Curtin University

Venue	____________________
Student Number	|__|__|__|__|__|__|__|__|
Family Name	____________________
First Name	____________________

Department of Computing

EXAMINATION

Semester 2 & Trimester 3B, 2015

COMP1000 UNIX and C Programming

This paper is for Bentley Campus students.

This examination has a total of 100 marks.

Examination Duration:	120 minutes.
Reading Time:	10 minutes.

Exam Conditions:

This is a CLOSED BOOK exam – no text books or written materials permitted.

Students are permitted to write notes during reading time in the margins or the reverse of the exam paper.

This question paper can be released to students after the exam.

No calculators are permitted in this exam.

Materials Permitted In The Exam Venue:

none

Materials To Be Supplied To Students:

1 x 16 page answer book

Instructions To Students:

This exam contains FIVE (5) questions. Answer all questions in the answer book provided.

The marks allocated for each question are shown beside the question.

For Examiner Use Only

Q	Mark
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	

Total _________

IMPORTANT INFORMATION

The possession or use of **mobile phones**, or any other device capable of communicating information, is prohibited during examinations.

Electronic Organisers/PDAs (with the exception of calculators) or other similar devices capable of storing text or restricted information are prohibited during examinations.

Only **calculators** approved specifically by the school/department may be used during this examination. Prior to the commencement of the examination, calculators will be checked for compliance by the examiner.

Any breach of examination regulations will be considered cheating and appropriate action will be taken in accordance with University policy.

Question 1 (19 marks)

(a) Give the type declarations needed for a linked list. The list must contain a length field, and each node in the list must contain:

- An enum, having possible values **`GREEN`**, **`YELLOW`** (which must be equal to 3) and **`RED`**.
- A pointer to a function. The function takes two parameters, a string and an integer, and returns an integer. **[4 marks]**

(b) Write a function called **`sumResults()`** that:

- Takes two parameters: a string, and a linked list of the type declared in part (a).
- Traverses the linked list.
- At each node, calls the function being pointed to, passing (1) the same string parameter supplied to **`sumResults()`**, and (2) the total number of elements in the linked list.
- Adds up all the return values from those calls.
- Returns the sum (an integer). **[9 marks]**

(c) Write a **`count()`** function that:

- Takes three integer pointer parameters (**`green`**, **`yellow`** and **`red`**), as well as a linked list of the type declared in part (a).
- Traverses a linked list.
- Counts the number of times that each of the values **`GREEN`**, **`YELLOW`** and **`RED`** appears in the linked list.
- Exports these three separate counts via the pointer parameters.
- Returns nothing. **[6 marks]**

Question 2 appears on the next page

Question 2 (10 marks)

For each of the following UNIX shell scripts, explain:

- Where the input (if any) is coming from;
- What actions are being performed; and
- What output could be generated.

(a)
```
#!/bin/bash
for f in $*; do
    cat $f | grep -E 'alpha[0-9][0-9]'
done
```

[3 marks]

(b)
```
#!/bin/bash
d=$(date)
if ./program $d; then
    echo $d
done
```

[4 marks]

(c)
```
#!/bin/bash
grep -E '^(eagle|falcon)*$' $1
```

[3 marks]

Question 3 appears on the next page

Question 3 (20 marks)

Consider the following code:

```c
int **a, **b;
int** c[2];
int i;

a = (int**)malloc(3 * sizeof(int*));
c[0] = (int**)malloc(4 * sizeof(int*));
a[0] = (int*)malloc(8 * sizeof(int));
for(i = 0; i < 8; i++)
{
    a[0][i] = 0;
}

b = a + 2;
c[1] = c[0];
*(a + 1) = &a[0][1];
a++;
*b = *a;
(*c)[0] = *b + 3;
c[1][1] = &b[0][3];
c[1]++;
c[1][1] = *a + 4;
(*c)[3] = **c + 2;

**a += 1;
**b += 2;
***c += 4;
c[1][1][1] += 8;
c[1][2][0] += 16;
```

Based on this:

(a) Draw a diagram showing all the pointer relationships created. If any pointers change, *neatly* and *clearly* cross-out the old arrows. **[16 marks]**

(b) Show all final integer values (except **i**). **[4 marks]**

Question 4 appears on the next page

Question 4 (21 marks)

The following function handles collision detection in a simple 2D game.

The game contains a series of moving "elements". Each element has a shape, which defines its position, size and perimeter. If the shapes of two elements overlap (because they've moved into each other), that is considered a collision. When a collision happens, the game needs to perform some action (depending on the kinds of elements).

You believe that **detectCollisions()** itself is working, but you suspect that the functions it relies on may have defects. You have already added two **printf()** calls to assist in debugging.

```c
void detectCollisions()
{
    GameElement* elementArray;
    Shape *elementShape1, *elementShape2;
    int nElements, i, j;

    elementArray = getElements(&nElements);

    printf("Starting comparisons\n");
    for(i = 0; i < nElements; i++)
    {
        elementShape1 = getShape(elementArray[i]);
        for(j = i + 1; j < nElements; j++)
        {
            elementShape2 = getShape(elementArray[j]);

            printf("Comparing %d and %d\n", i, j);
            if(shapesOverlap(elementShape1, elementShape2))
            {
                handleCollision(elementArray[i], elementArray[j]);
            }
        }
    }
}
```

Question 4 continues on the next page

For each situation below:

(i) Give **two plausible hypotheses** for what might be wrong (in the code *not* shown).
(ii) How do the hypotheses fit the observations?
(iii) What debugging steps (e.g. breakpoints, monitoring of particular variables) will help narrow down the problem?
(iv) How will you know which hypothesis is correct (or more likely to be correct)?

State any relevant (and realistic) assumptions you make about the code not shown.

(a) You happen to know that there are four elements, and exactly two of them have collided. The following is printed out:

```
Starting comparisons
Comparing 0 and 1
Comparing 0 and 2
Comparing 0 and 3
Comparing 1 and 2
Segmentation fault
```

That is, the function segfaults before finishing. **[7 marks]**

(b) The function consistently segfaults, and only the following is printed out:

```
Starting comparisons
Segmentation fault
```

[7 marks]

(c) The function displays all expected "Comparing x and y" messages, and finishes without segfaulting, but no collisions are ever actually handled. **[7 marks]**

Question 5 appears on the next page

Question 5 (30 marks)

For this question, refer to the following standard C function prototypes:

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *fp);
int fscanf(FILE *stream, const char *format, ...);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);

int sscanf(const char *str, const char *format, ...);
int strcmp(const char *s1, const char *s2);
char *strcpy(char *dest, const char *src);
int atoi(const char *nptr);
```

(a) Declare suitable C datatypes to represent each of the following sets of information (as you would do in a header file):

(i) Electricity usage data for a customer of a power company. This includes the customer's account number (an integer), their name (a string of up to 100 characters), and the relevant year and month (integers). Most importantly, it also includes the energy used (a real number) for each day in that month. (A month can, of course, have 28–31 days.) **[2 marks]**

(ii) A collection of the records you defined in part (i), consisting of data for various accounts in various months. **[2 marks]**

Question 5 continues on the next page

(b) Write a C function called **getRecords()** to read electricity usage records from a text file.

The first line of the file contains a single integer giving the total number of records. Each subsequent line contains one record, with information in the following order, separated by spaces except where otherwise indicated:

- The account number;
- The year and month, separated by a dash "–";
- 28 to 31 real numbers indicating the energy used on each day in the month;
- The customer's name (which may also contain spaces).

Here is an example file:

```
4
11307889 2015-1 6.83 4.1 0.3 9.9 [27 more real numbers] Joanne P Smith
11307889 2015-2 11.8 6.2 4.5 5.1 [24 more real numbers] Joanne P Smith
20338451 2015-1 18.4 26.9 13.3 18.1 [27 more real numbers] Jerry Li
20338451 2015-2 19.6 23.8 17.1 34.8 [24 more real numbers] Jerry Li
```

(The text "*[27 more real numbers]*" is not literally part of the file, of course. It is written here because the lines would be too long to print otherwise.)

For simplicity, you have access to a function for determining the number of days in a given month, declared as follows:

```c
int getNDays(int month, int year);
```

Your **getRecords()** function should:

- Take in a filename parameter (a string).
- Read the file, according to the above specifications.
- Dynamically allocate the structures you designed in part (a).
- Store the file data in these structures.
- Return a pointer to the main structure from part (a)(ii).

If the file *cannot* be opened, your function must return **NULL**. An error message is not required. If the file *can* be opened, you may assume that it definitely conforms to the specification.

[16 marks]

Question 5 continues on the next page

(c) Write a **`main()`** function to compute the total energy used for a given account.

Your code should:

- Accept two *command-line* parameters: an account number and a filename.
- Call **`getRecords()`** to retrieve the energy usage data for the given file.
- Calculate and display the total energy used for the given account. (Note: this could involve *zero or more* monthly records in the file.)
- Clean up.

For each possible error, your code must display the word "Error". However, you may assume the account number (if provided) is valid. **[10 marks]**

End of Examination