1. All of the “accounts” on a unix/linux system are represented by a file named “/etc/passwd”. Lines in /etc/passwd look like this:

```
    ajr:x:300003:500:Alan Rosenthal:/cmshome/ajr:/bin/bash
```

The sixth field is the home directory. The cut command takes a “-d” option which says what the field separator is, so the sixth field could be extracted with “cut -d: -f6”.

Write a shell script which outputs the percentage of the entries in /etc/passwd which have a home directory somewhere under /cmshome. If useful, you can assume that there are no spaces in any home directory names.

Possible hint: Remember the case statement, which can match “glob” patterns such as “/cmshome/*”.

```
inhome=
inother=

for name in $(cut -d : -f 6 /etc/passwd) ; do
    f=$(echo $name | grep '^/cmshome/')
    if [ -n "$f" ] ; then
        inhome=$(expr $inhome + 1)
    else
        inother=$(expr $inother + 1)
    fi
done
# echo "$inhome $inother"
sum=$(expr $inhome + $inother)
echo $((100 * $inhome/$sum))
```

2. A programmer writes the statement:

```
test "$x" = "$y"
```

as an 'if' condition. Remember that '=' is a string comparison in *test*.

(a) Why are the double quotes necessary? Explain by giving possible values for the variables x and y such that "test $x = $y" would yield a syntax error from test, but the above version with the double quotes would work as expected.
The double quotes are necessary for when either variable has spaces. If there were no double quotes and the variable had spaces, test would give a syntax error.
x is: test test
y is: test test

(b) Using a version of test written along the lines of our "simpletest", the programmer gets a syntax error anyway, even with the double quotes! That's because the variable x contained the two character string "-f", so "=" was considered to be a file name, and the value in y was then a syntax error. How would you change the above test statement to protect against this possibility?

```
tr -d "-f" $x $y # deletes "-f" from variables if they exist
test "$x" = "$y
```

3. State three distinct items of information (data) in the inode of a file in the unix filesystem (you needn't give their names from <sys/stat.h> ; just describe them with a few words each).

   (a) atime: the last time file was accessed

   (b) User ID: the user ID of owner of file

   (c) Filesize: size of file in bytes

4. Write a complete C program (except that you can omit the #includes) which takes one mandatory command-line argument, which it expects to be a directory. Other than "." and "..", that directory is expected to contain only plain files. Your program reads all of the files in that directory and outputs the total byte count.
   You must do all usual error checking and output appropriate error messages.
   Hint: If you chdir() to the directory, you can use the file names from readdir() as is.

```c
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
#include <sys/stat.h>

int dircheck(const char *path) {
    if (chdir(path) == -1) {
        perror("chdir");
        return -1;
    }
    DIR *folder;
    folder = opendir(".");
    struct dirent *entry;
    struct stat filestat;
    if (folder == NULL) {
        fprintf(stderr, "Unable to read directory\n");
        return -1;
    }
    int sum = 0;
    while( (entry=readdir(folder)) )
    {
        // if (entry->d_type == DT_REG) {
            stat(entry->d_name, &filestat);
            sum += filestat.st_size;
        // }
    }
    return sum;
}

int main(int argc, char *argv[])
{
    printf("%d\n", dircheck(argv[1]));
}
```

5. The diff program contains a sophisticated algorithm which decides which lines to compare to which. For a "trivial diff" program for this exam question, we evade this algorithm by assuming that files all have at most one line (n.b. *at most*; they could still be empty).
   Write a complete C program (except that you can omit the #includes) which takes two filenames (no other possibilities, no options) and behaves as diff except that it reads only the first line (if available). You can assume that the first line is a maximum of 1000 chars long, although you may not exceed array bounds even if it isn't. And all usual error checking and error messages are required.
   If the files are identical, there is no output. Otherwise, if the first file is empty and the second consists of the line "foo", *diff* will output:

```
    0a1
    > foo
```

   If the first file consists of the line "foo" and the second file is empty, *diff* will output:

```
    1d0
    < foo
```

   If both files consist of one line but it is different, *diff* will output:

```
    1c1
    < foo
    ---
    > bar
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

#define MAX_STR 1000

int main(int argc, char *argv[])
{
    char file1[MAX_STR], file2[MAX_STR];
    FILE *f1, *f2;
    if ( ((f1 = fopen(argv[1], "r")) == NULL) || ((f2 = fopen(argv[2], "r")) == NULL) ) {
      // Opens both files from args and checks for errors
        fprintf(stderr, "Error reading file\n");
        return -1;
    }
    // Uses fgets to read from file, and if fgets fails, puts '\0' as the first element of
     array
    if (( (fgets(file1, MAX_STR, f1) == NULL) && (file1[0] = '\0')) || ((fgets(file2,
    MAX_STR, f2) == NULL) && (file2[0] = '\0')) ) {
        fprintf(stderr, "File could not be read");
    }
    if (strcmp(file1, file2) == 0) {
        return 0;
    }
    else if (file1[0] == '\0') {
        printf("0a1\n> %s", file2);
    }
    else if (file2[0] == '\0') {
        printf("1d0\n< %s", file1);
    }
    else {
        printf("1c1\n< %s---\n> %s", file1, file2);
    }
    return 0;
}
```

6. Write a complete C program (except that you can omit the #includes) which calls fork(); the child executes the command "tr e f <file1 >file2" (without using *sh* or system()); and the parent calls wait() and outputs the child's exit status in the exact format "child process exit status was %d". You may hard-code the fact that 'tr' is /usr/bin/tr. All usual error checking and error message reporting is required.

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int f1, f2;
    if ((f1 = open(argv[1], O_RDONLY | O_CLOEXEC)) == -1) {
        perror("open");
        return -1;
    }
    if ((f2 = open(argv[2], O_WRONLY | O_CLOEXEC | O_CREAT, 00666)) == -1) {
        perror("open");
        return -1;
    }
    int status;
    switch (fork())
    {
        case -1:
            perror("fork");
            exit(1);
            break;
        case 0:
            dup2(f1, STDIN_FILENO);
            close(f1);
            dup2(f2, STDOUT_FILENO);
            close(f2);
            execl("/usr/bin/tr", "/usr/bin/tr", "e", "f", (char *) NULL);
            break;
        default:
            wait(&status);
            if (WIFEXITED(status)) {
                printf("child process exit status was %d\n", WEXITSTATUS(status));
            }
            exit(0);
            break;
    }
    return 0;
}
```

7. Write a complete C program (except that you can omit the #includes) which listens on port 2000 and receives messages, as follows: Anyone can connect to port 2000 and send data, and that data is output on the standard output, without alteration. You do not need to worry about the network newline conversion.

   Once a client connects, you read all data from it in a loop until end-of-file (caused by the client disconnecting), then you loop around for the next client; you do not need to handle multiple clients simultaneously.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAX_SIZE 256

int main()
{
    // create socket
    int listen_soc = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_soc == -1) {
        perror("server: socket");
        exit(1);
    }


    // initialize server address
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(2000);
    memset(&server.sin_zero, 0, 8);

    // server.sin_addr.s_addr = INADDR_ANY;
    if (inet_aton("127.0.0.1", &server.sin_addr) == 0) {
        fprintf(stderr, "Address not valid\n");
        exit(1);
    }

    // bind socket to an address
    if (bind(listen_soc, (struct sockaddr *) &server, sizeof(struct sockaddr_in)) == -1) {
      perror("server: bind");
      close(listen_soc);
      exit(1);
    }

    // Still needed the listen line or accept wouldn't work
    if (listen(listen_soc, 5) < 0) {
        perror("listen");
        exit(1);
    }

    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(struct sockaddr_in);
    client_addr.sin_family = AF_INET;

    char name[MAX_SIZE];

    for (;;) {
        int client_socket = accept(listen_soc, (struct sockaddr *) &client_addr, &
    client_len);
        if (client_socket == -1) {
            perror("accept");
            exit(1);
        }
```

```
        FILE *socfd = fdopen(client_socket, "r");
        for (;;) {
            // close(listen_soc);
            if (fgets(name, MAX_SIZE, socfd) == NULL) {
                fprintf(stderr, "Client disconnected\n");
                close(client_socket);
                fclose(socfd);
                break;
            }
            else {
                printf("%s", name);
            }
        }
    }
    return 0;
}
```

8. From a high-level view, main() is the first thing executed in a C program. But there has to be some code which calls main().
   When your program is executed in unix, it starts executing at a certain address. Code known as "crt0" (for "C runtime, the very first thing") sets up various things needed by the C library, and calls main() with the appropriate arguments.
   The last line of crt0.c (before a closing brace) is usually something very similar to:

   ```
   exit(main(argc, argv));
   ```

   What is this line doing?

   This line flushes all open file descriptors for crt0 and calls main with the required arguments all in one. (Not sure about this one)