FIT2100 Practical #2
Managing C Programs, Debugging with GDB, and Running I/O System Calls in C
Week 4 Semester 2 2017

Dr Jojo Wong
Lecturer, Faculty of IT
Email: `Jojo.Wong@monash.edu`

August 11, 2017

Revision Status:

`$Id: FIT2100-Practical-02.tex, Version 1.0 2017/08/10 10:30 Jojo $`

The content presented in Sections 2.1 and 2.2 in this practical were adapted from the courseware of FIT3042 prepared by Robyn McNamara.

The content presented in Section 2.3 and the practical tasks (Section 3) were adapted from David Curry's texts.

- David A. Curry (1989). *C on the UNIX System*, O'Reilly.
- David A. Curry (1996). *UNIX Systems Programming for SVR4*, O'Reilly.

Contents

1 Background **5**

2 Pre-lab Preparation (4 marks) **5**

2.1 Managing C programs 5

2.1.1 Organising your source files 5

2.1.2 Building complex C programs 6

2.1.3 The `makefile` format 6

2.1.4 Running the `make` command 8

2.1.5 Defining macros in `makefile` 8

2.1.6 Try it yourself (2 marks) 9

2.2 Debugging C programs 10

2.2.1 Handling errors with defensive programming 10

2.2.2 The `gdb` debugger 11

2.2.3 Dealing with the program bugs 12

2.2.4 Try it yourself (2 marks) 14

2.3 I/O System Calls in C 15

2.3.1 File descriptors 15

2.3.2 Opening and creating files 16

2.3.3 Reading and writing files 17

2.3.4 Moving around files 18

3 Practical Tasks (6 marks) **20**

3.1 Task 1 (2 marks) 20

3.2 Task 2 (1 mark) 22

3.3 Task 3 (3 marks) 23

1 Background

This practical has two objectives. Firstly, you will learn how to manage complex C programs and debug C programs using the debugging tool known as `gdb`. Secondly, you will learn how to perform low-level I/O systems calls in C.

There are some pre-lab preparation (Section 2) that you should complete before attending the lab. The practical tasks specified in Section 3 are to be completed and assessed in the lab.

2 Pre-lab Preparation (4 marks)

2.1 Managing C programs

Often times, working with a small programming project, you could probably get away with putting all your code in one source file. For much larger projects, you would need to organise your functions together into various source files by grouping related functions together.

A general *rule of thumb* is that your functions should not be much larger than a screenful (approximately 25 lines of code), excluding header comments. If your functions appear to be larger than this, you should consider breaking them down into smaller functions. (Note that your function names should be descriptive to help other coders to navigate your source files.)

2.1.1 Organising your source files

One of the characteristics of the C programming language is that it encourages the idea of "separate compilation".

Instead of putting all the code in one file and compiling that one file, C allows you to write many `.c` files and compile them separately when necessary. The usual way to group related C source files is to put them under the same directory.

What goes into the `.c` file? Anything that causes the compiler to generate code should go into the `.c` source file. A `.c` file usually consists of:

- implementation of functions (i.e. function bodies)
- declaration of global variables — in order to set aside memory for global variables

Anything that is intended as a message to the compiler should go into the `.h` header file, which include:

- function prototypes
- `struct` definitions
- `typedef` statements
- `#define` statements

Note: Only `.h` files should ever be included using the `#include` statement — you should never include `.c` files.

2.1.2 Building complex C programs

A C program can potentially consist of hundreds of source files (both `.c` and `.h` files), and possibly splitting across many directories.

The standard command given below is used to compile a complex C program with multiple source files. However, *this approach does not scale!*

```
$ gcc -Wall sourcefile1.c sourcefile2.c ... -o myprogram
```

Whenever the program is rebuilt, this command recompiles every source file, which can be time consuming. It should only be recompiling those source files that have been modified instead of every single file. The solution to this is by utilising the `make` command in C.

`make` is a utility tool in C that automates the building of program binaries from source files. A number of built-in rules are defined, such that `make` knows how to compile source code (`.c` files) into object code (`.o` files). `make` requires a special file called `makefile` that specifies the source files and the targets (which include the executable code) to be build.

2.1.3 The `makefile` format

A `makefile` typically consists of one or more entries. Each entry consists of the following:

- a target (which usually is a file but not always);
- a list of dependencies (files which the target depends on);

- and the commands to execute based on the target and its dependencies

The basic format for each entry in a `makefile` looks like below:

```
<target>: [ <dependency> ]*
 [ <TAB> <command> <ENDLINE> ]+
```

(**Note:** There must be a <TAB> character at the start of each command defined under the target.)

An example of a `makefile`:

```
sourcefile1.o: sourcefile1.c headerfile1.h
	gcc -Wall -c sourcefile1.c
```

In the example above, the `-c` option is needed to create the corresponding object file (`.o`) for the given `.c` file.

How and when a target is constructed? Each entry in the `makefile` defines *how* and *when* to construct a target based on its dependencies. The dependencies are used by the `makefile` to determine when the target needs to be reconstructed (by re-compiling the source code).

Each file has a *timestamp* that incidates when the file was last modified.

The `make` command first checks the timestamp of the file (i.e. the target) and then the timestamp of its dependencies (which are also files). If the dependent files have a more recent timestamp than the target file, the command lines defined under the entry of that target are executed to update the target.

Note: The updates are done recursively. If any of the dependencies themselves are also targets (i.e. there is a separate entry in the `makefile` for each of the dependencies), the `make` command will check whether these dependent files need to be updated, before updating the initial target.

How to create the executable code using targets? The purpose of having the `makefile` is to create the executable code (i.e. `a.out`). The executable file is often the first target.

```
myprogram: sourcefile1.o sourcefile2.o sourcefile3.o
	gcc -Wall sourcefile1.o sourcefile2.o sourcefile3.o -o myprogram
```

2.1.4 Running the `make` command

Once you have a `makefile` written, you can make use of the `make` command to update your files whenever you make changes. There are two ways to run `make` as shown below.

```
$ make
$ make -f <makefilename>
```

If you just type `make` on the command prompt, the file called `makefile` in the current directory will be interpreted and the commands of the first target will be executed, provided that the first target has dependencies listed.

Alternatively, if you have more than one `makefile` with different file names, you will run the `make` command with the `-f` option by supplying the file name of the specific `makefile` that you would like to run.

2.1.5 Defining macros in `makefile`

You can use *macros* in your `makefile`, which allow you to define variables that can be substituted.

Macros have a similar syntax to shell variables, where you can use the "=" operator to set a value to the macro, and use the "$" operator when using the value of the macro. (Note that the parentheses are required if the macro name has more than one character.)

Important: Don't get confused with the shell command prompt which can also be represented by a "$" symbol.

```
CC = gcc
CFLAGS = -Wall -c

sourcefile1.o: sourcefile1.c headerfile1.h
	$(CC) $(CFLAGS) sourcefile1.c
```

The example above demonstrates two common macros `CC` and `CFLAGS`. Below are some of the other common examples of macros.

Macro	Description
`CC`	the name of the compiler (`gcc`)
`DEBUG`	the debugging flag (`-g`)
`CFLAGS`	the flags used for compiling (e.g. `-c`, `-Wall`)
`LFLAGS`	the flags used for linking (e.g. `-L`, `-I`)
`$@`	the file name of the target
`$^`	the list of dependencies
`$<`	the first item in the dependencies list

Putting it all together Here is a sample of `makefile` for creating the executable program called `myprogram`.

```
OBJS = sourcefile1.o sourcefile2.o sourcefile3.o
CC = gcc
CFLAGS = -Wall -c
LFLAGS = -Wall

myprogram: $(OBJS)
	$(CC) $(LFLAGS) $(OBJS) -o myprogram

sourcefile1.o: sourcefile1.c headerfile1.h
	$(CC) $(CFLAGS) sourcefile1.c

sourcefile2.o: sourcefile2.c headerfile2.h
	$(CC) $(CFLAGS) sourcefile2.c

sourcefile3.o: sourcefile3.c headerfile3.h
	$(CC) $(CFLAGS) sourcefile3.c
```

2.1.6 Try it yourself (2 marks)

Refer to Task 3 that you have completed for the Week 1 Tutorial. First, organise your code into three separate files: `main.c`, `arithmetic.c`, and `arithmetic.h`.

- `main.c`: consists of the `main()` function
- `arithmetic.c`: consists of the implementation of the four arithmetic functions (i.e. function bodies)
- `arithmetic.h`: consists of the function prototypes of the four arithmetic functions

Then, create a `makefile` and try to compile using the `make` command.

2.2 Debugging C programs

Some kinds of errors, such as typos and incorrect algorithms, are equally common in all programming languages. However, there are some kinds of errors that are more likely to affect C programs, such as buffer overflows and misuse of pointers.

The underlying reason is that C generally lacks safeguards that other programming languages may have. C does not attempt to hide implementation details from the programmers; likewise C does not prevent the programmers from attempting to access memory that is not allocated to the programmers' code. Notice that one of the common error under Unix/LInux is "segementation fault" (`segfault`).

What is a segmentation fault? A segmentation fault occurs when you attempt to access memory that has not been allocated to it. It is generally a signal sent from the operating system (OS) to your program, and causing your program to exit. The OS that manages memory will not let your program to read from or write to memory segments that do not own by your program.

2.2.1 Handling errors with defensive programming

The best approach to handle errors in C is by preventing errors through "defensive programming" techniques.

- Always check return values from functions (such as `malloc()` and `fopen()`)
- Always initialise pointers to `null` on creation if the valid target is not given immediately
- Always check that pointers are not `null` before deferencing them (see below)

```c
if (ptr_str)
    *ptr_str = "FIT2100";
```

Logging and diagnostics Often times, defensive programming is unable to catch all classes of errors. You can however instrument your code by getting it to output informative messages. For instance, `fprintf(stderr)` is one way to handle this — `fprintf()` can also be used to output messages to a logfile. Alternatively, use `prinf()` as diagnostic statements in your code.

Note that these programming techniques are not specific to the C language or Unix/Linux. You would have certainly used diagnostic output statements in any other programming language that you have programmed in.

2.2.2 The gdb debugger

A debugger is a program that runs your program inside it, which allows you to inspect what the program is doing at a certain point during its execution. The debugger also enables you to manipulate the program state while it is running. As such, errors such as segmentation faults may be easier to detect with the help of a debugger.

The debugger that is often used with the C compiler gcc under the Unix/Linux environment is the gdb — GNU Debugger.

Usually, you would compile a C program as follows:

```
$ gcc [flags] <source files> -o <output file>

$ gcc -Wall sourcefile1.c sourcefile2.c ... -o myprogram
```

To run your program with the gdb debugger, you add a -g option to enable the built-in debugging support which is needed by the gdb.

```
$ gcc [other flags] -g <source files> -o <output file>

$ gcc -Wall -g sourcefile1.c sourcefile2.c ... -o myprogram
```

Starting up the debugger To start up the gdb debugger, you could just try with one of the following commands. (Note: myprogram is the executable program that you would like to debug.)

```
$ gdb
$ gdb myprogram
```

You will then get a prompt that looks as follows:

```
(gdb)
```

If you didn't specify a program to debug while starting up the debugger, you will have to load it with the "file" command after started up the debugger.

```
$ gdb
(gdb) file myprogram
```

Running your program with the debugger To run your program (say `myprogram`), just type "`run`":

```
(gdb) run
```

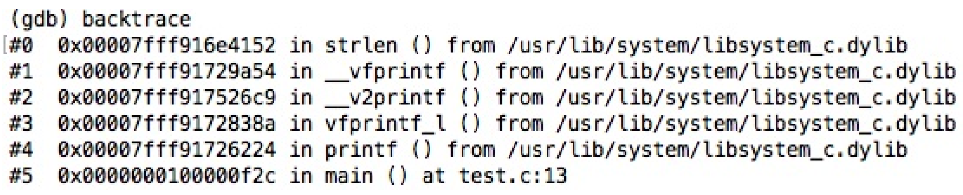
Your program will get executed. There are two possible outcomes here:

- If there are no serious problems (such as, no segmentation fault occurred), your program should run accordingly with the program output presented.
- If your program did have some problems, you should be getting some useful information about the error. (See below for an example of the error message.)

```
Program received signal SIGSEGV, Segmentation fault.
0x00007fff916e4152 in strlen () from /usr/lib/system/libsystem_c.dylib
```

The "`where`" command can be used to find out the line of code where your program crashed. This is in fact the metadata that the -g flag includes in the executable program.

Another command that can be helpful for finding out where your program crashed is the "`backtrace`" command — which produces a stack trace of the function calls that lead to a segmentation fault. (An example of a stack trace is shown below.)



```
(gdb) backtrace
#0  0x00007fff916e4152 in strlen () from /usr/lib/system/libsystem_c.dylib
#1  0x00007fff91729a54 in __vfprintf () from /usr/lib/system/libsystem_c.dylib
#2  0x00007fff917526c9 in __v2printf () from /usr/lib/system/libsystem_c.dylib
#3  0x00007fff9172838a in vfprintf_l () from /usr/lib/system/libsystem_c.dylib
#4  0x00007fff91726224 in printf () from /usr/lib/system/libsystem_c.dylib
#5  0x0000000100000f2c in main () at test.c:13
```

2.2.3 Dealing with the program bugs

If your program is running successfully, you would not need a debugger like gdb. But what if your program is not running properly or crashed? If that is the case, you do not want to run your program without stopping or breaking at a certain point during the execution. Rather, what you would want to do is to *step through* your code a bit at a time, until you discovered where the error is.

Setting the breakpoints A *breakpoint* instructs the gdb debugger to stop running your program at a designated point. To set breakpoints, use the command "`break`" in one of the following ways:

- To break at line 5 of the current program:

```
(gdb) break 5
```

- To break at line 5 of a source file named `sourcefile.c`:

```
(gdb) break sourcefile.c:5
```

- To break at a function named `myfunction.c`:

```
(gdb) break myfunction
```

You can set multiple breakpoints within your program. If your program reaches any of these locations when running, your program will stop execution and prompt you for another debugger command.

Once you have set a breakpoint, you can try to run your program with the command "`run`". The program should then stop where you have instructed the debugger to stop — provided no fatal errors occurred before reaching the breakpoint.

To proceed with the next breakpoint, use the "`continue`" command. (You should not try to use the "`run`" command here as that would restart the program from the beginning, which is not going to be helpful.)

Note: Any breakpoints can be removed with the command "`clear`" — e.g. to delete the breakpoint at line 5:

```
(gdb) clear 5
```

Tracing your code When you program hits the breakpoint, you can step through your program one line at a time with the following commands:

- "`step`" command: go to the next line, stepping *into* any functions that the current line calls
- "`next`" command: go to the next line, but stepping *over* any functions that the current line calls
- "`finish`" command: finish the current function and *break* when it returns

Keeping an eye on variables You can also tell the gdb debugger to show the values of various variables or expressions at all times using the "`display`" or "`print`" command. Both of these commands take an optional format argument to display the value in a specific format:

- "`print/x`" command: print in hexadecimal

- "`print/c`" command: print in character
- "`print/t`" command: print in binary

```
(gdb) print myvariable
(gdb) print/x myvariable
```

If you would prefer to monitor or watch a particular variable whenever its value changes, use the "`watch`" command. The program will be interrupted whenever a watched variable's value is modified.

```
(gdb) watch myvariable
```

If you ever get confused about any of the `gdb` commands, use the "`help`" command with or without an argument.

```
(gdb) help [command]
```

Finally, to exit the `gdb` debugger is the "`quit`" command.

```
(gdb) quit
```

2.2.4 Try it yourself (2 marks)

For the given C program below, type it out and name the source file as "`test.c`". Then, run through it with the `gdb` debugger. You will need to step through the program code to detect where the error occurred. Once you have found the error, fix it so that the program will run successfully.

```c
#include <stdio.h>

int main(void)
{
        char str1[] = "FIT2100 Operating Systems";
        char str2[] = "Week 4 Practical";
        char *ptr = NULL;

        ptr = str1;
        printf("%s\n", ptr);

        ptr = str2;
        printf("%s\n", *ptr);

        return 0;
}
```

2.3 I/O System Calls in C

Note: You should complete the reading of this section before attempting the practical tasks in the next section.

The Standard I/O Library (`stdio`) provides a collection of *high-level* routines to perform input and output (I/O). This enables C programmers to perform reading and writing data easily and efficiently.

In general, the `stdio` routines perform three important functions for the programmers:

- Buffering on input and output data is performed automatically;
- Input and output conversions are performed using routines like `printf` and `scanf`;
- Input and output are also automatically formatted.

However, these functions such as buffering and I/O conversions are not always applicable. In the event where performing I/O directly from a device such as a disk drive, programmers would need to be able to determine the actual buffer size to be used instead of relying on the `stdio` routines.

Thus, a direct interface to the operating system is desirable and this can be achieved by issueing *system calls* or using the *low-level* I/O interface.

2.3.1 File descriptors

When dealing with the low-level I/O interface, each of low-level I/O routines requires a valid "file descriptor" to be passed to them — which is used to reference a file to an open stream for I/O.

A file descriptor is simply a small integer, which can be allocated from a file index table maintained for each process (or program) by the operating system.

There are three pre-defined file descriptors:

- File descriptor `0`: refer to the standard input
- File descriptor `1`: refer to the standard output
- File descriptor `2`: refer to the standard error output

When a process (or program) begins with its execution, it starts out with three opened files — i.e. the three pre-defined file descriptors `0`, `1`, and `2`. When the process opens the first file, it will be attached with file descriptor `3`.

2.3.2 Opening and creating files

In order to read from or write to a file (including the standard input and output), that file must first be opened. The routine (or function) used to *open* a file or to *create* a file for reading and/or writing is called `open()`.

The `open()` function takes three arguments:

- a character string consists of the path name of a file to be opened;
- a set of flags or integer constants specifying how the file is to be opened;
- an integer *mode* used to specifying the access permission when creating a new file.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

The second argument is constructed by OR-ing together a number of flags or constants (see below) defined in the header file `sys/fcntl.h` for System V systems, and `sys/file.h` for BSD (Berkeley Software Distribution) systems.

A subset of the flags (constants) that control how a file is to be opened are shown in the table on the next page. Note that the first three flags are mutually exclusive.

The open routine returns a file descriptor for the file if it is opened successfully. Otherwise, the value of `-1` is returned if it fails to open that file. An error code indicating the reason for failure is stored in the external variable `errno` defined in the header file `errno.h`. You can print out the error message with the `perror` function.

Flag (Constant)	Description
O_RDONLY	The file is opened for read only
O_WRONLY	The file is opened for write only
O_RDWR	The file is opened for both read and write
O_APPEND	The file is opened in append mode
O_CREAT	Create the file if it does not exist; the mode argument must be supplied to specify the access permission on the file
O_EXCL	Check if the file to be created is already exists; if already exists, open() will fail
O_NONBLOCK or O_NDELAY	The file is opened in non-blocking mode
O_TRUNC	If the file is opened for writing, truncate the length of the file to zero

Often times, the opened file should be closed once a process (or program) has finished using that file. The routine to *close* a file is called `close()`, which takes in only one argument representing the file descriptor of the file to be closed. If the file was closed successfully, the value of `0` is returned; otherwise, the value of `-1` is returned if an error occurs.

```c
#include <unistd.h>

int close(int fd);
```

2.3.3 Reading and writing files

With the low-level I/O interface, the `read()` and `write()` routines are used for reading from and writing to a file respectively.

```c
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

The `read()` attempts to read up to `count` bytes from the file referred to by the file descriptor `fd` into the buffer pointed by `buf`. The function returns the number of bytes actually read if no error occurs; other the value of `-1` is returned and the external variable `errno` is set with the error code. If the return value is `0`, that indicates the end of the file has been reached and there is no data left to be read.

The `write()` writes up to `count` bytes from the buffer pointed by `buf` to the file referred to by the file descriptor `fd`. If the write is successful, the number of bytes actually written is returned; otherwise the value of `-1` is returned if an error occurs and `errno` is set accordingly. (If nothing was written, the return value is `0`.)

When using these low-level functions, the programmers have to decide on the size of the buffer in bytes (as indicated by the third argument `count` in both functions). Note that each time the `read()` or `write()` function is executed, the operating system will access the disk (or other I/O device). If the buffer size is set to 1, meaning that you would only be able to read or write one byte (one character) at a time — the execution of your program will turn out to be extremely slow.

Note: Both `read` and `write` do not attempt to perform data conversion or formatting. The contents of data that we are dealing with when using these low-level functions are in the byte form rather that the human-readable form.

2.3.4 Moving around files

Often times, we need to be able to move around within a file to access the data stored at a specific location in the file. Each file is associated with a value, known as the *file offset*. It is often set to `0` indicating the beginning of the file when a file is opened or created.

The value of the file offset can be obtained or re-set by using the low-level routine `lseek()`.

```c
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

The `lseek()` function re-positions the file offset (`offset`) of a file referred to by the file descriptor `fd`, based on the position in the file specified by the last argument in the function `whence`. The values that can be assigned to `whence` are as follows:

Offset (Constant)	Description
SEEK_SET	The file offset is set to `offset` bytes from the beginning of the file
SEEK_CUR	The file offset is set to `offset` bytes from its current position
SEEK_END	The file offset is set to `offset` bytes from the end of the file

So, if you were to move to the beginning or the end of a file:

```c
lseek(fd, 0, SEEK_SET);   /* move to the beginning of the file */
lseek(fd, 0, SEEK_END);   /* move to the end of the file */
```

To obtain the value of the current offset of the file:

```c
off_t current;
current = lseek(fd, 0, SEEK_CUR);
```

Note that the `lseek()` returns the new offset value in bytes from the beginning of the file if it runs successfully. If there is an error, the value of `-1` is returned. (The `errno` variable is set to indicate the error.)

3 Practical Tasks (6 marks)

3.1 Task 1 (2 marks)

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>   /* change to <sys/fcntl.h> for System V */
#include <unistd.h>

/*
 * appendfile.c: append the contents of the first file to the second file
 */
int main (int argc, char *argv[])
{
    int n, infile, outfile;
    char buffer[1024];

    if (argc != 3) {
        write(2, "Usage: appendfile file1 file2\n", 30);
        exit(1);
    }

    /*
     * Open the first file (file1) for reading
     */
    if ((infile = open(argv[1], O_RDONLY)) < 0) {
        perror(argv[1]);
        exit(1);
    }

    /*
     * Open the second file (file2) for writing
     */
    if ((outfile = open(argv[2], O_WRONLY | O_APPEND)) < 0) {
        perror(argv[2]);
        exit(1);
    }

    /*
     * CODE HERE: Copy data from the first file to the second file
     */

    /*
     * Close the two files before exiting
     */
    close(infile);
    close(outfile);
    exit(0);
}
```

appendfile.c is a partially completed C program, that takes two file names — file1 and file2 — as the command-line arguments. It opens the first file (file1) for reading and the second file (file2) for writing. The contents of the first file (file1) is appended at the end of the second file (file2).

Your task is to complete the missing code segment in the program as indicated by the comment — /* CODE HERE */. To test the program, create two files and name them as file1 and file2. The contents of each file is given below by using the less command in Unix.

```
$ less file1
Line 1: the first sentence in file1
Line 2: the second sentence in file1
Line 3: the third sentence in file1

$ less file2
Line 1: the first sentence in file2
Line 2: the second sentence in file2
Line 3: the third sentence in file2
```

Note: To run or test the program in Unix, use the following command:

```
$ appendfile file1 file2
```

3.2 Task 2 (1 mark)

Type out the given program below, and you may name it as `seekinfile.c`. Compile the program and run it. Your task is to describe what does the program do. Include your description with 2-3 sentences as a comment at the beginning of the program code.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>     /* change to <sys/fcntl.h> for System V */
#include <unistd.h>
#include <string.h>

struct record {
    int userid;
    char username[6];
};

char *usernames[] = { "userA", "userB", "userC", "userD"};

int main(int argc, char *argv[])
{
    int i, outfile;
    struct record eachrec;

    /*
     * Open the file (recordfile) for writing
     */
    if ((outfile = open("recordfile", O_WRONLY | O_CREAT, 0664)) < 0) {
        perror("recordfile");
        exit(1);
    }

    for (i = 3; i >= 0; i--) {
        /*
         * Create a new record
         */
        eachrec.userid = i;
        strcpy(eachrec.username, usernames[i]);

        /*
         * Write the record into the file
         */
        lseek(outfile, (long) i * sizeof(struct record), SEEK_SET);
        write(outfile, &eachrec, sizeof(struct record));
    }

    close(outfile);
    exit(0);
}
```

3.3 Task 3 (3 marks)

By modifying the program that you have understood in Task 2 (3.2), write a C program that reads the records from the `recordfile` in the following order: the second record (1), the last or fourth record (3), the first record (0), and the third record (2). Then print the records out on the terminal screen.

Don't forget that you have to first open the `recordfile` for reading; and use the low-level `read()` to read each of the records.

(**Note:** As long as your program is able to print out the username of each record, you have achieved the task given in this last section.)