Question 1 (5 x 5 = 25 marks)

1a) What will the following program print?

```
#include <stdio.h>

int f(int a)
{
  static int b = 12;
  int c;

  c = b-- - a;
  return c;
}

int main(void)
{
        int i;
        for(i = 2; i < 7; i++)
              printf("%d\n",f(i));
        return 0;
}
```

1b) Not discussed in week 9 as material still to be taught (topic: macros).

1c) What will the following program print?

```
#include <stdio.h>

int f(int num)
{
        int a = 12, b;
        b = a/(num++);
        return (num = b);
}

int main(void)
{
        int num = 3;
        f(num);
        printf("%d\n",num);
        return 0;
}
```

1d) What will the following program print?

```
#include <stdio.h>
#include <string.h>

int main(void)
{
        char* txt[5] = {"Welcome",
                        "To",
                        "COSC1076",
                        "Exam",
                        "Semester 2 2018"};
        printf("%s\n",txt[1]);
        printf("%d\n",strlen(txt[2]));
        printf("%s\n",*(txt + 4));
        printf("%c\n",*(*txt + 1));
        printf("%c\n",*txt[2]);
        return 0;
}
```

1e) What will the following program print?

```
#include <stdio.h>

typedef struct cust
{
        char name[20];
        int age;
        float income;
} customer;

int main(void)
{
        customer premium = {"Jem Star", 24, 92000.00};
        customer *p;

        p = &premium;
        printf("%s\n",premium.name);
        printf("%5.0f\n",premium.income);
        printf("%d\n",(*p).age);
        printf("%c\n",*(p->name+2));


        return 0;
}
```

Question 2 (20 marks)

Write a program which accepts the names of two text files, a source and destination file from the command line. You may assume each line is no more than 100 characters long. The program copies each line of text in the original text file to the destination text file, with the letters of each line in the original file reversed. The following example illustrates this requirement.

Original file	Destination file
`Those who say it`	`ti yas ohw esohT`
`cannot be done`	`enod eb tonnac`
`should not interrupt`	`tpurretni ton dluohs`
`those doing it`	`ti gniod esoht`

Your program should check that both file names have been provided, with no other arguments apart from the executable name itself. If that is not the case, the program should exit with a suitable error message.

Question 3 (8 + 7 + 10 = 25 marks)

3a) Define a structure called *Course* for storing university course records. A course record stores an 8 character course *code* (e.g. COSC1076) as a string, a course *title* of up to 40 characters, an integer number of hours of instruction (called *hoursOfInstruction*), an integer number of credit points (called *creditPoints*) and a string course *description*. No assumption as to the length of *description* should be made.

3b) Assume a function called *deprecateCourse* is required to 'deprecate' a course record (as defined in 3a above). 'Deprecating' the record means adding the phrase ":DEPRECATED' to the end of the course description. The function has a void return type and accesses the course record to deprecate via a parameter. Write this function.

3c) Assume the functionality described in 3a) and 3b) above is extended to include a function *countOccurences* which will be used to search a given course *description* and determine the number of occurrences of a target word. The prototype of this function is:

```
int countOccurrences(char *str, char *targetWord);
```

The function has two parameters: a string, *str*, and a target word, *targetWord*. The string *str* is searched for occurrences of the target word, and the count of such occurrences is returned. The letter-case of *targetWord* and a matching word in *str* must match exactly in order for the count to be incremented. Words in *str* are delimited by spaces. No assumption as to the length of *str* or *targetWord* should be made.

Write function *countOccurrences* to implement the described functionality.

Question 4 (10 + 6 + 14 = 30 marks)

Not discussed in week 9 as material still being taught– covers dynamic data structures

Appendix A — Some commonly used C library functions

The following list presents selected standard library functions available in the C language, with brief descriptions as they might appear in a range of available texts and online manuals. For your convenience, you may wish to detach this and following pages, for ease of reference during the examination.

stdio.h

```
FILE * fopen(const char * filename, const char * mode)
```

`fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails. Legal values for mode include "r" open a file for reading, "w" create a file for writing; discard previous content if any, etc.

```
int fclose(FILE * stream)
```

`fclose` flushes any unwritten data for `stream`, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. It returns `EOF` if any errors occurred, and zero otherwise.

```
int fprintf(FILE * stream, const char * format, …)
```

`fprintf` converts and writes output to stream under the control of `format`. The return value is the number of characters written, or negative if an error occurred. The `format` string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications. (eg. %c, %d, %f, %s, etc.)

```
int fscanf(FILE * stream, const char * format, …)
```

`fscanf` reads from stream under control of `format`, and assigns converted values through subsequent arguments, *each of which must be a pointer*. It returns when `format` is exhausted. `fscanf` returns `EOF` if end of file or an error occurs before any conversion; otherwise it returns the number of input items converted and assigned.

```
int printf(const char * format, …)
```

`printf(…)` is equivalent to `fprintf(stdout, …)`

```
int scanf(const char * format, …)
```

`scanf(…)` is identical to `fscanf(stdin, …)`

```
int fgetc(FILE * stream)
```

`fgetc` returns the next character of `stream` as an unsigned char (converted to an int), or `EOF` if end of file or error occurs.

```
int fseek(FILE *stream, long offset, int whence);
```

`fseek()` function sets the file-position indicator for the stream pointed to by `stream` and returns 0 on success and –1 on error.

The new position, measured in bytes, from the beginning of the file, is obtained by adding `offset` to the position specified by whence, whose values are defined in `<stdio.h>` as follows:

`SEEK_SET`
Set position equal to offset bytes.

`SEEK_CUR`
Set position to current location plus offset.

`SEEK_END`
Set position to `EOF` plus offset.

ctype.h

`int isspace(int c)`

returns non-zero (true) if the argument `c` is the character code for a space, form-feed, newline, carriage return, tab, or vertical tab. Returns zero (false) for any other value of `c`.

`tolower(int c)`

if `c` is an uppercase character it returns the corresponding lowercase character, otherwise it returns `c`

`toupper(int c)`

if `c` is an lower case character it returns the corresponding uppercase character, otherwise it returns `c`

string.h

`char * strcat(char * s, const char * ct)`

concatenate string `ct` to end of string `s`; returns `s`.

`char * strtok(char * s, const char * ct)`

`strtok` searches `s` for tokens delimited by characters from `ct`; see below. To use `strtok`, see example below:

```
#define SEPCHARS " \t\n"

int main(void)
{
```

```
        char * word;
        char line[] = "Hello there         sir";

        word = strtok(line,SEPCHARS); /*Finds first token*/
        while (word != NULL)
        {
            printf("\"%s\"\n",word);
            word = strtok(NULL,SEPCHARS); /*Finds next token*/
        }
        return EXIT_SUCCESS;
    }
```

Output:

Hello
there
sir

`int strcmp(const char * cs, const char * ct)`

compare string `cs` to string `ct`; return < 0 if `cs<ct`, 0 if `cs==ct`, or >0 if `cs>ct`.

`size_t strlen(const char * cs)`

return length of `cs`.

stdlib.h

`void * malloc(size_t size)`

`malloc` returns a pointer to memory for an object of size `size`, or `NULL` if the request cannot be satisfied. The space is uninitialised.

`void free(void *p)`

`free` deallocates the memory pointed to by `p`; it does nothing if `p` is `NULL`. `p` must be a pointer to memory previously allocated by `calloc`, `malloc`, or `realloc`.

`int atoi(const char * s)`

converts `s` to `int`;

End of Appendix A