Laboratory 9

Practical Software Design

9.1 Overview

This laboratory looks at the process of designing software and documenting the development process in the form of a report. The lab is split into two parts:

- the first part of the lab (sections 9.3 and 9.4) gives you an existing design for a piece of software. This is exactly the kind of design that you will be expected to produce. You will implement the design in C, and then add to the design to give you practice at designing software.
- the second part of the lab (section 9.5) is a reference guide, giving you some simple tips to help you write your own software development reports.

Section 9.2 will introduce you to the lab, and the software development process.

9.2 The Structure of Software Projects

This laboratory is a bit different to the ones you have done so far. In this lab, rather than look at another part of the C programming language, we will look at how you should use the C that you already know about in a formal software development process.

The best way to produce good quality software is to follow a well structured software engineering approach. A formal approach to the development process helps to prevent errors. A small error in an early stage of the process can grow to become a large problem in the finished program. Such large problems tend to be very expensive and time-consuming to fix.

The software engineering process is usually broken down into the following steps:

- understanding the problem so that a set of **Requirements** may be created. This stage is often called *Requirements Capture* or *Requirements Elicitation*.
- analysing the requirements to understand how the problem may be solved. This stage is usually referred to as *Requirements Analysis* or just **Analysis**. The aim is to gather information ready to produce a formal specification for what the software should do.
- the creation of a formal **Specification**. A specification is typically split into things that the software is *required* to do, and things that are *desirable* for the software to do.
- using the specification as the starting point for a software **Design**. There are many approaches to designing software but the aim is always the same: to decide exactly how the software will work, and how it will be used *before* any actual programming takes place.
- the **Implementation** of the program. In your case this would be done in the C programming language. This is the only part of the process that actually involves programming. The way in which the design is implemented is documented in an **Implementation Report**.

- the software that has been produced must be carefully tested against the original specification and its functionality verified against the original requirements. There are a number of ways to approach the **Testing and Verification** of software. Usually more than one method is used to improve confidence in the correct operation of the program.
- before the software is passed on to a user, or the original customer or client, the operation of the program is documented in a **User Manual**.

An important part of the software engineering process is the documentation of each stage. In the list above, the words in **bold** indicate the names of each section of a typical software report. A high quality of documentation is essential when software is being developed by a team of engineers rather than a single programmer. Section 9.5 is a reference guide for software report writing, giving you some tips for producing your own programs and reporting on the development process.

Rather than spend a lot of time in this lab doing paper work you will be learning about program design by using a design which has already been written. This lab script will give you the specification and design for a piece of software, assuming that the requirements and analysis have already been done. They are not given here. You will have to implement the software design you have been given. Doing this will help you understand what is necessary in a good software design. This should help you when you come to write your own software reports.

There are two designs in this script: one is for the graphics option, the other is for the music option. You should attempt the one you are most familiar with.

9.3 Graphics Option

This design takes the example of a simple piece of software intended to behave like a simple drawing device (an "Etch-a-Sketch").

9.3.1 Specification

The program is required to:

- allow the user to draw a line using the arrow keys;
- allow the user to clear the drawing and start again;
- drawing should start from the centre of the screen;
- allow the user to exit at any time.

Additionally, it would be desirable if the program:

- ensures that the drawing does not leave the visible screen area;
- is easy and straightforward to use.

9.3.2 Design

The design will be split into three stages: the user interface design, a structural design and an algorithm design. This process roughly corresponds to a top-down design approach, which starts from what the program should appear to do and works towards the question of how that functionality should be achieved.

User Interface Design

The program will start by welcoming the user, informing them how to use the program and then displaying the graphics window in which they can draw.

At program start up, the following text should be displayed:

```
Welcome to SketchPad

Arrow keys let you draw
Press 'c' to clear your drawing
Press 'q' to quit

Press any key to start drawing...
```

The program should then wait for a key press.

Once the user has pressed any key the graphics window should be displayed. To ensure that the window fits comfortably onto most people's screens, the graphics window should have a drawing area of 640 by 480 pixels. To allow these numbers to be changed easily they should be defined as symbols:

Symbol Name	Value
`WINDOW_SIZE_X`	640
`WINDOW_SIZE_Y`	480

Once the graphics screen is displayed, the drawing may begin. Drawing should begin at the centre of the screen, i.e. at the location (`WINDOW_SIZE_X` / 2, `WINDOW_SIZE_Y` / 2).

When the user presses a key it should be handled as follows:

- the 'up' arrow key should cause a line to be drawn from the current location upwards for a small amount;
- the 'right' arrow key should cause a line to be drawn from the current location right for a small amount;
- the 'down' arrow key should cause a line to be drawn from the current location downwards for a small amount;
- the 'left' arrow key should cause a line to be drawn from the current location left for a small amount;
- the 'c' key should cause the drawing to be cleared and the drawing location reset to the middle of the screen;
- the 'q' key should cause the program to exit.

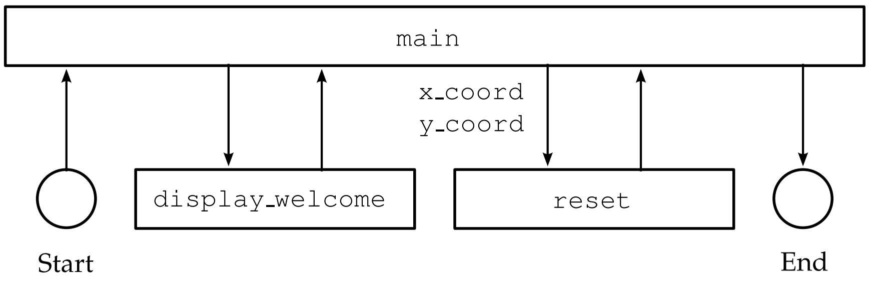
All key presses should be case insensitive. Once a line has been drawn, the "current location" will be the end of the new line. The amount that a line is drawn by should be small, but not too small, 10 pixels is reasonable in a 640 by 480 pixel window. To allow this amount to be changed easily, it should be defined as a symbol:

Symbol Name	Value
`MOVE_DISTANCE`	10

Program Structure Design

The program can be split up into separate functions corresponding to the different tasks the program must perform. Separate functions are used for programming clarity and to enable code reuse.

The structure of the program is shown in the diagram below:



Each of the structural units in the diagram is a function. These functions are documented in the table below:

Function Name	Input Parameters	Return Value	Description
`main`	(None)	Integer, error level passed to environment	Driving function, calls all others, always returns 0. Handles the actual drawing.
`display_welcome`	(None)	(None)	Displays the welcome message and instructions to the user
`reset`	x-coordinate (Integer) y-coordinate (Integer)	(None)	Clears the graphics window and sets the current location to the x- and y-coordinate specified

Program Algorithm Design

This section examines each of the functions identified by the structural design and outlines an algorithm for its implementation. All algorithms conform to the data conventions identified in the previous section.

The `main` function is responsible for carrying out most of the program's functionality. It calls the other functions, whenever necessary, to display the welcome message and reset the graphics window. The most important role of the `main` function is the processing of key input from the user and the drawing of lines in the graphics window.

The `main` function uses four variables:

Variable	Range	C Type	Description
`curr_x`	0 – 640	`int`	The x-coordinate of the current drawing location
`curr_y`	0 – 480	`int`	The y-coordinate of the current drawing location
`leave`	0 or 1	`int`	0 indicates that the program should continue running, 1 indicates the program should finish
`key`	←, ↑, →, ↓, 'c', 'C', 'q', 'Q'	`int`	The number of the key that was pressed by the user

The structure for the `main` function algorithm is as follows:

```
display welcome message
initialise graphics window
set initial position to be centre of screen
set leave to be zero
begin loop
    get a key press from the user
    if it was an arrow key, set current position correctly
    if it was 'c' or 'C' set the current position to be the centre of screen
    check that the position does not fall off the edge of the window
    if it does, adjust so that it is within the window
    if the key was an arrow key draw a line to the new current position
    if the key was 'c' or 'C' clear the screen and reset the position to centre
    if the key was 'q' or 'Q' set leave to be 1
loop while leave is zero
close the graphics window
```

The detailed algorithm for the `main` can be described in pseudo-code as follows:

```
function main
    call display_welcome function
    initialise the graphics window to WINDOW_SIZE_X, WINDOW_SIZE_Y
    set curr_x to be WINDOW_SIZE_X / 2
    set curr_y to be WINDOW_SIZE_Y / 2
    call reset specifying curr_x and curr_y
    set leave to be 0
    begin loop
        call getch to get a key from the user, put the result in key
        if key is zero then
            call getch again, putting the result in key
        end if
        switch using key
            case up arrow key
                subtract MOVE_DISTANCE from curr_y
            case left arrow key
                subtract MOVE_DISTANCE from curr_x
            case down arrow key
                add MOVE_DISTANCE to curr_y
            case right arrow key
                add MOVE_DISTANCE to curr_x
            case 'C' or 'c'
                set curr_x to be WINDOW_SIZE_X / 2
                set curr_y to be WINDOW_SIZE_Y / 2
        end switch
        if curr_x is greater than 640
            set curr_x to be 640
        else if curr_x is less than zero
            set curr_x to be zero
        end if
        if curr_y is greater than 480
            set curr_y to be 480
        else if curr_y is less than zero
            set curr_y to be zero
        end if
        switch using key
            case any arrow key
                draw a line to the new curr_x and curr_y position
            case 'C' or 'c'
                call reset specifying curr_x and curr_y
            case 'Q' or 'q'
                set leave to be 1
        end switch
    loop while leave is zero
    close the graphics window
end function main
```

The `main` function always returns 0.

The two other functions are considerably simpler. The `display_welcome` function simply displays the welcome message and then waits for a key press. The pseudo-code is as follows:

```
begin function display_welcome
```

```
    display welcome message
    call getch to wait for a key press
end function display_welcome
```

The `display_welcome` function has no variables or parameters.

The `reset` function takes two parameters:

Parameter	**Range**	**C Type**	**Description**
`x`	0 – 640	`int`	The x-coordinate that the drawing location should be set to
`y`	0 – 480	`int`	The y-coordinate that the drawing location should be set to

The `reset` function clears the graphics window and sets the current location to the x- and y-coordinate specified by the parameters. The pseudo-code is as follows:

```
begin function reset
    call cleardevice to clear graphics window
    call moveto passing x and y parameters
end function reset
```

The `reset` function has no variables.

This completes the design.

9.3.3 Implementation

You should now take the time to make sure that you understand how the design is proposing to solve the problem. The pseudo-code for the `main` function is quite complicated. Have a think about how you are going to translate it into C.

Change the target to "lab9". You will find that the "lab9.c" source file is completely empty. This is ready for your own code.

Exercise 9.1: Implementing the Design

Implement the 'SketchPad' program according to the design. You will need to choose appropriate C statements that match with the pseudo-code. You will also need to make sure that you `#include` the correct header files.

Exercise 9.2: Checking Your Implementation

Test your implementation to make sure that it works. Does it do what you expected it to?

Try out an implementation that a friend has done (you could let them try yours). How similar is it? If it is different, why is it different?

If a design is good then different implementations of it should be indistinguishable to a user.

Do you think the design could be improved? If so, how?

9.3.4 Extending the Design

You now have the chance to add to the design. Although you are not producing a software report for this program, you should make some notes as if you were writing a proper design. This gives you the chance to practice writing a small part of a design.

Exercise 9.3: Adding to the Design

Imagine that the specification contained an extra point: The program is required to:

- allow the user to change between red, green, blue and white drawing colours.

Make notes on how you would amend the design to meet this new part of the specification. Make sure that each decision you take is fully justified.

Exercise 9.4: Implementing Your Design Addition

Follow your design, and implement the additions you have just made. How easy is it to follow your design? Are you having to correct design mistakes whilst you program? Do you think that you missed anything in your design?

9.4 Music Option

This design takes the example of a simple piece of software intended to make the QWERTY (computer) keyboard behave like a basic musical keyboard with only one octave.

9.4.1 Specification

The program is required to:

- allow the user to play any note from an octave;
- use the octave that starts on middle-C;
- each note should be activated from a key press on the QWERTY keyboard;
- each note should be played for a short, but fixed amount of time;
- allow the user to exit at any time.

Additionally, it would be desirable if the program:

- uses keys on the keyboard that roughly correspond in layout to a musical keyboard;
- is easy and straightforward to use.

9.4.2 Design

The design will be split into three stages: the user interface design, a structural design and an algorithm design. This process roughly corresponds to a top-down design approach, which starts from what the program should appear to do and works towards the question of how that functionality should be achieved.

User Interface Design

The program will start by welcoming the user and informing them how to use the program. The program will then play notes in response to key presses, until the user requests that the program should exit by pressing the 'q' key.

At program start up, the following text should be displayed:

```
Welcome to KeyNote
Press 'q' to quit

The keyboard is the following keys:
[a] C
   [w] C#
[s] D
   [e] Eb
[d] E
[f] F
   [t] F#
[g] G
   [y] G#
[h] A
   [u] Bb
[j] B
[k] C
```

The program should then wait for a key press.

When the user presses a key that is part of the list displayed, the corresponding note should play for half a second (500ms). To allow this duration to be changed easily, it should be defined as a symbol:

Symbol Name	Value
`NOTE_DURATION`	500

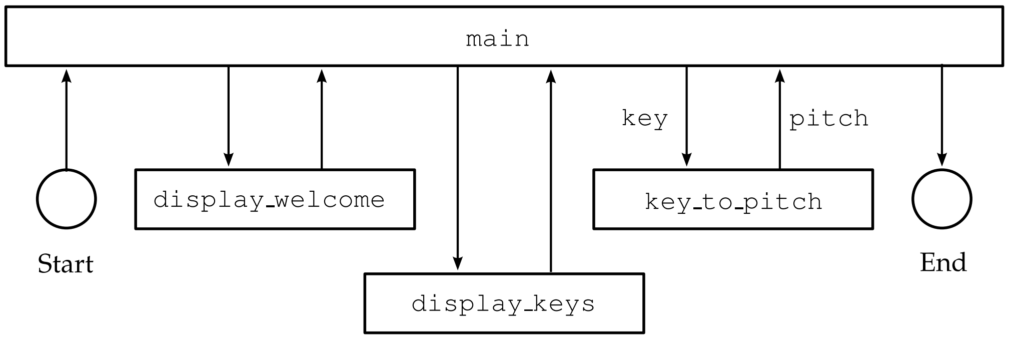
For example, when a user presses the 'a' key, a middle-C should be played for half a second. When the user presses the 't' key, the F# above middle-C should be played for half a second. All key presses should be case insensitive, i.e. 'a' and 'A' should be treated identically.

If the user presses the 'q' key, the program should end.

Program Structure Design

The program can be split up into separate functions corresponding to the different tasks the program must perform. Separate functions are used for programming clarity and to enable code reuse.

The structure of the program is shown in the diagram below:



Each of the structural units in the diagram is a function. These functions are documented in the table below:

Function Name	Input Parameters	Return Value	Description
`main`	(None)	Integer, error level passed to environment	Driving function, calls all others, always returns 0
`display_welcome`	(None)	(None)	Displays the welcome message to the user
`display_keys`	(None)	(None)	Displays the instructions for how to use the keyboard to the user
`key_to_pitch`	key code (Integer)	Integer, the pitch of the note to play or zero if the key was not valid or -1 if the program should exit	Converts a key number to a pitch, or to zero if the key wasn't valid, or to -1 if the key was 'q' or 'Q' and the program should exit

The `key_to_pitch` function translates a key numbers (like that returned from `getch`) into a pitch. If the key was not a valid note key it returns zero. If the key was 'q' or 'Q' it returns -1. To allows these numbers to be easily read and recognised symbols should be defined for them as follows:

Symbol Name	Value
`PITCH_INVALID`	0
`PITCH_QUIT`	-1

The `main` contains the main loop which keeps the program running. Its most important role is to accept key presses from the user and then to call the `key_to_pitch` function to translate the key numbers into a pitch value. It then reacts to the pitch number that is returned and plays a note as necessary.

The `main` function uses three variables:

Variable	Range	C Type	Description
`leave`	0 or 1	`int`	0 indicates that the program should continue running, 1 indicates the program should finish
`key`	'a', 'A', 'w', 'W', 's', 'S', 'e', 'E', 'd', 'D', 'f', 'F', 't', 'T', 'g', 'G', 'y', 'Y', 'h', 'H', 'u', 'U', 'j', 'J', 'k', 'K', 'q', 'Q'	`int`	The number of the key that was pressed by the user
`pitch`	60 – 72, 0, -1	`int`	The pitch of the note to play or zero if the note should not be played, or -1 if the program should exit

The detailed algorithm for the `main` can be described in pseudo-code as follows:

```
function main
    call display_welcome function
    call display_keys function
    set leave to be 0
    begin loop
        call getch to get a key from the user, put the result in key
        if key is zero then
            call getch again, putting the result in key
        end if
        call key_to_pitch passing key, assigning the result to pitch
        if pitch is equal to PITCH_QUIT
            set leave to be 1
        else if pitch is not equal to PITCH_INVALID
            turn on a midi note of pitch, on channel 1 with velocity 64
            pause for NOTE_DURATION
            turn off a midi note of pitch, on channel 1
        end if
    loop while leave is equal to 0
end function main
```

The two display functions are very simple. The `display_welcome` function simply displays the first part of the welcome message. The pseudo-code is as follows:

```
begin function display_welcome
    display "Welcome to KeyNote"
    display "Press ’q’ to quit"
end function display_welcome
```

The `display_welcome` function has no variables or parameters.

The `display_keys` function displays the list of keys with their corresponding notes (see earlier in the design).

```
begin function display_keys
    display list of keys and notes
end function display_welcome
```

The `display_keys` function has no variables or parameters.

The `key_to_pitch` takes one parameter:

Parameter	**Range**	**C Type**	**Description**
`key`	‘a’, ‘A’, ‘w’, ‘W’, ‘s’, ‘S’, ‘e’, ‘E’, ‘d’, ‘D’, ‘f’, ‘F’, ‘t’, ‘T’, ‘g’, ‘G’, ‘y’, ‘Y’, ‘h’, ‘H’, ‘u’, ‘U’, ‘j’, ‘J’, ‘k’, ‘K’, ‘q’, ‘Q’ (Other values should be handled as invalid)	`int`	The number of the key that should be translated to a pitch

The `key_to_pitch` also has one variable:

Variable	**Range**	**C Type**	**Description**
`pitch`	60 – 72, 0, -1	`int`	The pitch of the note to play or zero if the note should not be played, or -1 if the program should exit

The way in which the `key_to_pitch` function works is very simple, but quite long. It simply recognises each of the valid key numbers and translates them to a pitch. The pseudo-code is shown below:

```
begin function key_to_pitch
    switch using key
        case 'a' or 'A'
            set pitch to be 60
        case 'w' or 'W'
            set pitch to be 61
        case 's' or 'S'
            set pitch to be 62
        case 'e' or 'E'
            set pitch to be 63
        case 'd' or 'D'
            set pitch to be 64
        case 'f' or 'F'
            set pitch to be 65
        case 't' or 'T'
            set pitch to be 66
        case 'g' or 'G'
            set pitch to be 67
        case 'y' or 'Y'
            set pitch to be 68
        case 'h' or 'H'
            set pitch to be 69
        case 'u' or 'U'
            set pitch to be 70
        case 'j' or 'J'
            set pitch to be 61
        case 'k' or 'K'
            set pitch to be 72
        case 'q' or 'Q'
            set pitch to be PITCH_QUIT
        any other case
            set pitch to be PITCH_INVALID
    end switch
    return from the function with pitch
end function display_welcome
```

This completes the design.

9.4.3 Implementation

You should now take the time to make sure that you understand how the design is proposing to solve the problem. The way that the `main` function uses the special pitch values `PITCH_INVALID` and `PITCH_QUIT` is quite complicated. Have a think about how this translates into C.

Change the target to "lab9". You should find that the "lab9.c" source file is completely empty. This is ready for your own code.

Exercise 9.5: Implementing the Design

Implement the 'KeyNote' program according to the design. You will need to choose appropriate C statements that match with the pseudo-code. You will also need to make sure that you `#include` the correct header files.

Exercise 9.6: Checking Your Implementation

Test your implementation to make sure that it works. Does it do what you expected it to?

Try out an implementation that a friend has done (you could let them try yours). How similar is it? If it is different, why is it different?

If a design is good then different implementations of it should be indistinguishable to a user.

Do you think the design could be improved? If so, how?

9.4.4 Extending the Design

You now have the chance to add to the design. Although you are not producing a software report for this program, you should make some notes as if you were writing a proper design. This gives you the chance to practice writing a small part of a design.

Exercise 9.7: Adding to the Design

Imagine that the specification contained an extra point: The program is required to:

- allow the user to change the volume at which the notes are played using the up and down arrow keys.

Make notes on how you would amend the design to meet this new part of the specification. Make sure that each decision you take is fully justified.

Exercise 9.8: Implementing Your Design Addition

Follow your design, and implement the additions you have just made. How easy is it to follow your design? Are you having to correct design mistakes whilst you program? Do you think that you missed anything in your design?

9.5 Writing Your Own Report

This section contains some very brief tips to help you write your own software reports. These points should reinforce the material you have covered in the lectures. Each of the sections below is intended to reflect a section of your software report.

9.5.1 Requirements

The Requirements section should state the problem you have been given, in its entirety. You can imagine that the problem you have been given is a problem from a customer, or client. Quite often you will have requirements that are not explicitly stated in the problem. In your case these requirements arise because you are doing a course on C programming. Even though these requirements are implicit, you should still state them. They should include:

- the fact that you are expected to use the C programming language;

- the platform that you are expected to program for (a PC running Microsoft Windows);
- the graphics or MIDI library you are expected to use.

9.5.2 Analysis

Your Analysis section should take the Requirements section as its starting point. You should then attempt to analyse the requirements to extract more information from them. This is where you should flesh out the requirements with assumptions of your own, if you have to.

The most important rule is that when you are doing an analysis you must treat the problem as a *black box*. Treating something as a black box means that you should only be concerned with the the problem itself, and its effects, **not how you will solve the problem**. You are aiming to gather information so that you can put together a concrete specification for what the software should do.

The main things you should be concerned with in an analysis are:

- The inputs to the problem including units and expected range.
- The outputs the solution will **have** to produce including units and expected range.
- What kind of screen output is implied by the specification; will it be graphics or text?
- What size screen (especially for graphics) is reasonable on your platform?
- What is the available range of midi notes/channels?
- All formulae your program will use including calculation formulae and others such as physics, tempo conversion etc.
- State how the results of your program will appear/sound, in general terms.
- Who is your user? What are their needs? Are they technically or musically literate? Are they computer literate?

As a general rule of thumb, it shouldn't matter that you are going to implement your program in C. The design should be independent of programming language. This means that it should not contain C terms like `int`, `double` or functions like `scanf` or `midi_note`.

Be careful not to begin your design in your analysis. You cannot begin a design until you have a formal specification.

9.5.3 Specification

You should use your analysis to put together a list of **exactly** what your program needs to do. It is very useful if you split this list into things that your program **must** do, and what it would be **desirable** if your software did.

You may find it helpful if you number the points in your specification so that you can refer to them in later stages of your report.

9.5.4 Design

As you will have seen in this laboratory, your design is where you make all the major decisions about **how** your program will work. All these decisions should be made before you begin writing source code because you have a far better understanding of how the different pieces of your program fit together at the design stage.

A good design will be very detailed and should leave very few decisions to the programmer who implements the design. If you gave your design to someone else to code as well as yourself, a user should not be able to tell the difference between the two versions. No design decision should appear arbitrary — they should all be linked to the points in your specification.

Here are some guidelines for things that you should include in your design:

- Describe your user interface in detail including exact coordinates of all lines on the graphics screen (if appropriate), exact wording of text the user will see and how, when and where this will appear.
- Describe all acceptable user input and when it should be accepted by the program.
- Include a detailed structure diagram showing data flow between functions or parts of your program.
- Give a description of all algorithms used, including how you will validate user input.
- Specify a data table showing the name, type and value range of each variable you will use.

There should be a logical flow through your design, starting from the specifications and ending up at a full design for the program.

9.5.5 Source Code — Implementing the Design

You will be expected to include the full source code to your software as part of a software report. This is so that someone reading your report can understand how your software works and how they might add to it or change it.

Your code should be well commented. Assume the reader of your code understands C. Comments should explain why something is happening, not what is happening. They way you write code can also help a reader understand it. Make sure that you use variables and programming statements in a logical way. It helps if you choose logical names for variables and functions.

The way that you choose to lay out functions in your source files can heavily influence how logical your program seems to the reader. It is often best to put the `main` function first, in a C file with the same name as the program.

9.5.6 Implementation Report

Your source code should be supported by an Implementation Report. This section tells the reader how the different elements of the design have been turned into C.

You should include in your implementation report:

- decisions you have taken when implementing the design, such as what files you have used and what functions are in them;
- the way that your functions in different files interact, along with information about any header files that you have created;
- documentation and justification for any changes that you have made to the design whilst doing the implementation.

Someone should be able to understand your source code, and how it works, by reading your design, implementation report and the source code itself.

9.5.7 Verification and Testing

The Verification and Testing section of your report should aim to prove that your software functions correctly. This means that it should always react appropriately to user input, it should meet the specification that you set out after your analysis and it should satisfy the requirements that were originally given to you.

Testing software is a very complicated process. A perfect testing process, or *regime*, would test your software against all the possible inputs it could ever take, in all the conditions it all the conditions it could ever be run, making sure that the outputs are appropriate. In all but the most simple of cases, this is infeasible. It would just take too long. So most software testing is a compromise.

With your own testing, you must decide, and justify, what testing is necessary to prove that your software works. You should also comment on how you think your testing regime compares to the ideal in which every part of your program is tested. How much of your program has been tested is often called the test *coverage*.

Your testing and verification report should:

- describe your testing strategy to show that your program works correctly under all conditions;
- explain why you think you strategy is sufficiently comprehensive;
- include all test input data and test results and comment on your test coverage;
- detail any modifications you made following test failure and show the results of re-testing.

9.5.8 User Manual

You can imagine that most of your software report is written for someone who understands software development in C and wants to understand your program. The user manual, on the other hand, is written for the user that you identified in your analysis. It may be that this user is not technically literate. That way that your user manual is written should reflect the type of person you think will be using your program.

Your user manual should include:

- installation instructions — this may just be copying the executable file for your program;
- user and system requirements — you should say what type of computer is necessary to run the program and what kind of expertise you are expecting on behalf of the user;
- full usage instructions for all your programs features, ideally with examples;
- you might also like to include answers to frequently asked questions (FAQs).

9.6 Summary

Now that you have finished this laboratory you should have a good idea of what needs to be in a software design, and how you might go about writing one yourself.

You should also know that the reference section of this lab will be useful to you when you come to write your own software reports.