task is to implement a function which takes no input arguments, but which returns the unique runner id (the value of the `runid` field) of the runner who has the fastest `pb` field in the global database `slowAC`. You must complete the following function prototype:

```
int fastest()
```

[*10 marks*]

(c) The final task is to implement a function which takes one argument `res` of type `result_t`, and which *either* uses this input to *update the entry* of that runner (if `res.runid` has an entry in the database) *or* (if the runner with id `res.runid` is currently missing from the database) to add a new runner entry to the database, and update `slowAC.total`.

When updating the entry, you must update the `recent` field; `pb` should also be updated if no better time than `res.result` is previously known.

You must complete the function prototype below:

```
int updateDB(result_t res)
```

The value returned of the function should be 1 unless `res.runid` is a new runner and the database already has `MAXRUNNERS` (in which case you should return 0).

[*10 marks*]

Your code should be entered into `running.c`. This file contains some helper functions, including a function to `initialize` the database with a collection of runners. There is also a detailed `main` function which runs a menu allowing various functions to be tested.

When you are ready, submit the file `running.c`

2. In this question, you will implement some routines for a simple English–Latin dictionary.

   You are provided with two files, a skeleton program `dict.c` and a very small sample dictionary `latin.txt`. In the first two parts, you will use a tiny three-word dictionary built in to the program; in the third part, you will read in `latin.txt`.

   The main program reads the dictionary file if one is given on the command line, and then goes into a loop, reading English words from the user, looking them up, and printing the Latin translation.

   The dictionary is stored in a global array of dictionary entries. The struct type `dictentry_t` contains three fields: `english`, a character array of length `MAX_LENGTH+1` storing an English word; `latin`, a character array storing its Latin translation; `decl`, an integer giving the *declension* of the Latin word, a grammatical property we will use later.

   (a) Your first task is to implement the `Lookup()` function. This takes a string representing an English word, and searches through the dictionary array looking for an entry matching the argument string, and returns the index of the matching entry, or `-1` if none is found. If this is successfully implemented, a sample run of the program will be (where `krk:` is the command-line prompt)

   ```
   krk: ./a.out
   Printing out the dictionary.
   daughter filia 1
   gate porta 1
   master dominus 2
   Enter English word to look up: gate
   Latin translation is: porta
   Enter English word to look up: master
   Latin translation is: dominus
   Enter English word to look up: slave
   No translation found!
   Enter English word to look up: ^C
   ```

   [*5 marks*]

   (b) Your second task is to extend the program so that if the initial lookup of the input word fails, it sees if the word might be the plural of a word in the dictionary, and if so, returns the corresponding Latin plural form. Add code to the indicated place to do the following:
   If the input word ends in `"s"`, then remove the `"s"`, and look up the resulting word. If it is not found, do nothing (do not print any additional message). If it is found, then print a message, as in the example run that follows, and then print out the plural of the Latin word. The rules for making Latin plurals are:

- For a declension 1 word, add `"e"` to the end of the word.
- For a declension 2 word, replace the final `"us"` by `"i"`.

An example run is:

```
krk: ./a.out
Printing out the dictionary.
daughter filia 1
gate porta 1
master dominus 2
Enter English word to look up: masters
No translation found!
Found singular master
Latin translation is: domini
Enter English word to look up: gates
No translation found!
Found singular gate
Latin translation is: portae
Enter English word to look up: ^C
```

[*10 marks*]

(c) The program can be given an argument, the dictionary file:
`./a.out latin.txt`
Your final task is to complete the function `ReadDict()`, which reads the dictionary file and parses it into the dictionary array. The dictionary file contains one entry per line, with the English, Latin and declension separated by spaces. `ReadDict()` should parse this file into the array `dict[]`, returning `1` if it succeeds, and `0` in the event of any error. You should detect the following errors

- A word in the file is longer than `MAX_LENGTH`;
- There are more than `DICT_SIZE` lines in the file;
- A line does not have the correct format (word, word, integer).
- The declension is not `1` or `2`.

You should print an appropriate error message if you detect an error. [*10 marks*]

Notes: You may wish to use character-by-character processing, or scanf-processing. Since you are reading from the file `dictfile`, for character-by-character processing you should use `fgetc(dictfile)` to read a character; for scanf-processing you should use `fscanf(dictfile,`*format*`,`*args*`...)`.

Remember that the newline at the end of each line counts as white space for `scanf`; if you are reading character-by-character, you may find the `isspace()` function useful.

When you are ready, submit the file `dict.c`