UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08022 COMPUTER PROGRAMMING SKILLS AND CONCEPTS

Tuesday 18th December 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 `arith.c` containing a skeleton program. Complete the main routine to produce a program that demonstrates C arithmetic. The program should prompt the user for three integers, and then print the result of applying the C arithmetic operators `*`, `+` and `/` in the three expressions shown in the example below. The output should appear exactly as in the following example execution: [*10 marks*]

```
Enter first integer: 4
Enter second integer: 8
Enter third integer: 7
4 * 8 * 7 is 224
4 + 8 + 7 is 19
4 * 8 / 7 is 4.571429
```

   You must treat `/` as floating point division, and not integer 'div'. You do **not** need to handle erroneous input (neither non-`int` input, nor an attempt to divide by 0).

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

2. You are given the file `meansd.c` containing a skeleton program. Complete the main routine to produce a program that asks for four real numbers, computes their *mean* (the average) and *standard deviation*, and outputs those values to within 2 decimal places. Recall that the mean *mn* and standard deviation *std* of any 4 numbers a, b, c, d are defined by [*10 marks*]

$$
\begin{aligned}
mn &= \frac{1}{4}(a+b+c+d) \\
std &= \frac{1}{4}\left((a-mn)^2+(b-mn)^2+(c-mn)^2+(d-mn)^2\right)
\end{aligned}
$$

   Recall also that we use `%lf` to read a `double` with `scanf`.

   The output should appear exactly as in the following example execution:

```
Enter the 4 numbers: 3.0 5.5 88 301.0
Mean 99.38, standard deviation 14720.67
```

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

3. Complete the file dates.c to develop a program which prompts the user for a date in dd/mm format (day and month), and then prints that date in the *1st Apr, 20th Sep, 31st Jul* style. You must match the day's extension (*st, nd, rd* and *th*) correctly to the day. The rule for this is that all numbers between 1 and 31 take *th* **except** for the following: *[10 marks]*

   - 1, 21, 31 (which take *st*)
   - 2, 22 (which take *nd*)
   - 3, 23 (which take *rd*)

   dates.c includes an array declaration which has the 3-letter abbreviations for the Months. The output should appear exactly as in the following examples:

```
Please enter the day and month in dd/mm format: 5/11
Date is 5th Nov.

Please enter the day and month in dd/mm format: 2/8
Date is 2nd Aug.

Please enter the day and month in dd/mm format: 3/7
Date is 3rd Jul.
```

   You do **not** need to handle erroneous input: you may assume that the user will enter months within the range 1 to 12, and days within the range 1 to 31 (and an appropriate day for the given month).

   When you are ready, submit the file dates.c

4. In the template file maxtofront.c, you can see the function prototype: *[10 marks]*

```
  void maxToFront(int *a, int n)
```

   together with some main code which declares two arrays, and then makes calls to maxToFront, printing the results to standard output.

   Implement the maxToFront function so that it moves the largest element of a to the front of the array a, but leaves all other elements in their original relative order. If you implement the maxToFront function correctly, the output of the program (with the given main code) should look exactly like this:

```
After maxToFront, b is 44, 6, 2, 4, 5, -10, -6, 5, 8, 2.
After maxToFront, c is 44, 2, 3, -6, 4, 8, -2, 44, 9, 6, 1, 3, 4, -11, 0.
```

   When you are ready, submit the file maxtofront.c

5. In this question we consider the problem of "compressing" a string saved in the `char` array `name` - by compressing, we mean that we remove all non-alphabetical characters *except* the '`\0`' character at the end of the string, and that we make all alphabetical characters be lower case. [*10 marks*]

   The file `namecompress.c` contains some important `#include` directives for the problem, as well as setting up a test. Please complete the file to solve this problem.

   Note you may need to add variable declarations inside the answer block (and hence after the initial `strcpy`).

   You may find it helpful to check the "Character identification" section of the **C Quick Reference**.

   If your code is correct, the initial test in the file will give the output below:

```
Name after compression: williambgates
```

   However, you should also test your code on other strings.

   When you are ready, submit the file `namecompress.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 work with the `descartes` graphics library, and extend the functions of that library to create a function `DisplayPoint` to display a point (this functionality is not available in `descartes`). We then develop two other functions `DisplaySegments` (which allows the user to select a sequence of points, and draws these as a collection of disjoint line segments) and `BoundingBox` (which allows the user to select a sequence of points, and which re-draws the rectangular "bounding box" for the points-to-date after "every 5th point").

   Recall that a single point p in the plane is represented by an x and a y coordinate, $p = (x, y)$. The `descartes` library defines a structured type `point_t` to store points, and a line segment type `lineSeg_t` to store line segments (ordered pairs of points). It also offers a suite of functions for working with points and line segments through a graphics window. All these type declarations and function prototypes can be examined in the file `descartes.h` made available to you. *These also appear on the C Quick Ref sheet.*

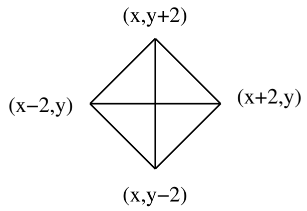
   Recall that the user signals they are finished entering points by clicking on the graphics window with the *middle mouse button*; this causes `GetPoint` to return a point with the dummy co-ordinates $(-1, -1)$.

   You will create your solution in the template file `pointsbounds.c`. It provides the *function prototypes* for the functions you are asked to write. It also provides a helpful `main` function set up to allow you to test each of your 3 functions.

   Your functions should not call `OpenGraphics()` or `CloseGraphics()` (you may assume this is taken care of by `main`).

   (a) The first function you must implement is `DisplayPoint`:
   `int DisplayPoint(point_t p)`. [*7 marks*]

   The goal is to use the functionality of the `DrawLineSeg` function in `descartes` for drawing line segments (`lineSeg_t`), in order to mimic the drawing of a point (`point_t`), as this latter functionality *does not* exist in `descartes`. Suppose we have a point `p` whose x and y-coordinates are `x` and `y`. Each of these values is guaranteed to lie within the integer range $[0, 500]$. Your function must achieve the drawing of `p` on the graphics window by considering the 4 points `(x, y+2)`, `(x, y-2)`, `(x-2, y)`, `(x+2, y)` and drawing 6 tiny line segments to connect them as shown in the picture overleaf.



Your function should return `1` (indicating success) if the point entered has both co-ordinates between 2 and 498, and `0` otherwise (in which case the point should not be drawn).

(b) The second function you must implement is `DisplaySegments`.
`int DisplaySegments (void)` [*8 marks*]

Your code must accept a sequence of points through the graphics window, drawing them with `DisplayPoint` as they are entered. After every *second* point, your code should draw a line segment between the current point and the previous one. Then as points are added, the window will display a collection of individual line segments.

You should return `1` (indicating success) if the user enters at most `MAXPOINTS` points before entering a *middle mouse button* click, or `0` if the user attempts to enter more than `MAXPOINTS` points. In the latter case, the function should `return 0` immediately after the attempt to enter the `MAXPOINT+1`-th point.

You may ignore the return value of `DisplayPoint`.

(c) The third function you must implement is `BoundingBox`:
`int BoundingBox(void)` [*10 marks*]

Your code must accept a sequence of points through the graphics window, drawing them with `DisplayPoint` as they are entered. After every *fifth* point, your code should draw the rectangular "bounding box" of your points (using the `DrawLineSeg` function from `descartes`); that is, the minimal rectangle which contains all the points entered so far.

You should return `1` (indicating success) if the user enters at most `MAXPOINTS` points before entering a *middle mouse button* click, or `0` if the user attempts to enter more than `MAXPOINTS` points. In the latter case, the function should `return 0` immediately after the attempt to enter the `MAXPOINT+1`-th point.

For this function you may ignore the return value of `DisplayPoint`.

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

2. In this question, we consider the problem of maintaining a stock database for a cafe. We will support certain operations: calculating the total value of the stock and current account of the cafe, checking the quantity of a particular product in stock, reading in the details of a new order of stock, and processing a new order of stock.

   The starting point for this question is the file `cafe.c`, which contains a suite of type declarations and function prototypes for this question (and the code for some helper functions). You can read the type declarations, displayed here at the bottom of the page. Before presenting the declarations, we give a short discussion.

   The first type declaration is a structured type `stock_t` to store the details (name, sale price and quantity) of a product in stock. `order_t` is another structured type containing the fields `name`, plus two fields of type `int`, those being `costprice` (the price-per-unit charged for that order) and `quantity` (the number of units ordered). The structured type `cafe_t` consists of an array of size `MAXPRODUCTS` of type `stock_t`, an `int` field `number` which we will use to maintain the count of current products in the cafe, and finally a field called `balance`, which is of type `double` and stores the current account balance for the cafe.

```
typedef enum {false, true} bool_t;

typedef struct {
  char name[MAXNAME];
  double saleprice;
  int quantity;
} stock_t;

typedef struct {
  int numproducts;
  stock_t products[MAXPRODUCTS];
  double balance;
} cafe_t;

typedef struct {
 char name[MAXNAME];
 int quantity;
 double costprice;
} order_t;
```

   In the file `cafe.c`, we have declared one global variable of type `cafe_t` named `swedish`, and this will be the database which *all* our functions refer to. We assume that if there are n different products in stock in the database, that `swedish.numproducts` will have the value n, and the `stock_t` entries for all these products are stored at indices $0, \ldots, n-1$ of the `swedish.products` array (with exactly one array entry for every product).

Note that for parts (b) and (d) you will probably want to use the `strcmp` function from the `string.h` library.

(a) Your first task is to implement the `valueSwedishCafe` function:
`double valueSwedishCafe (void)` [*5 marks*]

This should return the current value of the liquid assets of the Swedish cafe, where this value is equal to the current account balance (saved as `swedish.balance`), and the total *sale value* of all the current stock listed in the `swedish.products` array. Note that for any particular product in stock, its contribution to this overall value is the product of the *sale price* multiplied by the number of units in stock.

For the initial state of `swedish` after `initialiseSwedishCafe` has been run, the value returned by `valueSwedishCafe` should be 14900.00.

(b) Your second task is to implement the `quantityInStock` function:
`int quantityInStock (char *item)` [*5 marks*]

This function is called with a string `item` representing the name of the product of interest. The function must search the `products` array of the `swedish` database for an entry where the `name` field equals `item`. The function should `return` the `quantity` stored for the product if `item` is found in the `products` array, and should `return` 0 if `item` does not appear in `products`.

(c) Your third task is to implement the `readOrder` function:
`order_t readOrder()` [*5 marks*]

This function should ask the user (using `printf`) to enter the details of an order of type `order_t`, and return the details as an object of `order_t`. The phrasing and formatting of the input/output dialogue should be exactly as shown in the example below.

```
Which item? TomatoSoup
How many units? 200
Cost per unit? 3.00
```

You do not have to handle erroneous input.

(d) Your final task is to implement the `processOrder` function:
`bool_t processOrder (order_t order)` [*10 marks*]

This function takes as input an object of type `order_t`, and updates the `swedish` database to reflect the consequences of processing that order. If we are entering a new product, its `saleprice` will be set to 1.25*`costprice` in the database; otherwise, the `saleprice` will be left unchanged.

- The function should first check `swedish.balance` to check whether there are sufficient funds to pay for the order (the cost of the order is calculated from the `costprice` and the `quantity` of `order`).
  The function should return `false` if the funds are not there.
- Assuming that `swedish.balance` is large enough to pay for the order, the function should search for an entry in the `swedish.products` array with the same name as `order`.
  - If an entry is found, that entry's `quantity` should be updated to take account of that order. Finally `swedish.balance` should be updated (using `quantity` and `costprice` of `order`), and the function should return `true`.
  - In the case that no entry is found, then if `swedish.numproducts` is less than `MAXPRODUCTS`, a new entry in `swedish.products` should be created from `order` (in creating the new enty, `saleprice` should be set as `1.25` times the value of `costprice`).
    Also `swedish.numproducts` should be updated, `swedish.balance` should be adjusted as described above, and `true` should be returned.
    If `swedish.numproducts` is equal to `MAXPRODUCTS`, then no processing is carried out, and `false` should be returned.