Dublin City University

School of Electronic Engineering

Module: EE105

Software Engineering 2

Module Notes

Academic Year 2005/2006: Semester 2

Module Co-ordinator: Dr. Cristina Hava Muntean
Room: S220
Telephone: 700-7645
Email: ee105@eeng.dcu.ie

© 1994-2000 Dr. Barry McMullin
© 2000-2003 Dr. Noel E. O’Connor
©2003-2005 Dr. Gabriel-Miro Muntean
©2005-2006 Dr. Cristina Hava Muntean
## Contents

1. **Introduction**  
   1.1 Module Overview  
   1.2 Instructors  
   1.3 Textbook  
   1.4 Delivery  
   1.5 Tutorial  
   1.6 Examination  
   1.7 The Email Conference: ee105-talk  
   1.8 On-Line Resources  

2. **The C Standard Library (libc)**  
   2.1 On Re-inventing Wheels  
   2.2 Mechanics  
   2.3 Implementation-defined Limits: limits.h  
   2.4 Error Conditions: errno.h  
   2.5 Utilities: stdlib.h  
   2.6 Input and Output: stdio.h  
   2.7 Diagnostics: assert.h  
   2.8 Character Class Tests: ctype.h  
   2.9 String Functions: string.h  

---

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Module Overview</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Instructors</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Textbook</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Delivery</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Tutorial</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Examination</td>
<td>7</td>
</tr>
<tr>
<td>1.7 The Email Conference: ee105-talk</td>
<td>7</td>
</tr>
<tr>
<td>1.8 On-Line Resources</td>
<td>7</td>
</tr>
<tr>
<td>2.1 On Re-inventing Wheels</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Mechanics</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Implementation-defined Limits: limits.h</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Error Conditions: errno.h</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Utilities: stdlib.h</td>
<td>11</td>
</tr>
<tr>
<td>2.6 Input and Output: stdio.h</td>
<td>11</td>
</tr>
<tr>
<td>2.7 Diagnostics: assert.h</td>
<td>17</td>
</tr>
<tr>
<td>2.8 Character Class Tests: ctype.h</td>
<td>17</td>
</tr>
<tr>
<td>2.9 String Functions: string.h</td>
<td>17</td>
</tr>
</tbody>
</table>
int strcmp(char *s, char *t) .................................................. 18

2.10 Mathematical Functions: math.h ........................................... 18
double sin(double x) .............................................................. 18
double cos(double x) .............................................................. 18
double tan(double x) .............................................................. 18
double asin(double x) .............................................................. 18
double acos(double x) .............................................................. 18
double atan(double x) .............................................................. 18
double atan2(double y, double x) .............................................. 18
double exp(double x) .............................................................. 18
double log(double x) .............................................................. 18
double log10(double x) ........................................................... 18
double pow(double x, double y) .................................................. 18
double sqrt(double x) .............................................................. 18

2.11 Conclusion ............................................................................ 18

3 Defining C Functions ................................................................ 19
3.1 Specification .......................................................................... 19
3.2 Coding ................................................................................... 20

4 Pointers and Structures in C ................................................... 21
4.1 Pointers .................................................................................. 21
4.2 Structured Data Types ........................................................... 22

5 Laboratory Report Guidelines .................................................. 24
5.1 General .................................................................................. 24
5.2 Typing Up .............................................................................. 24
5.3 Structure ............................................................................... 25
5.3.1 Heading ............................................................................ 25
5.3.2 Plan .................................................................................. 25
5.3.3 Development and Test ....................................................... 25
5.3.4 Conclusion ................................................................. 26
5.4 Including Source Files .......................................................... 26
5.5 Problems ................................................................................. 27
5.6 Presentation ........................................................................... 28
5.7 Marking Scheme ................................................................. 29
6 Laboratory Notes

6.1 General Comments .......................................................... 30
6.2 Command Line Redirection .............................................. 30
6.3 Session 1: Trigonometry ................................................... 31
   6.3.1 Exercise 1: The \( \sin() \) Function (20%) ......................... 31
   6.3.2 Exercise 2: Angle Conversion (20%) .............................. 31
   6.3.3 Exercise 3: Function Definition (20%) .......................... 32
   6.3.4 Exercise 4: Refinement (20%) ..................................... 32
   6.3.5 Exercise 5: Finale (20%) ............................................ 32
6.4 Session 2: Encryption ...................................................... 32
   6.4.1 Exercise 1: The Caesar Cipher (50%) ............................ 32
   6.4.2 Exercise 2: Cryptanalysis (20%) .................................. 33
   6.4.3 Exercise 3: The Vigenère Cipher (30%) ......................... 33
   6.4.4 What next? ............................................................. 33
6.5 Session 3: Projectiles ..................................................... 33
   6.5.1 Problem Specification ............................................... 33
   6.5.2 Hints ................................................................. 34
6.6 Session 4: Letter Frequencies .......................................... 35
   6.6.1 Exercise 1: Counting A! (30%) .................................. 35
   6.6.2 Exercise 2: More Counting... (30%) ............................ 35
   6.6.3 Exercise 3: Statistics (20%) .................................... 35
   6.6.4 Exercise 4: Plotting a Histogram (20%) ....................... 35
6.7 Session 5: Exploring Pointers ......................................... 36
   6.7.1 Exercise 1: Basic Pointer Manipulation (40%) ............... 36
   6.7.2 Exercise 2: Advanced Pointer Manipulation (60%) ........... 36

7 General Examination Instructions ......................................... 38

7.1 Overview ........................................................................ 38
7.2 Regulations .................................................................... 38
7.3 Preparation ..................................................................... 38
7.4 Conduct of the Exam ....................................................... 38
7.5 Equipment Failure .......................................................... 39
7.6 Report Preparation .......................................................... 39
7.7 Completion ..................................................................... 39
7.8 Links to the Specific Exam Exercises ................................. 39

A The ASCII Character Set ...................................................... 41

B Skeleton Report File ............................................................ 42
C Program Listings

C.1 sum0.c ....................................................... 45
C.2 sum1.c ....................................................... 46
C.3 sum2.c ....................................................... 47
C.4 ptr0.c ....................................................... 48
C.5 ptr1.c ....................................................... 49
C.6 ptr_net.c ..................................................... 50
Chapter 1

Introduction

This is the documentation for the module EE105 Software Engineering 2, for the session 2005/2006. This is a required module for all students enrolled for the following academic programmes offered in Dublin City University:

- B.Eng. (Electronic Engineering) Year 1
- B.Eng. (Telecommunications Engineering) Year 1
- B.Eng. (Digital Media Engineering) Year 1
- Common Entry to Electronics and Communications Engineering Year 1
- B.Eng. (Mechatronic Engineering) Year 1
- B.Eng. (Information and Communications Engineering) Year 1
- B.Eng./M.Eng. (Electronic Systems) Year 2

This module follows directly from module EE102 Software Engineering 1. It is assumed here that you are already familiar with all the material presented in that module.

These notes are also available online at:

The online version may be incrementally revised and extended as the semester progresses.

1.1 Module Overview

This module builds on the introductory materials already presented in module EE102 Software Engineering 1, focusing on software development skills using the C language. In particular, the module introduces the concepts of functional decomposition (“sub-programs”), parameter passing (by value and by reference), structured data types, pointers, and the facilities of the C standard library. Example application areas include numerical analysis and cryptography.

The module is still preparatory in nature, providing necessary foundations for a variety of modules in subsequent stages of the relevant programmes.

For further general information, including the syllabus, check the formal Module Specification.

1.2 Instructors

The Module Co-ordinator is Dr. Cristina Hava Muntean. There will also be two laboratory demonstrators and a tutor assigned to the module.

1.3 Textbook

You must have a textbook on the C programming language (not C++) to participate in this module. The online notes for the module EE102 Software Engineering 1 suggest some particular books that would be suitable.

It is recommended that the textbook be brought along to every lab session.

1.4 Delivery

The module is delivered via lectures (two per week), laboratory sessions (one three hour session every second week), tutorials, and is supplemented by various additional online resources.

1.5 Tutorial

In addition to lectures and lab sessions, a one hour tutorial session will be offered each week. Timetable details will be announced when they are available. A
tutor and/or instructor will be present at each tutorial session. You can also, of course, submit queries by e-mail at any time (see the email conference).

1.6 Examination

The module will be examined by way of a supervised laboratory examination at the end of the semester.

The format of the examination will be similar to the format of the lab exercises. Each student will work alone. Instructions will be accessed online (i.e., via the Web). The work to be done will be derived from work done during the normal lab sessions—though it will, of course, differ in detail. A report will be required, submitted electronically. This report will form the basis for marking the examination. See the chapter on General Examination Instructions for more information.

1.7 The Email Conference: ee105-talk

The module has an associated email conference, called ee105-talk. All students enrolled in the module are automatically subscribed to this conference. That means that you will receive, by email, every message contributed by any member of the conference. The messages are also automatically archived on the Web at URL:

http://list.eeng.dcu.ie/pipermail/ee105-talk/

You can contribute to the conference by addressing an email message to:

ee105-talk@list.eeng.dcu.ie

Messages related to the EE105 module only are accepted in this email conference.

In exceptional circumstances, if you have a query relating to the module, but which you do not wish to raise in the public forum of the ee105-talk conference, you can address private email to me at:

ee105@eeng.dcu.ie

1.8 On-Line Resources

The following additional materials are available online:

- **Module Specification:**
  

  This is the formal specification of this module, approved by the relevant academic programme boards. It includes the official syllabus for the module.

- **C Source File Directory:**
  

  This directory provides access to all C source files introduced in the lectures or labs, as the module progresses.

  The Borland C++ Compiler (bcc) is recommended for use with this module. This is a high performance compiler which is available free for Windows platform.

  Borland C++ 5.5 Compiler is a comprehensive package, with many additional tools over and above the compiler. This software is available on the Borland web site.

  http://www.borland.com/bcppbuilder/freecompiler/

  A brief description on how to use Borland C++ environment is provided in Appendix D.

  For text editing of all sorts (C source files, lab reports, etc.) I recommend TextPad. This is available on Windows platforms and it is already installed on all the machines from the lab.

  There is a very wide range of other resources available elsewhere on the Internet, on the general subject of C programming. Pointers to many of these are available through the Yahoo indexing service at URL:

  http://www.yahoo.co.uk/

  For example you could check the section on "C Programming".

  The following site provides a nice informal and short introduction to C programming:

  http://www.strath.ac.uk/IT/Docs/Ccourse/

  If you identify any other particularly good (or bad!) Web resources, please post the details on the module email conference ee105-talk.
Chapter 2

The C Standard Library (libc)

2.1 On Re-inventing Wheels

It is well known that one should not waste time “re-inventing the wheel”. In Engineering, this means not redesigning something one has already got a perfectly satisfactory design for. In Software Engineering, it means not rewriting software to perform operations that one has previously written (and tested!) software for.

In the case of the C language there are a range of things that programmers very frequently wish to do, which are so common that standard software for the purpose is actually distributed along with every C compiler. This software is called the C Standard Library. You are more or less guaranteed that every C compiler will come with an implementation of the Standard Library. It is important to be aware at least of the existence of the Standard Library, and to have some outline idea of the software contained in the Library. In this way you can avoid unnecessarily writing and testing software which is already effectively available, in a well tested form, for free.

The Borland C++ Compiler (bcc) is recommended for use with this module. This is accompanied by the Standard C Library. Detailed documentation for the latter is available at URL:

www.cppreference.com

This chapter is intended only to provide an introduction to a very small subset of the Standard C Library, and is not a substitute for the full documentation at the URL above.

2.2 Mechanics

In order to use the facilities of the Standard Library, one must have some understanding of the mechanics of how the compiler will access it.

Some very simple C programs can be completely “self-contained”: all the code for all the functions making up the program is contained in a single program source file. In general, however, this need not be the case; in particular, it will not be the case if you wish to make use of the Standard Library.

In principle, of course, the Standard Library could be distributed simply as a set of one or more C source code files. You would then use a library function simply by copying the source code out of the relevant library file and pasting it into your own program file. However, this is not the mechanism which is used in practise for a variety of reasons:

- Since the code for the Standard Library is rarely if ever modified, it is very inefficient to have to recompile or retranslate it every time a programmer makes a change to his own code.
- The library functions are, in some cases, not actually written in the C language at all.
- The compiler suppliers often prefer not to distribute source code for the Standard Library as this would facilitate unscrupulous pirating of their work.

What is actually done is that the Standard Library software is distributed in a sort of “canned” or precompiled form, as a set of “object files” (or, more commonly, a single “object library” file). Your program file(s) are then compiled completely separately from the source code for the Standard Library, yielding one or more further object files. Finally, all the required “object” files, both your own, and those making up the Standard Library, are combined or linked together to yield the “executable” file, which can actually be run.

To a large extent this process can be made automatic and invisible. The compiler compiles your source file(s) to the “object” code form, and then automatically links these together with the any required library object file(s).

8
However, it cannot be made completely automatic. In practise there is certain information, associated with the Standard Library, which the compiler must be made aware of at the time it is compiling your source files, in order to make sure that the object files produced will correctly link up with the object files for the Standard Library.

Since, for the most part, the Standard Library simply consists of a set of functions which you can call in your program, the most important such information is the way data is to be exchanged with each such function; i.e., how many parameters does the function take, of what types, in what order, and what type of return value does the function yield (if any)?

The compiler could, of course, try to infer this information from the way the function is actually called; but this is not always technically possible, and, in any case, it is much better if the compiler has some independent way of knowing how the function should be called, because then it can automatically check whether you have, in each case, called it correctly. It turns out that this is extremely useful, and is an effective way of automatically detecting a range of very common programming mistakes.

The compiler is given this information about how a function should be called by the use of a so-called function prototype. This is simply the “header” of the function definition, which states the function name, the return type, and the formal parameter list. But whereas, in a function definition this is then followed by a compound statement which actually defines the function, in a function prototype it is simply followed by a terminating semicolon. Thus, a prototype might look like this for example:

```c
int multiply(unsigned *multiplicand, 
             unsigned multiplier, unsigned *product);
```

This tells the compiler that the function called multiply should take three parameters, respectively of types (unsigned *), unsigned and (unsigned *), and will yield a return value of type int. Function prototypes should appear at the outermost level of a source file—i.e., not within the definition of any function.

So far, so good. It seems that if you wish to call or invoke any of the Standard Library functions in your program, you must simply insert, somewhere before the function(s) which make such call(s), a suitable function prototype, so that the compiler will then be able to decide whether the calls are correct or not.

But how are you going to know what is the correct prototype for each function in the first place?

Well, one possibility is to look up the function in the detailed technical documentation for the Standard Library: this will normally include the function prototype. You can then copy that into your own file.

However, this is clearly unsatisfactory. Apart from being laborious, it is error prone—and the possibility of an error in the function prototype underlines a primary point of using prototypes in the first place, namely that it allows the compiler to crosscheck for valid function calls! A better idea is if the prototypes are provided to you in a machine readable form—i.e., in one or more files on the computer. Then you can simply copy the required ones, and paste them into your own source files.

Well, yes, this is a major improvement, but still leaves something to be desired. For one thing, duplicating the prototypes in every source program wastes disk space. More seriously, there is always a danger that you might (accidentally) modify a prototype, again confounding the idea of allowing the compiler to automatically crosscheck the prototype against the invocation(s) of the function.

A final possibility is to leave the prototypes in one or more separate files; but have the compiler automatically access or scan the relevant files immediately before, or as part of the process of, compiling your files. In this way, there is no laborious, manual, copying of the prototypes, but no duplication and no risk of accidental modification either.

Files which are used for this kind of purpose—which contain no actual executable code, but which contain only function prototypes (and possibly other things such as symbolic constants etc.) which allow the compiler to correctly mesh the file it is compiling with some other software which has been “pre-compiled”, are called header files. Header files are normally given the extension ".h" to distinguish them from files which actually contain executable code—function definitions etc.—which will normally have the extension ".c".

It would be possible, in principle, to put all the prototypes for all the functions in the Standard Library, plus any other required information (symbolic constants etc.), in a single ".h" file, and have the compiler automatically include it in compiling any file. However, in practise this is not done for various reasons. For example, most ".c" files only involve calling a small subset of the functions in the Standard Library; it is then wasteful and time consuming for the compiler to process prototypes for all functions in the Standard Library.

The mechanism that is actually used then is as follows. A series of separate ".h" files are supplied
with the compiler. Each one provides prototypes (plus other required information) relating to only some coherent or related subset of the functions in the Standard Library. The programmer must then explicitly instruct the compiler to process just those ".h" files which are required in order to properly compile any particular ".c" file. And this is done by inserting, into the ".c" file one or more so-called #include directives. A #include directive is something much the same as a #define in the sense that it is not handled by the compiler “proper” but by the “pre-processor” which runs (automatically) immediately before the compiler. In this case, the pre-processor processes a #include by concatenating together the ".h" file and the original ".c" file, to produce a big temporary file which is what is actually then processed in the compilation phase proper.

Thus, if you want to use a function from the Standard Library, you must first look up, in the relevant technical documentation, which header file contains the prototype etc., for that function; and then insert a line something like the following in your ".c" file:

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

The angle brackets around the name of the header file tell the pre-processor to search for the file in the “standard” directories for such things; normally these will be set up when the compiler is installed, and you, as a programmer, need not worry about what these standard directories actually are.

You may, of course, need to #include several header files, depending on the particular selection of Standard Library functions you wish to use.

All required #include directives are normally placed close to the top of your ".c" file—typically either at the very top, or immediately after an initial introductory comment which documents the overall contents of the source file. Each #include must, in any case, precede any calls or invocations of the relevant functions.

The rest of this essay is concerned with introducing just a very small selection of the several hundred functions normally available in the Standard Library. It is organised into sections according to the distinct ".h" files required.

### 2.3 Implementation-defined

#### Limits: limits.h

The header file limits.h essentially just provides #define directives defining symbolic constants for the maximum and minimum values allowed with the standard C integral data types—i.e., it does not provide any function prototypes as such. It is important to be able to refer to these values in your programs in order to prevent, or at least detect, overflow situations. The values potentially differ from one C compiler to another (hence “implementation-defined”), but the symbolic constants defined in limits.h always have the same names. Thus by using the symbolic constants to refer to these limits it should be possible to write your programs in such a way that they will automatically adjust to whatever the limits actually are with any particular compiler.

Some constants which are commonly used are:

- **INT_MAX**: maximum value of `int`
- **INT_MIN**: minimum value of `int`
- **LONG_MAX**: maximum value of `long`
- **LONG_MIN**: minimum value of `long`
- **UINT_MAX**: maximum value of `unsigned int`
- **ULONG_MAX**: maximum value of `unsigned long`

This is not an exhaustive list; examine a copy of limits.h for yourself to see others. However, note that there is, of course, no need for constants called, say, UINT_MIN or ULONG_MIN since these minima are always guaranteed to be simply zero.

#### 2.4 Error Conditions: errno.h

Many functions in the standard library will detect “exception” or “error” conditions in certain circumstances—essentially if the function has been requested to do something which, for some reason, it can’t. The exact action of the function in such situations depends on the details of the particular function and the particular exception condition. However, the most usual strategy is for the return value from the function to signal, with some special value, that something has gone wrong. It is then up to the calling site to react to this in some “appropriate” way. Minimally this will mean giving some kind of overt or “visible” external signal of the problem.

In any case, while the standard library functions typically provide an initial or gross indication that something has gone wrong via the return value, it is often also useful for the calling site to have access to more detailed information which clarifies exactly the nature of the problem. This is usually achieved by having the standard library function record a detailed “error number” in a global int variable called errno. This variable is defined within the standard...
library itself: but your programs can gain access to it by a so-called `extern` declaration. This declaration is already provided in the header file `errno.h`: so if you `#include` this, you will then be able to access `errno` just as any other variable is accessed. By examining its value immediately after a call to a standard function, your program can generally establish what (if anything) went wrong.

As well as the declaration of `errno` the file `errno.h` also provides a series of `#define` directives which define symbolic names for the standard error numbers or codes which may be recorded in `errno`. This would allow your program to test for specific error codes by comparing `errno` to these symbolic values. We shall also see later how `errno` can be automatically translated into a corresponding textual error “message”, and, say, displayed, on the computer screen (see the discussion of the function `perror()`, prototyped in `stdio.h`).

### 2.5 Utilities: `stdlib.h`

The header file `stdlib` provides prototypes for a selection of miscellaneous “utility” functions, as well as a few more symbolic constants. A small selection of the more commonly used functions are detailed in the sections below.

#### `int atoi(char *s)`

This takes a string representation of an integer (i.e., a string something like “145” or “-99999” or “000000” etc.) and converts it into the corresponding internal representation of type `int`, which is the `return` value from the function.

The conversion will ignore any leading white space in the string as well as any trailing data (e.g., “ 143yu*” gets converted as “143”).

The behaviour where the answer would overflow (i.e., be greater than `INT_MAX` or less than `INT_MIN`) is generally unpredictable or indeterminate—so it is your responsibility to make sure this never arises; if the conversion simply cannot be done (e.g., `atoi("xyz")`) the `return` value will be 0.

#### `long atol(char *s)`

This takes a string representation of an integer (i.e., a string something like “145” or “-999999” or “25763178” etc.) and converts it into the corresponding internal representation of type `long`, which is the `return` value from the function. Leading white space in the string is ignored.

Again, the conversion will ignore any trailing data in the string, and the behaviour where the answer would overflow (i.e., be greater than `LONG_MAX` or less than `LONG_MIN`) is indeterminate; and if the conversion simply cannot be done (e.g., `atol("")`) the `return` value will be 0L.

#### `void exit(int status)`

This function may be called to forcibly terminate the program at any point. The parameter `status` is simply a number which is, in some sense, made available to the “external” environment of the program. In any case, if you are using `exit()` to terminate your program, you should normally just give it one of two pre-defined exit values which have been given symbolic names in `stdlib.h`: use `exit(EXIT_SUCCESS)` if the program is terminating “normally” and `exit(EXIT_FAILURE)` if it is terminating because of some unexpected or intolerable exception or error being encountered.

### 2.6 Input and Output: `stdio.h`

`stdio.h` (“standard input/output”) provides basic functions for accessing data “external” to a program. This naturally includes data in files on disk, but also covers data coming from the keyboard, or displayed on the screen, or data routed via any of the other input/output “ports” of the computer. All of these sources or destinations for data may be generically referred to as `streams`, or, more simply, `files`.

Files are classified into two kinds: `text` and `binary`. A text file is divided into “lines”, where each line has zero or more characters, and is terminated by a newline character ‘\n’. A binary file is simply a sequence of unprocessed or uninterpreted bytes, with no line organisation superimposed upon it.

Files are accessed through data structures called `file pointers`. Technically, a file pointer is the address of an object of a special type, denoted `FILE`. This type is defined by the `stdio.h` header file, and will not be recognised by the compiler unless the header file has been `#include`d.

A file pointer must be created and associated with each particular external disk file, or input/output port, before any data in that file can be accessed (“read” or “written”). This process of

---

1. In the case of programs run under DOS, the `exit()` status can be accessed via the `errorlevel` parameter in the DOS `if` command—though this is normally only used in DOS batch files.

2. In general, the terms `character` and `byte` can be regarded as almost synonymous here.
creating a file pointer and associating it with a disk file or input/output port is called opening a file, and is performed by the \texttt{fopen()} function described in more detail below. The return value from \texttt{fopen()} is the value of the file pointer, and must be stored in a suitable variable to allow the file to be accessed subsequently.

Files are opened for access in a particular mode: reading, writing, or (occasionally) both. The file is treated as a sequence of characters (or, more generally, bytes). When the file is first opened it is positioned at the very start; the contents can then be read or written in sequence until the end of the file. With certain kinds of file (namely those stored on disk) it may be possible to “reposition” the file in an arbitrary way—go back to the start, or directly to the end, etc. If a file is opened for writing then its previous contents, if any, will normally be lost.

A program may have many files open at any given time, each one associated with its own distinct file pointer.

File contents can be read, one character at a time, in sequence, with the \texttt{fgetc()} function (provided the file is in read mode); or written with the \texttt{fputc()} function (in write mode). There are also a variety of more sophisticated reading and writing functions such as \texttt{fprintf()} and \texttt{fscanf()}, which typically read or write many characters in one go, and automatically translate between external “text” representations, and internal “binary” representations, for the various native data types (\texttt{int}, \texttt{long} etc.).

When reading a file it is necessary to be able to recognise when the end of the file is encountered. This is signalled by \texttt{fgetc()} yielding a special, reserved, return value, with the symbolic name \texttt{EOF} (this value is also—somewhat misleadingly—the return value if \texttt{fgetc()} encounters any kind of error or exception condition).

Now this raises a problem: if a special value is reserved to denote the end of file condition, does this mean that this special value can never actually be stored within a file (since, once it is read, it would be mistakenly taken as signalling that the file is ended)? If so, this would be a serious restriction.

In practice, this problem is solved in a rather ingenious, but also subtle and confusing manner. A file is treated as a sequence of values of type \texttt{char}; but the return value from \texttt{fgetc()} is actually made of type \texttt{int}. Normally this is just an “encoded” or “numerical equivalent” of the \texttt{char} value read from the file. But since \texttt{int} supports more distinct values than \texttt{char} does, it is possible to encode \texttt{EOF} as an \texttt{int} value that has no “equivalent” in the \texttt{char} data type. In this way, a file can contain any arbitrary \texttt{char} values, without restriction. But, in turn, this means that it is very important that before the return value from \texttt{fgetc()} is transformed back into a \texttt{char} value (e.g., by assigning it to a variable of type \texttt{char}) it must be tested to see is it \texttt{EOF}.

For consistency with this behaviour of \texttt{fgetc()} many other functions in the Standard Library (such as \texttt{fputc()} for example) also use the \texttt{int} data type to effectively deal with \texttt{char} values, but allowing also for the special \texttt{EOF} value.

As I said, this mechanism is quite subtle; in my opinion it may be just a little bit too subtle! It involves the programmer in reyling, willy nilly, on automatic conversions between types—a practice of which I am generally severely critical. Unfortunately, this is now a \texttt{de facto} standardised way of doing things with the Standard Library, and cannot be helped at this stage. Nonetheless, it certainly helps if you at least understand what is going on.

Once the program is finished processing a particular file, the file should be closed with the \texttt{fclose()} function. This essentially involves discarding the data structure of type \texttt{FILE} which was associated with the file: it is therefore very important that, once a file is closed, the file pointer which was associated with it is not used again (since it no longer points at anything meaningful!). Files are automatically closed when a program terminates in a “controlled” fashion (either by reaching the end of the \texttt{main()} function, or by an explicit invocation of the \texttt{exit()} function).

There are three files which are opened by default, and automatically, for every program. File pointers associated with them are defined in \texttt{stdio.h}, as follows:

\texttt{stdin} : The standard input file. By default this is a read mode file, associated with the computer keyboard. However, when the program is invoked from the \texttt{bash} command line, this file may be “redirected” to be associated with, say, a disk file, using the \texttt{bash} input redirection operator “\texttt{<}”.

\texttt{stdout} : The standard output file. By default this is a write mode file, associated with the computer display screen (specifically, the \texttt{bash} window within which the program is being run). Again, however, this file may be “redirected” to be associated with, say, a disk file, using the \texttt{bash} output redirection operator “\texttt{>}”.

\texttt{stderr} : The standard error file. This is a write mode file, again associated with the computer keyboard.
display screen by default. Under bash it may be “redirected” to be associated with a disk file, using a slightly modified form of the output redirection operator “2>” (The “2” here is a somewhat obscure reference to an internal numbering scheme for these file pointers.)

A program can do input and output on these three files immediately, without having to call fopen() first. The three files have certain conventional usages, as the names imply. Thus many programs take one input file, or stream of data, and transform it in some way into one output stream of data. If this is the case, the input data would normally be read from stdin, and the output data would be written on stdout. stderr is reserved for signalling “errors” or exception conditions, separate from the “normal” output. Programs written in this way can then be conveniently connected together in “pipelines”, with the output from one being automatically routed as the input to another; in bash pipelines are set up with the pipe operator "|".

Thus, for example, suppose we have two programs part1 and part2 which perform two separate transformations on a data stream. The simple (dos) command:

```
part1
```

would cause part1 to be started up, reading its input from the keyboard, and writing its output to the screen.\(^3\)

By contrast, the command:

```
part1 out.dat <in.dat
```

would cause part1 to read its input from the disk file "in.dat" and write its output to the disk file "out.dat". Note that, despite the redirection of stdout, any error or exception messages, written to stderr, would still appear on the screen.

Finally, the command:

```
part1 <in.dat | part2 >out.dat
```

would cause the output of part1 to be automatically routed as the input to part2, whose output would then finally be routed to the file "out.dat".

Thus, if you design your programs in such a way that they take input from stdin and write output to stdout (and errors etc., to stderr), this will mean that these programs can be subsequently mixed and matched in a very flexible way to achieve a variety of different effects. This is a very powerful idea. But even if you have no intention of linking programs in this way, it will still often be convenient to use stdin as a source of keyboard input, and stdout (and/or stderr) as a route for output to the screen. There are specialised versions of several stdio.h functions which access one or the other of these files by default (e.g., printf() is a variant of fprintf() which automatically writes to stdout).

A small subset of the functions declared in stdio.h will now be described in more detail.

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

fopen() is used to create a file pointer or stream, for subsequent input or output to a file. Thus, the return value from fopen() must normally be stored in a variable, so that it can then be used as an argument to other functions such as fgets() etc.

filename is a string giving the name of the desired file. This should conform to the conventions of the “environment” or “operating system” under which the program will be running. Within the UNIX like environment file names consist, in general, of a device specifier (e.g., "/h" ), followed by a “path” identifying the desired subdirectory, followed by an individual file name proper. On Windows platforms there is generally no distinction between upper and lower case characters anywhere in a file name; whereas on Unix platforms they are considered to be distinct. The directory separator character in the UNIX environment is the slash ‘/’ character.

Examples of well formed filename arguments under UNIX like environment might be:

```
"myfile"
"yourfile.h"
"//h/something.xyz"
"//f/NetscapeNT/bookmarks.html"
```

Examples of well formed filename arguments under Windows environment might be:

```
"myfile"
"yourfile.h"
"h:something.xyz"
"f:\NetscapeNT\bookmarks.html"
```

Of course the filename argument, as with any of the other string arguments to be discussed, might equally be a string “variable”—the name of a char
array which had previously been loaded with the desired file name (including the usual null character as the string terminator: `\0`).

*mode* is a string specifying the desired file “access mode”—i.e., what kind(s) of operations are going to be carried out on it. Two examples of legal values for this would be:

"r": Open the file for reading.

"w": Open the file for writing.

In general, *fopen()* will open a file for access either as a binary or a text file—but the default is implementation specific. In any case, regardless of the default, the type of access can be explicitly specified in the *mode* string by adding either a t or b character for text or binary respectively, e.g.:

"rt": Open for reading as a text file.

"wb": Open for writing as a binary file.

If the call to *fopen()* is successful, then the return value is simply the value of the created file pointer. However, if the call fails for any reason (e.g., trying to open a file on device "d:" when no such device is present on the machine) then the return value will be the special zero or null pointer value defined in C. *stdio.h* (and, indeed, several of the other header files) define the symbolic name NULL for this value. Thus, after a call to *fopen()* your program should always check the return value to see whether it is NULL—and take some appropriate action if so (e.g., issue a suitable message and call *exit()*). In any case, if the return value from *fopen()* is set to NULL then *errno* will also be set to some more specific error code.

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

*fgetc()* reads the next sequential character (byte) from the open file identified by the file pointer *stream*. This character is converted into an int value, and is the return value from the function. However, if the end of file is encountered (the last character has already been read), or if any other error is encountered in reading, then the return value will be EOF, and *errno* will be set appropriately. In the case of a text file, the newline character '\n' should be interpreted as terminating a line.

```c
int getchar(void)
```

*getchar()* is equivalent to *fgetc*(stdin).

```c
int putc(int c, FILE *stream)
```

*putc()* converts the int value *c* into the corresponding char value, and writes that to the next sequential position in the open file identified by the file pointer *stream*. The return value is normally set to be equal to the input argument *c*; however, if any error is encountered then the return value will be EOF and *errno* will be set appropriately.

```c
int putchar(int c)
```

*putchar()* is equivalent to *putc*(c, stdout).

```c
int fprintf(FILE *stream, char *format, ...)
```

*fprintf()* is a “print-and-format” function. It provides for transformation of values of any of the native C data types into a corresponding textual or string form; this whole string is written to the open file identified by the file pointer *stream*. *fprintf()* is actually just one of a whole family of functions which provide minor variations on the basic idea of formatted output.

*fprintf()* takes a variable number of arguments. The arguments *stream* and *format* are required—they must always be present and must be of the types specified in the prototype; but these may then be followed by an arbitrary number (including zero!) of further arguments of arbitrary types.

*format* is a string which indirectly specifies how many further arguments are being passed in (if any) and their types.

More precisely, *format* contains “ordinary” characters interspersed with “conversion specifications”. The ordinary characters are simply written to *stream*. But each time *fprintf()* encounters a conversion specification, another input argument is converted into a textual or string form, and output on *stream* instead.

Thus, to simply print a string on the screen (and assuming that *stdout* has not been redirected) one might use the function call:

```c
fprintf(stdout, "Hello World!!!!!!!")
```

Conversion specifications are introduced by the character "%". Examples of some simple conversion specifications are as follows:

"%d" Convert an int value to a normal (base 10) textual representation. The mnemonic way of remembering this is by reading the "%d" as standing for “decimal notation” (there are other format specifications which would render
int values in non-decimal notations, such as octal or hexadecimal). Note, in any case, that "%d" definitely does not stand for double.

"%u" Convert an unsigned int value.

"%ld" Convert a long value.

"%f" Convert a double value. This specification also normally works for float values (technically, float values normally get automatically converted to double values whenever they appear in an argument list—so that it is “right” for fprintf() to treat them the same way as double values). You can think of "%f" as a mnemonic for any “floating point type”, since both float and double are of this generic kind. However, be very careful to note that, on input (i.e., using the fscanf() function), "%f" works only for type float; the specification "%lf" (read it as “long float”) must be used for type double.

"%p" Convert a pointer value. The preferred rendering of pointer values varies from platform to platform; but they are typically presented as unsigned integers in base 16 (hexadecimal).

"%c" “Convert” a char value.

"%s" “Convert” a string value (i.e., the argument should be technically of type (char *) and point at a normal nul-terminated array of characters).

In the last two cases, the argument is already essentially textual, so there is strictly no “conversion” to be done as such.

There are many other more complex conversion specifications, but they will not be detailed here.

Examples of the use of fprintf() might be as follows:

fprintf(stdout, "The answer is: %d.", 42);

It is important to understand that although the int value 42 appears in textual form in the C statement above, when the program is compiled this is converted into an internal, non-textual, representation. This is why, in order to display this value on the screen again, fprintf() must be used to convert back to a textual representation. In this case the nett effect of this call to fprintf() will be that the string:

The answer is: 42.

will appear on the screen (assuming stdout has not been redirected). Precisely the same display would, of course, result from the statement:

fprintf(stdout, "The answer is: %d.", 7 * 6);

Again, the same display would result from the sequence of statements:

x = 7;
y = x;
x--;
y++;
fprintf(stdout, "The answer is: %d.",
(y * x) - 6);

where x and y are int variables. Similarly, if a, b and c are variables of type int, long and char respectively, we might encounter the sequence of statements:

a = 32000;
b = 32000L * 32000L;
c = '!';
fprintf(stdout, "%d times %d is %ld%c",
a, a, b, c);

resulting in the following message on the screen:

32000 times 32000 is 1024000000!

Note carefully, how the extra arguments in the call to fprintf() must pair off exactly with the conversion specifications in the format argument: i.e., for each conversion specification, in sequence, which appears in the format string, there must be one extra argument, of just the right type. For technical reasons, which will not be explored here, it is not always possible for the computer to automatically check that conversion specifications do, in fact, match up with arguments correctly. A mismatch between conversion specifications and arguments in a call to fprintf(), or some related function, is one of the most common kinds of errors encountered in C programs—precisely because the computer cannot always automatically detect it. The effects of such errors are completely unpredictable, and may not be immediately apparent at all, so that they can be extremely difficult to track down. You have been warned!

Some trivial examples of such mismatching would be as follows:

fprintf(stream, "The answer is: %d", 42L);
fprintf(stream, "Or perhaps it is: %l", 42);
fprintf(stream, "Or there again, maybe its %d");
fprintf(stream, "My last offer is: %d", 42, 53);
Note that if you want to move onto a new line on the display screen, you must explicitly write the newline character '\n' to the relevant stream. Try to predict the precise effect of this sequence of statements for example:

```c
fprintf(stdout, "Hello. ");
fprintf(stdout, "Hello?" );
fprintf(stdout, "\ls that you?" );
fprintf(stdout, "Yes?\nNo...\n\nMaybe????\n");
```

Note that because the "%" character in the `fprintf()` (namely as introducing a conversion specification), there is a problem if one actually wants to just output this character. To get around this you just put in two successive "%" characters, thus: "%%". The second one will cause `fprintf()` to recognise that this is not a conversion specification after all, and it will simply output one "%" character without further ado (and without attempting to convert any argument).

The `return` value from `fprintf()` is the number of characters which have been written out, or EOF if any error has been detected. C programmers commonly neglect to check this `return` value from `fprintf()`, presumably because error conditions which it can detect and signal are rather rare. Nonetheless, as a general practice I would recommend that you include code in your programs to check this `return` value (at least to see if it is EOF), unless the your usage is extremely trivial (e.g., just printing the `format` string, without having any extra arguments to be converted).

```c
int printf(char *format, ...)
printf(format, ...) is equivalent to
fprintf(stderr, "message")
```

Note that because the "%" character in the `format` string is interpreted in a special way by `fprintf()` (namely as introducing a conversion specification), there is a problem if one actually wants to just output this character. To get around this you just put in two successive "%" characters, thus: "%%". The second one will cause `fprintf()` to recognise that this is not a conversion specification after all, and it will simply output one "%" character without further ado (and without attempting to convert any argument).

The `return` value from `fprintf()` is the number of characters which have been written out, or EOF if any error has been detected. C programmers commonly neglect to check this `return` value from `fprintf()`, presumably because error conditions which it can detect and signal are rather rare. Nonetheless, as a general practice I would recommend that you include code in your programs to check this `return` value (at least to see if it is EOF), unless the your usage is extremely trivial (e.g., just printing the `format` string, without having any extra arguments to be converted).

```c
int printf(char *format, ...)
printf(format, ...) is equivalent to
fprintf(stderr, "message")
```

Again, it is up to you, the programmer, to ensure that the arguments match the conversion specifications correctly; and again, if they do not, then this will not be automatically detected, and will have entirely unpredictable effects.

The conversion specifications handled by `fscanf()` are quite similar in format to those handled by `printf()`. They will not be described further here except to note one critical difference: to do input conversion to the `double` data type, you must use the specification "%lf" rather than "%f". This inconsistency is just an historical quirk of the C language. You can read "%lf" as being mnemonic for “long float” in this special case.

The `return` value from `fscanf()` is the number of input fields or values successfully scanned, converted, and stored, or EOF if end of file or any error is encountered. In general, it is a very good idea for your programs to check this `return` value to see that it has the expected value.

```c
int scanf(char *format, ...)
scanf(format, ...) is equivalent to
fscanf(stdin, format, ...).
```

```c
int fclose(FILE *stream)
fclose() simply closes the open file associated with stream. The `return` value is zero if this is completed successfully; otherwise it is EOF and `errno` is set.
```

While open files should be closed automatically if or when your program exits anyway, it is still a good practice to explicitly close files with `fclose()` when your program is finished with them. This makes the behaviour of the program more intelligible to someone reading it, and also provides a little extra “robustness” against the possibility that your program might terminate abnormally (i.e., CRASH!).

```c
void perror(char *s)
perror(s) outputs the string s on stderr, followed by an error message which describes or elaborates the error code currently stored in `errno`. It is equivalent to:
```

```c
fprintf(stderr, "message")
```

where `error message` is a string describing whatever error condition is represented by the current value of `errno`.

`perror()` might usefully be called on any occasion when your program detects that a standard library function has failed to operate as expected.
2.7 Diagnostics: assert.h

assert.h provides the declaration of one “function-like” object with the following prototype:

```c
void assert(int status)
```

For technical reasons, this object is not implemented as a function in the normal sense, but is what is called a macro instead. However, for most practical purposes, it can be regarded just as any “normal” function.

If `status` is non-zero then `assert()` will have no effect; but if `status` is zero then a message of the following form will be output to stderr:

```
Assertion failed: status, file filename, line nnn
```

where `status` is the value of `status` (i.e., zero), `filename` is the name of the source file containing the call to `assert()`, and `nnn` is the line number in that file at which the call appears. `assert()` then causes the program to terminate.

`assert()` is potentially a very useful way of dealing with error or exception conditions detected within your programs, where the condition is such that you are unable (or unwilling) to try to make the program deal with it in any more “intelligent” way. The beauty of `assert()` is that, if it is activated, the resulting message immediately pinpoints exactly the position in your source code where the problem is detected. However, note that this benefit will be almost totally lost if you place the invocation of `assert()` within a function of your own, which you then call from wherever the exception is detected: for then the file name and line number output by `assert()` will always be the same (namely within your exception handler) and will not give any indication of the “real” position at which the problem was detected.

2.8 Character Class Tests: ctype.h

ctype.h provides prototypes for a selection of functions which test a char value for membership of a given class. In each case the function takes a single argument, which is technically of type int, but which represents a char value or, possibly, EOF; this is for compatibility with fgetc() as discussed earlier. The return value is always of type int, being zero (false) if the value is not a member of the relevant class, or non-zero (true) if it is.

Some examples are as follows:

```c
int isalpha(int c) : Letter (i.e., “alphabetic”)?
int isdigit(int c) : Digit?
int isalnum(int c) : Letter or digit (“alphanumeric”)?
int islower(int c) : Lower case letter?
int isupper(int c) : Upper case letter?
int isspace(int c) : “Whitespace” (i.e., space, newline, tab, etc.)?
int isprint(int c) : Printable or displayable (including space)?
```

cctype.h also provides two functions which convert the case of letters:

```
int tolower(int c)
int toupper(int c)
```

2.9 String Functions: string.h

string.h provides prototypes for a wide range of functions which operate upon strings in various ways. A brief description of a small sample of these follows. Note that where one of these functions potentially modifies a string, it might, in general, make it longer; therefore in passing in a pointer to the string which will be modified, you must make sure that the char array holding this string is big enough to hold the longest possible string (including the null terminator) which might conceivably result from the string operation(s). If you fail to watch out for this the results will generally be unpredictable—and definitely not very pleasant.

```c
char *strcpy(char *s, char *t)
Copy the string pointed to by t to that pointed to by s. The return value is simply s (i.e., a pointer to the destination string); this is sometimes convenient in forming more complex expressions, but, more usually, it can be ignored (discarded).
```

```c
char *strcat(char *s, char *t)
Concatenate the strings pointed at by s and t; the result is pointed to by s. That is, in effect, the characters from string t are added on at the end of string s. The return value is again simply s.
```
int strcmp(char *s, char *t)
Compare the strings pointed to by s and t. The return value is zero if the two strings are identical; it is negative if s would come before t in an alphabetical ordering; and positive if t would come before s.

2.10 Mathematical Functions: math.h
double sin(double x)
The return value is the sine of x, where x is an angle expressed in radians.
double cos(double x)
The return value is the cosine of x, where x is an angle expressed in radians.
double tan(double x)
The return value is the tangent of x, where x is an angle expressed in radians.
double asin(double x)
The return value is the inverse sine or arcsine of x. x must be in the interval $[-1, 1]$. The return value will be expressed in radians and lie in the interval $[-\pi/2, \pi/2]$.
double acos(double x)
The return value is the inverse cosine or arccosine of x. x must be in the interval $[-1, 1]$. The return value will be expressed in radians and lie in the interval $[0, \pi]$.
double atan(double x)
The return value is the inverse tangent or arctangent of x. The return value will be expressed in radians and lie in the interval $[-\pi/2, \pi/2]$.
double atan2(double y, double x)
The return value is the inverse tangent or arctangent of $y/x$. The return value will be expressed in radians and lie in the interval $[-\pi, \pi]$.

The advantage of atan2() over atan() is that, because it is given separate access to the two original (rectangular) co-ordinates, it is able to distinguish between angles in the left and right half planes (which is not possible if only $y/x$ is passed in, as is the case with atan()).
double exp(double x)
The return value is the exponential function of x, i.e. $e^x$.
double log(double x)
The return value is the natural logarithm of x, i.e. $\ln(x)$. x must be greater than zero.
double log10(double x)
The return value is the base 10 logarithm of x, i.e. $\log_{10}(x)$. x must be greater than zero.
double pow(double x, double y)
The return value is $x$ raised to the power $y$, i.e. $x^y$. It is an error to invoke this function with $x$ equal to zero and $y$ less than or equal to zero; or with $x$ less than zero and $y$ not an integer. Note that although $y$ must always be of type double, its value may be mathematically an integer, e.g. 4.0.
double sqrt(double x)
The return value is the square root of x, i.e. $\sqrt{x}$. x must be greater than or equal to zero.

2.11 Conclusion
This chapter serves only to skim the surface of the facilities offered by the C Standard Library. It is well worth your while to become reasonably familiar with the full range of services which are available in the library: it can speed up programming of many applications very considerably. Again, you can get more detailed and comprehensive documentation at URL:

Chapter 3

Defining C Functions

A minimal C program consists of a single function, which must be called `main()`. However, practical C programs or any size of complexity are never constructed simply with one enormous `main()` function. Instead, programs consist of a number of distinct functions (or “sub-programs”). This chapter reviews some aspects of how to design and code such functions.

3.1 Specification

When defining a new function, you need to first prepare an adequate specification. This will consist of answers, expressed in plain English, to the following questions:

- **What will it be called?**
  
  C function names, like C variable names, consist of a sequence of characters, including upper and lower case alphabets, the digits 0 through 9, and the underscore character `_`. The name should always begin with an alphabetic. Names can be as long as you wish, but anything much longer than 20 characters becomes unwieldy and error-prone and should usually be avoided. All function names must be unique within a given program. Traditionally, most C programmers use only lower case alphabets in function and variable names (i.e., no capitals). However, this is only a convention, and is not a compulsory part of the C language definition. Names of functions are usually chosen based on verbs because they do something—whereas names of variables tend to be based on nouns (they just “are”). It is important to think carefully about the names of both functions and variables in a C program. While the compiler really doesn’t care (as long as the names are unique) these choices can make a big difference to how easy or difficult it is for another human being to understand how the program is intended to operate.

- **What (if any) information will be passed in from the calling site?**

  Many functions can behave differently based on certain information passed into them by their calling function. For example, the standard function `printf()` is able to print out any string on the screen. Which particular string it will print out on any particular occasion is specified by the calling site—the invocation specifies a particular string to be passed into the function.

  The specification of the information to be passed into a function will thus determine the layout of its argument list. The argument list states how many arguments the function will accept, what types they must be, and what names will be used within the body of the function to refer to these arguments.

- **What (if any) information will be passed back to the calling site?**

  Many functions generate a result of some sort, which must be made accessible to its calling site, after the function has finished execution. In the special case of objects of type string i.e. an array of `chars`, this can be achieved by the calling site passing in a string variable to the function; any changes which the function makes to this string, will still be visible, after the function terminates, at the calling site. This is called passing by reference, and is the mechanism used, for example, by the `gets()` function.

  However, this does not work for information of other types. Thus, for example, if a variable of `int` is passed into a function, all that “really” happens is that a copy of this variable is
made available to the function. Any changes which the function makes to that value are *lost* when the function exits. This is called *passing by value*. In this case, the idea of passing in a variable for a function to modify cannot be used to allow a function to pass information back to its calling site. In that case, you should use the *return* value mechanism. If you place a *return* statement in your function then, when that statement is executed, the function will immediately terminate, and the specified value will be made available at the calling site, in the position where the function was invoked. This is the mechanism used, for example, by the *getc()* function. Typically, this *return* value is assigned to a variable at the calling site, so that it can be stored for use in subsequent statements.

The *return* mechanism *cannot* be used to pass back strings (or indeed any array). In that case, you *must* use the passing by reference mechanism, where a string variable, belonging to the calling site, is passed into the function as an argument, and modified by it.

The specification of the information to be passed back from a function may thus lead to the specification of additional arguments (passing by reference), or to the specification of a *return* type (or, possibly, both).

- **What will the function actually do?**

  That is, having stated what, if any, information is passed in, and what, if any, information is passed back, you need to state, in plain English, what the *effect* of the function is supposed to be.

  The idea of defining a new function usually arises in the context of some “higher level” function which you are already writing. Essentially, you want to defer coding some more detailed activity or calculation. This is called “top-down” design, and is a typical strategy for all forms of engineering design, not just software. So, for the time being, you just want to put in a call to a function, which, at some later stage, you will define in detail. The coding of this function-call will, implicitly, at least, answer all of the questions we have just raised.

### 3.2 Coding

Having detailed the function specification, you can code the *skeleton* definition of the function:

```c
<return type> <function name> ( <argument list> )
{
 <function body>
}
```

You then go back and fill in the slots. For example, suppose we decide to define a function called *foo*, which will accept one argument of typ *float*, and generate no *return* value, we can immediately make out the following skeleton:

```c
void foo(float arg)
{
}
```

You need to minimally provide this skeleton, just to allow an *invocation* of the function to even be compiled properly. You may well temporarily put in a “dummy” body to the function—called a *stub*—just to allow you to test out the higher level invocation. But, eventually, you must come back and fill in the body of the function definition with the actual *C* statements which will make it, upon invocation, carry out the activities specified for it.
Chapter 4

Pointers and Structures in C

This chapter provides a short introduction to the concepts and mechanics of used pointers and structures in C. For more detailed information, refer to your textbook.

4.1 Pointers

Pointer data types are a mechanism in the C language whereby a variable can contain, not a "normal" piece of data, but a pointer or reference to another different variable. A pointer is an abstraction of the underlying hardware idea of a memory address. It turns out that being able to use one variable to refer to, or point at, another is a tremendously useful concept, and it is very widely used in C programs.

Consider this simple variable declaration:

```
int i;
```

This declares a variable, i, of type int. Now contrast that with the declaration:

```
int *i;
```

This declares that i is a variable of type “pointer to int” or (int *). That is, i itself does not hold an int value, but the thing it points at does.

Note that you can happily declare pointer variables which point at other pointer variables. Thus:

```
int **i;
```

declares that i is a variable of type “pointer to pointer to int”. The value of i is a pointer; the thing it points at is now another pointer variable; but the thing this points at is an int. And so on!

A pointer value (the sort of thing that can be stored in a pointer variable) can be produced using the “address-of” operator, denoted with the ampersand character &. Thus, the following code fragment might make sense:

```
int i;
int *ptr;
ptr = &i;
```

The net effect is that ptr now points at i.

You can display a pointer value using the %p format specification with printf():

```
printf("Address of variable i is: %p\n", &i);
```

With our particular computer platform, addresses can be thought of as binary numbers with 32 bits. By convention, they are displayed by printf() in hexadecimal, or base-16, notation (where the digits 0 to 9 as used as normal, but the letters A through F are used to represent the base-16 digits for the numbers 10 to 15). So a typical result from executing the printf() above might be something like:

```
Address of variable i is: 0xbbbfc893
```

The 0x prefix is a standard C notation for showing that a number is in hex; this is then followed by 8 hex digits (since one hex digit represents a number between 0 and 15, it is equivalent to 4 binary digits; so a 32 bit number is equivalent to 8 hex digits). In any case, your only use for looking at pointer values will be to assess whether or not different pointer values are equal—and you can do that without knowing any of the details of interpreting hex notation.

A pointer value is dereferenced with the “dereference” or “indirection” operator, denoted with the asterisk character, *. A dereferenced pointer may be used anywhere a variable name can be used.

Thus, normally, a variable name evaluates as the value of a variable; so a dereferenced pointer normally evaluates as the value of the thing pointed at. But a variable name can also be used in an assignment, in which case its value is altered (overwritten); similarly, a dereferenced pointer can be used in an assignment, and the value of the thing pointed at will be changed.
Note very carefully that, whenever you are dealing with a pointer variable, you must be very clear about whether it is its own value that is being referred to, or the value of the thing pointed at; these are generally very different things (they are usually even of different types, never mind values).

A simple use of dereferencing is this:

```c
int i, j, k;
int *ptr;
ptr = &i;
i = 25;
j = 2 + *ptr;
*ptr = j + 2;
```

In this simple fragment, there are two variables of type `int`, and one of type pointer to `int`. The variable `ptr` is given the value `&i`, so it points at `i`; `i` is given the value `25`; so when `j` is given the value `2 + *ptr`, this means “2 plus the value of the thing `ptr` is pointing at” and this evaluates as `27`; finally, `*ptr`, which is to say `i`, is given the value `j + 2` which is to say `29`.

### 4.2 Structured Data Types

The idea of a “structure” in C is that you can group a number of distinct data items, of distinct types, together into one unit. That collection of data items can then (for some purposes at least) be manipulated as a unit. In particular, having once described the “format” or “shape” of the unit, one can declare, in a single step, “structured variables” which have separate slots for all the constituent parts or members. Look at this simple example:

```c
struct
{
    int age;
    double IQ;
} barry, john, mary, henry, susan;
```

`struct` is a keyword which introduces the declaration of structured variables. This is followed by braces which enclose a detailed description of the members of each structured variable. Finally come the names of the actual variables—five of them in this case. The effect of this declaration is to create five variables, each of which has two distinct members, one called `age` the other called `IQ`. To access these members we use the “dot” operator, thus:

```c
barry.age = 29;
john.age = 40;
mary.age = barry.age;
john.IQ = 150.0;
```

and so on. But we can also, to a limited extent, manipulate a structured variable as a whole—for example, by assignment:

```c
henry = john;
```

This would assign the values of the members of `john` to each of the corresponding members of `henry` in turn. Thus, it would be exactly equivalent to:

```c
henry.age = john.age;
henry.IQ = john.IQ;
```

So far, so good. It should be clear that structures might be useful whenever you have information that naturally divides into groups or units with a certain similarity.

This next point is that, if you are using structures at all, it is often handy to be able to declare structured variables (or function parameters for that matter), having a given shape (i.e. a given set of members) in several different places in your program. Now you can certainly do this in the “obvious” way by simply repeating the detailed declaration of all the members in each different place. But it would be nicer if the compiler could let you declare the shape once, and then refer back to it later. And, fortunately, it does—actually in a couple of different ways. But I will just show one:

```c
struct person
{
    int age;
    double IQ;
};
```

This time I have put an identifier (`person`) immediately after the `struct` keyword, and before the left brace. This does not create a structured variable called `person`. Instead, `person` acts as a tag or identifier for this particular shape of structured variable. Once I have given this declaration, then, at any subsequent point in the program, I can use simply `struct person` as a shorthand for the full declaration. So I could write, for example:

```c
struct person barry, john, mary;
```

This would then create variables `barry`, `john` and `mary` with the same shape as before. Then somewhere later on (or inside some other function) I might write:

```c
struct person henry, susan;
```

Be clear on the advantage of adding an identifier, such as `person` to a particular shape of structure: it
just saves having to re-enter the detailed description of the shape in several different parts of a program.

One slight difficulty might be evident here: in reading a program one can easily become confused between identifiers such as henry, which refer to actual variables, and person which simply identifies an abstract shape for variables which might get declared at some stage. It would be ridiculous, for example, to try to refer to person.age because there is no such variable as person. Rather, person simply provides a name for a template from which variables might be created. Once you get used to using structures this should not cause any problem. Whenever you see the keyword struct you will expect to see it followed either by a left brace or a name for a particular shape of structure. But when you are just starting off, it may be a good idea to adopt some kind of systematic identifier convention—i.e., to choose your identifiers so as to clearly distinguish between those that refer to actual variables and those that simply refer to an abstract shape. One simple convention that I sometimes use is to append .s (for “structure”) to the end of any identifier that refers to a structure shape. So I would actually use the identifier person.s. But note carefully that is only a convention: it has no significance for the compiler at all—it is purely a device to remind a reader of what’s going on.

OK, now we can declare a structure shape, and give it an identifier; we can declare variables or parameters of that shape; and we can access the members of such variables. We now join these ideas up with the idea of a pointer.

Roughly speaking, we can embed, within a structured variable, a pointer member that points at something else. In particular, we can embed a member that can point to another structured variable of the same type or shape. And once we can do that, we can create indefinitely large networks of structured variables that are linked together by pointers. This turns out to be an immensely powerful programming idea—which, regretfully, we shall not be able to pursue properly in this introductory module. For now, you simply have to take it on faith that this idea is potentially useful.

Consider this fragmentary program:

```c
struct person_s barry;
struct person_s *p;

p = &barry;
(*p).age = 44;
}
```

In this program there is a structured variable called barry, and a pointer variable called p.

For the purposes of the example, p is made to point at barry, and then the age member of barry is indirectly accessed via p. This is done with the construction (*p).age. That is, p is first dereferenced using the * operator; this yields the thing p points at (namely barry), and the dot operator is then used to access the age member of this structure.

The parentheses around *p are necessary here because the dot operator takes precedence over dereferencing by default. So if I simply write *p.age, the compiler thinks I want to first take the age member of p and then dereference that. Now, since p isn’t even a structured variable, and therefore has no members at all (never mind a particular one called age) this cannot make sense, and the compiler would simply generate an error. But in more complicated cases the compiler might not even diagnose the error effectively. Anyway, the point is that if you are starting with a pointer, and you want to get at a member of the object being pointed at by that pointer, then you must use something like (*p).age.

This construction is moderately convoluted; worse still, it turns out that this kind of operation or access is actually needed very commonly in C programs that operate on complicated data structures. The C language therefore provides a single operator, the arrow operator, which combines the effects of dereferencing and accessing a member of the structure thus accessed. It is written like this:

```c
p->age = 44;
```

Be clear that p->age is simply shorthand for (*p).age: if you ever find the arrow operator confusing, then try replacing it by the “longhand” version, with an explicit dereferencing and explicit dot operator.
Chapter 5

Laboratory Report Guidelines

This chapter gives guidelines for structuring and writing the Laboratory Reports submitted for this module. It also indicates the criteria which will be used in marking these reports. Note that these same criteria will be used in marking the reports submitted from the Laboratory Exam which will determine the grade ultimately awarded for the module.

It is a specific objective of this module to develop your written communication skills. Therefore, in assessing the module, we will be interested not only in your core ability to develop software, but also in your ability to communicate effectively about this process. Thus, it is very important that you study this chapter carefully, and practise applying the guidelines given here in every relevant lab session.

These guidelines are similar to those applying for the module Software Engineering 1. However, a significantly higher standard of reporting is expected in this more advanced module, and these guidelines are correspondingly more extensive and detailed. Therefore, even if you are already very familiar with the guidelines for Software Engineering 1, you will still need to carefully study the more detailed guidelines presented here.

Reports must be submitted by email. There is a special address for submission:

- ee105-reports@list.eeng.dcu.ie

If you have any queries or criticisms of these guidelines, please post them to the email conference at:

- ee105-talk@list.eeng.dcu.ie

Please take care to submit your reports to the indicated report address. Do not post reports to the conference email address or other email addresses. They will not be taken into consideration!

5.1 General

The Software Engineering 2 Lab reports are submitted using email. This implies, of course, that they must be typed up. However, apart from this, the purpose and content of these reports is similar to reports prepared for any other Laboratory Course. Therefore, before reading these guidelines, make sure that you are familiar with the Undergraduate Laboratory Handbook. That gives guidance for all Laboratory based exercises on your Undergraduate Programme, and outlines the general considerations applying to reporting all laboratory based work. These apply, as far as they are relevant, to the reports for the Software Engineering 2 module also.

The central principle of writing the reports for Software Engineering 2 is that they should explain where your time went during the lab session. If you successfully completed the assigned exercises, this can be relatively straightforward—copies of the file(s) you created or modified, details of the tests you carried out and their results. The difficulty arises where you do not complete the assigned work. In that case, you have to explain, as clearly as you can, what the problems were: whether you simply did not understand the exercise, whether you ran into compiler messages that seemed unintelligible, whether you could not imagine how to test the program, or it worked on some tests and failed others—and so on. The point is that, if you spend three hours in the Lab, you must have done something: so report on what that something was, no matter how far removed it may seem from what was actually requested.

5.2 Typing Up

The reports must be prepared and submitted during the Lab session.

It is recommended that you use a text editing
application to prepare the report as this may maintain certain formatting. If you use a word-processing package such as MS-Word etc., please indicate this and attach the report file to the e-mail.

It is recommended that you create the report file at the very start of the Lab session. Keep an editor window open for the file throughout the session. In this way you can add notes to it as you go along. This then minimises the work required to complete the report at the end of the session.

However: ensure that you save the report to disk every time you make any addition to it. Otherwise, if there is a power failure, or the PC crashes, you will lose all your notes up to that point.

In writing the report (and in preparing email messages in general) limit your lines to 72 characters wide at most; anything wider than that is likely to cause viewing problems at the receiving end.

5.3 Structure

It is helpful, both for writing and marking, if the report can have a somewhat standardised layout or format. We cannot be too exact in giving a layout because different exercises may require a somewhat different approach. Also, the layout will have to be varied depending on the detailed problems or difficulties which you run into. Nonetheless, the following sections outline a preferred, generic, layout of the report, which you can use as a starting point. These sections should be repeated as appropriate if the session is broken down into several separate exercises.

Note that, while you can simply try to write the report sequentially, as the Lab progresses, you may well find that you sometimes want to backtrack, and revise an earlier section, or add some new information to it. This is where the computer scores over a paper Lab book: it is easy to back up and revise, or delete, or augment what was already entered. You should not get carried away with this freedom of course; but equally, you should not feel compelled to write the report in a pure sequential manner, in exactly the order you tackle the exercises during the session.

Take a look at the skeleton report file (Appendix B). This shows the overall layout of a report. You can use this as the basis for your own reports.

5.3.1 Heading

Each Lab Report must start with a standard heading as shown in the [skeleton report file]. Edit in the appropriate information for the particular session. The Session Title should be filled in with the title as specified in the notes for the session.

5.3.2 Plan

Given the statement of a particular exercise for a Lab session you should try to formulate an outline plan as early as possible—and record it in your report. This holds even if you subsequently have to severely modify (or even abandon) your plan.

A plan should be a concise statement of how you intend to approach the exercise. The details will vary, of course, from case to case. But the plan will typically involve breaking the overall exercise into smaller, more manageable pieces; experimenting with things you don’t yet understand properly; identifying problems you anticipate; and perhaps an outline of how a program will be structured (what is technically called an algorithm). You should also try to anticipate how you will go about testing the program.

The lab instructions may well already outline a plan. But even in that case, you should consider whether you could usefully break down some of the steps suggested in the instructions into even smaller pieces.

There will be no unique, “correct” plan. Some questions we will ask in assessing the report are: Did you make a plan at all? Is the plan clear and concise? Is it understandable? Is it practical? Is it “internally” consistent? Does it ultimately correspond with what was asked for in the instructions?

A plan will normally be just a single paragraph. Subsequently in the report you can then describe progress compared to the plan. In particular, you can describe problems you encounter in following the plan, and any on-going changes you decide to make to it.

The plan is not a restatement or paraphrasing of the instructions already given in the online notes. Such restatement is a waste of time, and may well result in marks being deducted!

5.3.3 Development and Test

These are the two main sections of the report. They should deal with your activities as you develop and test whatever program is required by the particular exercise.
Development involves the following general kinds of activity:

- **Coding:** This is composing new C code.
- **Compilation:** This is using the compiler to translate the C code you have composed—the source code—into the executable form that the CPU can directly execute.
- **Execution:** This is executing the program to assess whether, or to what extent, it fulfils the minimum functionality required.

Test involves the following kind of activity:

- **Testing:** This means providing suitable test data to assess whether, or to what extent, your program behaves as required. It will generally be up to you to formulate satisfactory test cases. It is particularly important to try to test the limits of correct operation—to test the program with extreme or exceptional input data, to assess whether it maintains correct behaviour in these cases. For example, if something is supposed to work over a certain range of numbers, make sure you test it at the limits of this range. If something is supposed to deal in a certain way with "illegal" input data, then actually test it on "illegal" input data to check this.

Development and Test never just go straight through these individual phases (i.e. Coding, Compilation, Execution, Test) one single time. Problems will invariably arise which mean that you must cycle back through these phases, possibly many many times, before a satisfactory version of the program is completed. Thus it is not appropriate for the Development section of the report to be simply divided into subsections for these distinct activities. Instead, it should be divided into subsections according to the successive problem(s) you find yourself dealing with.

Thus, each subsection should try to clearly state a problem you are currently trying to solve, and your ultimate solution (if any!). A problem might be “Code an initial version of the program”, or “Get rid of a compiler error message saying ‘Unrecognised Symbol at Line 5’”, or “Test whether the variable x is being correctly updated” or whatever.

A problem will often be of the form that something is not behaving as you expected. In such cases try to be as clear as you can about what you expected and what actually happened. Distinguish between problems that are “compile-time”—i.e., they are indicated by some message from the compiler (and should be included in the Development section)—and those that are “run-time”—i.e., they are manifested only when the program is executed (these should be included in the Test section).

The Development section must include the evolving text(s) of your program. As you make significant changes or additions to the program, you should include the new program text—possibly in whole, or possibly just those parts that have changed, whichever you judge is more appropriate.

A good rule of thumb for how much detail to include in the Development section is that there should be enough information so that another person could reconstruct what you did more or less exactly.

### 5.3.4 Conclusion

The concluding section is concerned with summarising clearly and concisely what you achieved, relative to the assigned exercise. If you have learned anything new during the exercise, you can state that here too.

Do not make any bald or simplistic claims to the effect that a program (or function etc.) “works”—except insofar as that is backed up by test results you have already given in the Development section. A program that merely compiles without error is a very long way indeed from one that works. Keep this distinction clear in your mind when writing this section. Do not try to pad this section out, or disguise some shortfall in completing the exercise. The marks here go for clarity, and honesty, not actual achievement!

### 5.4 Including Source Files

The exercises typically require that you create or modify one or more C source files. You must include these source files, or relevant fragments of them in the Development section of your report. Do not include source code that was already provided as part of the original online notes.

You should clearly delimit such included source files or fragments in the report (so that the program or fragment can be clearly distinguished from the text of the report proper), and explain exactly what they are. Here is an example:

This is the program "BDAY.C". It is a modified version of the program "XMAS.C" from page 4063 of the textbook. It prints a birthday greeting.

--------------------------------------------------------------------------
/* BDAY.C */

#include <safe-c.h>

int main(void)
{
    put_string("Happy Birthday!!!");
}

---

Note that you can generally exploit the Copy and Paste facilities of whatever text editor you are using to copy from one file to another.

Where you have created a new source file from scratch during the Lab session you should include the complete file in the report. Where you have only modified, or augmented, some given file then you should only include the modified parts, or the additions, in the report. You have to exercise some judgment here. The basic principle is that the report is a report on your work, so you omit anything which was already done for you; but you may have to bend this rule somewhat in order to provide enough context for the report to be understandable. In any case, the surrounding text of the report should make it clear what you have included, and why.

Typically you will be developing one or more source files as the Lab progresses (correcting, debugging, enhancing etc.). So a natural question is whether you should separately include every single variant in your report. Again, this requires some judgment. The general principle is that the report should definitely contain the “final” versions of any relevant source file(s)—i.e., as they were when you finished working on them. This is true regardless of whether this “final” version is actually functional or not.

In addition to such final versions, the report should also contain earlier versions (or fragments of earlier versions) where that is useful or necessary to illustrate some significant problem or difficulty you encountered. This will be discussed further below.

5.5 Problems

Typically you will encounter one or more unanticipated “problems” in any given exercise. This is not a cause for alarm or despair. We expect you to encounter problems; you would not have the opportunity to learn anything if you did not. Problems are opportunities for learning—and opportunities for getting marks in your report!

Note that this applies to the Lab exams just as to the normal exercises. We do not expect that you will be able to complete a exam without running into any problems. The central point of the exam is not to see whether you can avoid running into any problems, but whether you can effectively respond to whatever problems do arise.

Some problems will be very minor and will not deserve reporting. For example, you leave out a semi-colon, or mistype a variable name. Provided these are easily and quickly fixed, you should omit them from the report. But, as a rule of thumb, any problem or difficulty which takes up anything in excess of 10 minutes of your time qualifies as significant, and something which must be described in the report.

Remember: you can still get a very respectable mark on a report, even if you make relatively little “progress” in completing the given exercise—provided you can clearly explain what problem(s) you encountered and what steps you took to solve them.

In fact, you will often find that the discipline of trying to explain clearly what your problem actually is often gives you an immediate insight into how to solve it!

Problems are extremely diverse, and it is impossible to give a complete description here; but there are some specific kinds for which we can give more detailed advice:

- **Problems of Understanding**: For example, “I don’t understand what a matrix is”, “I don’t know what type double means”, “I don’t understand what for does”, “I don’t understand what the given program SUPER.C is supposed to do”.

  These are all perfectly good statements of a problem. The sorts of steps you can consider in trying to solve them include (re-)reading any relevant section of your textbook, (re-)reading the lab notes, looking for information elsewhere in this manual, trying something in isolation in a small test program to see what happens, asking a colleague, asking a demonstrator.

- **Compilation Problems**: If messages are thrown up when you attempt to compile a program, then these definitely constitute prob-

---

1 Asking a colleague or demonstrator will not generally apply in a Lab exam of course; but even in that case you may ask for certain kinds of help—e.g., if you think the instructions you have been given are inconsistent or impossible to carry out.
lems. Some may be trivial, as already discussed; but some may be very difficult, in which case they should be discussed in the report. In that case, you should include, verbatim, at least the first message which is coming up, together with enough of the program file to see the context in which the message is appearing. Strategies for solution include: trying variations in the program to see what effect these have on the message(s); using the online documentation to try to get more, or better, information; trying to find a different way of programming whatever it was you wanted to program. And so on.

• **Run-time Problems**: These are generally of the form “I ran this program, gave it this data, and this (unexpected) thing happened”. Note carefully that an adequate description of this kind of problem requires you to say clearly what the “expected” or “correct” behaviour would have been. In fact, it will sometimes turn out that the solution to this kind of problem will be to recognise that you were mistaken in what you expected (i.e., the program is actually behaving “correctly”). In any case, solution strategies for this kind of problem include: rerunning the program with the same data (to see if the symptoms are repeatable); rerunning the program with different data (to see if a pattern of failure can be identified); adding extra code to the program to get more information on what’s going wrong; cutting down the program to try to localise the problem. And so on.

• **Equipment Problems**: You may encounter problems with the lab equipment—hardware or software. These will typically be outside your control to fix. But, if such a problem has impacted significantly on doing the exercise, you should include some discussion of it in the report. In particular, you should explain what the symptoms were, why you classified it as an equipment problem, and how long it took for it to be sorted out.

5.6 Presentation

The report must be clear, concise, and “reasonably” professional. That is, there should not be too many errors of presentation (grammar, spelling, etc.). The style need not be too formal (e.g. you can use the first person—“I did this..”, “I tried that...” etc.) if that is what suits you. However, you should not be too informal either (e.g., using slang terms etc.).

Take careful note of the following more specific points, which represent the most frequent presentation errors which have been observed in previous students’ reports:

- Lines must be limited to no more than 72 characters.
- Insert one blank line between paragraphs. Do not indent (i.e., leave extra space in front of) the first word in a new paragraph.
- Insert one space (no more, no less) after each comma, semi-colon (;) or colon (:).
- Insert two spaces after each full stop.
- Think of a left bracket as part of the following word; think of a right bracket as part of the proceeding word; so, do not insert a space after a left bracket or before a right bracket.
- Capitalise the first letter of the first word of each sentence—except where that word is an identifier (i.e. a name of a variable or function etc.) in a C program; in that case, use exactly the capitalisation (usually none at all) which appears in the program itself.
- “i.e.” means “that is”; “e.g.” means “for example”. If you use these at all, please use them appropriately, and do not confuse them with each other.
- When the letter s is added to make a plural form, do not insert an apostrophe (e.g. “I inserted two extra line’s of code” is wrong). Conversely, when the letter s is added to make the possessive case, do insert an apostrophe (e.g. “This line’s semi-colon had been mistyped as a colon” would be correct).
- On the other hand, the possessive pronoun “its” does not have an apostrophe. The only situation where an apostrophe is correct in “it’s” is where the apostrophe indicates a contraction of “it is”.
- The preferred method of showing emphasis in a plain ASCII file is to surround it with asterisks, like this:

  This is how to show *emphasis*.
5.7 Marking Scheme

The following general principles will apply:

- The report should follow the general outline detailed in previous sections.
- Full marks will be awarded only where all the elements asked for in the instructions have been successfully completed, tested, and adequately described in the report.
- Marks will generally be deducted if a report is too long, and, in particular, if it repeats any significant extracts from the online notes.

Where a session involves several distinct parts or exercises, the instructions will state how the total marks are allocated between these exercises. Within each such part or exercise, marks will be allocated as follows:

- **Plan**: 10%.
  In more detail: this will be awarded on a qualitative assessment of whether the plan is coherent, whether it is appropriate, whether it deals adequately with the exercise as stated etc. It will not depend on whether you actually succeeded in carrying out the plan!

- **Development**: 30%
  10% will be awarded for presenting code which is even approximately correct—correct in outline. A further 10% is awarded if the code is free of syntactical errors (i.e. compiles successfully)—or if the report gives a good explanation of what syntactical errors are still present, and what steps were taken in trying to solve them. A further 10% is awarded for program style—whether the code is sensibly indented, with good line breaks, sensible spacing, meaningful identifiers, and effective comments etc.

- **Test**: 30%
  10% will be awarded simply for specifying, in detail, suitable run time test cases: i.e. tests to be carried out, and the (correct) expected behaviour. 10% will be awarded for presenting actual execution time results for these cases—clearly stating, in each case, whether the test was passed. A further 10% will be awarded if the program passes all such test cases, or if the report gives a good explanation of what errors are still present, and what steps were taken in trying to correct them.

- **Conclusion**: 10%
  These marks will be awarded provided the conclusion is a clear, concise, and accurate summary of what was achieved and/or learned.

- **Bonus**: 20%
  The final 20% is reserved as a “bonus” to be awarded entirely at the discretion of the marker. These marks will be awarded for particular excellence of various sorts. For example, where the report is particularly well written and presented; where the testing has been particularly comprehensive; where the code is particularly elegant; or where the report demonstrates particularly good problem solving skills. This last is especially important: the point being that, even if you get mired down in difficulties, and make little progress on the assigned exercise, you might still qualify for some or all of this bonus if you can demonstrate that you adopted an effective, logical, structured and systematic approach to trying to sort out the difficulties.
Chapter 6

Laboratory Notes

6.1 General Comments

The primary way of studying software engineering is to do it. The laboratory exercises are crucial to this module: you could probably learn more from the time you spend in the laboratory than from lectures and tutorials combined.

Each student attends one three-hour scheduled lab session every two weeks. You should aim to spend at least three further hours of private study time at a PC over the same period. The scheduled lab sessions are deliberately structured so that each student has a separate PC; however, for private study you may find it easier to get at a machine, and also more effective, to work together with one or two others—provided that you spend time discussing what is happening, and taking turns at actually operating the PC.

These labs sessions are designed on the assumption that you will spend 1–2 hours studying the materials for each session in advance. Provided you have the discipline to do this, then all of the sessions should be completable in the assigned time.

For the purposes of the labs the total module class will be divided into four groups. Consult the lab schedule to find out what group you are in and what lab sessions you should attend.

You should buy a new lab logbook to be used solely for the lab sessions associated with this module. Use this notebook both to prepare notes in advance of each lab session, and to make additional notes, calculations, records of results etc., during the session. However, as in the module [Software Engineering 1], and in contrast to most other laboratory subjects, this logbook will not be used to hold formal lab reports. Instead, you will maintain the logbook just with your own personal notes; but these personal notes will serve as the basis for writing separate formal reports, which will actually be submitted (and archived) electronically.

You should consult the [Laboratory Report Guidelines] for advice on how to prepare lab reports, and the criteria which will be used in marking them. These same guidelines and criteria will also apply to the report produced in the final [laboratory examination].

6.2 Command Line Redirection

Several of the programs to be developed in the semester have the form:

- Gather some input data.
- Carry out some calculations.
- Display the resulting output data.

In the default case, the input data is read from the keyboard and the output data is displayed in the bash window in which the program is running. However, it will often be more convenient for the input data to be read from a disk file, and/or for the output data to be stored in a disk file. Perhaps the simplest way of arranging for this is to use what is called command line redirection.

Suppose your program is in a file called "foobar.c". After successful compilation, you will have a executable file called "foobar.exe". Normally you would execute this simply by giving the name "foobar" as a command. However, if you want to use command line redirection, you would give a command somewhat like this instead:

```bash
foobar >out.txt <in.txt
```

1Of course, that does not mean that you can afford to miss lectures or tutorials...
The < serves as the “input redirection” character in an bash command line. The effect is that the input for foobar will no longer be taken from the keyboard, but from this file instead. Similarly, the > character in an bash command provides for output redirection, to direct output to a file instead of the screen.

With any given command invocation you can choose to use no redirection at all (as we have been doing up to now), or to use input redirection only, or output redirection only, or both together.

6.3 Session 1: Trigonometry

This session is concerned with practical experimentation with the C facilities for breaking down a program into smaller, more manageable, pieces called functions.

The Standard C Library makes available a large range of standard mathematical functions. These include standard trigonometrical functions, sin(), cos() and tan(). These three functions each accept one argument (of type double) and return a result (also of type double). The argument must be an angle measured in radians.

Often it is more convenient to work with angles measured in degrees instead. The ultimate objective of this session is for you to write your own trigonometric functions (exploiting the existing standard functions) which will accept arguments in degrees rather than radians. However, we shall get there through a series of smaller steps.

Before this session it is essential that you review and understand some very basic example programs which illustrate functional decomposition (see Appendix C). These are:

- "sum0.c": This program prompts for two numbers, adds them, and prints out the sum. It does not use functional decomposition at all.
- "sum1.c": This is a modified version of "sum0.c" in which a new function has been defined, called get_input() which combines the effects of the standard C functions printf() and scanf(). This illustrates how to define a function which has to pass back, or return, some information to the place which it was called from; and how to use, or access, that information at the calling site.
- "sum2.c": This is a further modified version, in which a second new function has been defined, put_output(). This illustrates how to define a function where some information has to be passed from the calling site, into a function, and how that information can be accessed within the function.

The session is divided into five separate, short, exercises. Each of these builds on the previous one. It is important that you read through the instructions for all the exercises, in advance of the lab session. None of the exercises is very complex in itself, and you should be able to complete all five in the time available. However, it is more important to fully complete as many of the exercises as possible than to make an incomplete attempt at them all. So, it is essential that you take the exercises in the order given, and do not attempt to go on to a following exercise until you have fully completed its precursor.

Each exercise involves developing a program. Note carefully that “develop” here means writing, compiling (with no errors or warnings), and testing. The tests should exercise a reasonable range of behaviour before you consider them complete. Your report must specify exactly (including actual numerical values, where relevant) what tests you carried out, what results you expected, and what results the program generated. If the actual results are not exactly the same as the results you expected, you must comment on this.

6.3.1 Exercise 1: The sin() Function (20%)

Develop a program, called "trig1.c", which will prompt for an angle value (in radians), and print out the sine of that angle.

Where appropriate, this program should use the functions get_input() and put_output() defined in the example program "sum2.c". You can simply cut and past the relevant section of that program into your own program.

This program does not involve defining any additional new functions of your own.

6.3.2 Exercise 2: Angle Conversion (20%)

Modify "trig1.c" as follows, naming the new program "trig2.c".

"trig2.c" should prompt for an angle in degrees. It should then calculate the corresponding angle in radians, using the formula:

\[ \theta_r = \theta_d \times \frac{\pi}{180} \]
where $\theta_d$ and $\theta_r$ denote the angle expressed in degrees and radians respectively.

Having converted the angle to radians, it should then calculate, and print out, the sine (this behaviour is unchanged from "trig1.c").

The degrees conversion calculation should be carried out “directly” by suitable statement(s) within the main() function. That is, this program again does not involve defining any additional new functions of your own.

6.3.3 Exercise 3: Function Definition (20%)

Modify "trig2.c" as follows, naming the new program "trig3.c".

"trig3.c" should outwardly behave exactly as "trig2.c". However, instead of carrying out the angle conversion directly within the main() function, main() should invoke or call a new function, called deg_to_rad(), to do this conversion. This should accept a single argument, of type double, representing an angle in degrees. It should return a result, also of type double, being the corresponding angle in radians. Of course, you have to provide the definition of this new function.

6.3.4 Exercise 4: Refinement (20%)

Modify "trig3.c" as follows, naming the new program "trig4.c".

"trig4.c" should outwardly behave exactly as both "trig3.c" and "trig2.c". However, instead of calling deg_to_rad() and then calling sin(), main() should simply call a single new function, called deg_sin(). This should accept a single argument, of type double, representing an angle in degrees. It should return a result, also of type double, being the sine of this angle. Again, you have to provide the definition of this new function. It should use the function deg_to_rad() which you have already written.

6.3.5 Exercise 5: Finale (20%)

Modify "trig4.c" as follows, naming the new program "trig5.c".

"trig5.c" should prompt for an angle in degrees, and then print out the sine, cosine and tangent of this angle. The main() function should be structured to invoke functions called deg_cos() and deg_tan() to calculate the cosine and tangent respectively. You will have to add definitions for these functions, similarly to the definition of deg_sin().

6.4 Session 2: Encryption

“Encryption” is the process of disguising a message or text so that its meaning will not be apparent to anyone who may intercept it, accidentally or otherwise. Encryption is today a fundamental technology in the development of secure electronic payment systems, which, among other things, are allowing the increasing commercial exploitation of the Internet.

If you’d like to learn more about the general subject of cryptography, try some of these links:

- [http://raphael.math.uic.edu/~jeremy/crypt/crypt.html](http://raphael.math.uic.edu/~jeremy/crypt/crypt.html)
- [http://www.baltimore.ie/](http://www.baltimore.ie/)
- [http://www.turing.org.uk/turing/](http://www.turing.org.uk/turing/)

6.4.1 Exercise 1: The Caesar Cipher (50%)

In this exercise you will work with a very simple encryption procedure. This is called a substitution cipher, in which each letter of the original message (the “plaintext”) is replaced by a different letter to generate the coded message (the “ciphertext”).

To simplify matters further, we will restrict our plaintext to use only the upper case letters A to Z, together with the space and newline characters; furthermore, space and newline characters will be left unchanged by the encryption.

In each case, the replacement letter will be a fixed number of letters later in the alphabet compared to the original letter (and “wrapping around” back to A again after Z, as necessary). The number of letters to count forward will be called the encryption key.

Thus, with a key of 3, we would replace A by D, B by E and so on, with, finally, W being replaced by Z, X by A, Y by B and Z by C.

This simple kind of cipher is sometimes called a “Caesar Cipher” after Julius Caesar, who is said to have used it for secure battlefield messages.

To implement a Caesar cipher with a C program, note that, when accessing individual characters in C, what you are “really” dealing with are numeric codes for the characters. As such, you can do any normal arithmetic (addition, subtraction etc.) and comparison (greater-than, less-than etc.) operations on them.

The relationship between the characters, as displayed on your screen or in your editor, and the numeric codes accessed within your C program is, strictly speaking, arbitrary. It is not specified by the
C language standard. It is thus open to anyone developing a C compiler to adopt any character numbering scheme they like; you cannot reliably assume that the numbering scheme in effect for one C compiler will be the same as for another. Strictly, you need to check the documentation for your particular compiler to find out what the numbering scheme is.

However: there is one particular numbering scheme which is the most widely used, and to which most C compilers conform (including gcc): this is the so-called ASCII scheme. In ASCII, the upper case letters A to Z have consecutive numeric codes from 65 to 90. Thus, given a particular key value (a number from 1 to 25) simple addition will encipher a letter as required—except that if the sum is greater than 90 you must “wrap around” to the start of the alphabet again by subtracting 26.

You are required to develop a program which will input a series of lines of characters, and encode them with a Caesar cipher; as each line of plaintext is input, the corresponding line of ciphertext should be output. It is up to you to devise a mechanism whereby the user can indicate that all lines have been input, and to implement the program in such a way that it will then terminate.

The key for the encoding should be “hard coded” into your program (as opposed to being read in at run time). Thus, changing the key will involve modifying the source program and recompiling.

You may find it handy to be able to run your program on an input file (and producing an output file) rather than working only with the keyboard and screen. Use command line redirection to achieve this.

6.4.2 Exercise 2: Cryptanalysis (20%)

Now think about the problem of cryptanalysis: that is, suppose you are “the enemy” and have intercepted a transmission known to have been enciphered with a Caesar cipher. How would you set about trying to decipher it? You may wish to challenge a colleague in the class to see who can decipher the other’s ciphertext message most quickly. Record your conclusions in your report.

6.4.3 Exercise 3: The Vigenère Cipher (30%)

Finally develop a program to implement the somewhat more sophisticated Vigenère cipher, invented in the 16th century by Blaise de Vigenère. This has the same basic idea as the Caesar cipher, but the key no longer stays the same throughout the message: instead it follows some periodic cycle. A Vigenère key is thus not a single number but a sequence of some fixed length—say 10, 3, 25, which would mean the first letter should be shifted by 10, the second by 3, the third by 25, and then back to 10 again etc.

6.4.4 What next?

While these ideas are fresh in your mind, this would be good time to have a sneak preview of section 6.6 (the lab session on extracting letter frequencies—a key idea in code breaking).

6.5 Session 3: Projectiles

6.5.1 Problem Specification

The purpose of this exercise is to calculate the theoretical trajectory of a projectile in a constant gravitational field. In other words, we want to calculate the path followed by, say, a ball thrown in the air.

Historically, the calculation of trajectories of artillery shells during the second world war provided one of the major incentives for the development of high speed electronic calculating machines—which led directly to the development of modern digital computers.

We are going to make some simplifications. We will work in just two dimensions (i.e., as if we are looking exactly side on at the path of the ball). We will also neglect air resistance, and will treat the projectile as a point mass. This allows a simple analysis of the motion of the projectile, using newtonian theory.

In our two dimensional (x and y) coordinate system, x will denote horizontal position, and y will denote vertical position.

The initial value of x will be taken as zero.

A zero y value will correspond to the projectile being on the ground. The initial value of y will be some positive value (roughly, the height of the person throwing the ball).

\[ V_x \] will denote the horizontal component of velocity, and \[ V_y \] the vertical component. We will assume that both \( V_x \) and \( V_y \) are initially positive. Similarly, \( A_x \) and \( A_y \) will be the components of acceleration.

The “initial” information provided for a particular run of the program will be the initial height (y) and initial velocity components (\( V_x \) and \( V_y \)) of the projectile. These values will be assigned to suitable variables. You can either code values directly into

\footnote{If you really want to know, “ASCII” is an acronym for American Standard Code for Information Interchange.}
the program (in which case you will have to change the program, and recompile, every time you want to try different values) or you can have the program prompt for and read in values when it starts running.

The output from the program will be a display on the screen of the numerical values of $x$ and $y$ at successive, “small”, intervals of time. The program should terminate when $y$ becomes zero again (i.e., the projectile falls back to earth). This display may be captured into a text file and a graph plotted of $y$ versus $x$.

Informally, the motion of the projectile is pretty simple: it moves steadily to the right (positive $x$ direction), simultaneously going up for a while (positive $y$ direction), and then coming down again to land.

Formally, at each time step, the new $y$ co-ordinate may be calculated as:

$$y(t + \Delta t) = y(t) + \Delta y(t)$$

where (approximately):

$$\Delta y \approx V_y \times \Delta t$$

In words, if the velocity in the $y$ direction is $V_y$, then, in a time $\Delta t$, the projectile will move a distance of $V_y$ multiplied by $\Delta t$; and the new $y$ position is the old $y$ position plus this movement. This calculation would be exact if $V_y$ were constant; if $V_y$ is changing (as it will be in our case) then the calculation will still be approximately correct, as long as $\Delta t$ is “small”—namely small enough that the percentage change in $V_y$ in that time is “negligible”. Of course, you can only estimate this once you have some idea of how quickly $V_y$ will be changing...

The new $x$ co-ordinate may be calculated similarly.

Now, in principle, both $V_x$ and $V_y$ could also be simultaneously changing, and have to be recalculated, at each time step. But since we are neglecting air resistance, and since gravity works straight down, there will be zero acceleration in the $x$ direction ($A_x$ is zero), so that $V_x$ will actually remain constant, equal to its initial value. There will be acceleration in the $y$ direction however, of value $-g$, where $g$ is the acceleration due to gravity at the surface of the earth (about 9.81 meters per second per second). This is negative because our $y$ co-ordinate is positive in the upward direction and gravity works downward (last time I checked anyway...). This acceleration in the $y$ direction will be constant (since $g$ is not changing). Thus, the new $V_y$ at each time step can be calculated as follows:

$$V_y(t + \Delta t) = V_y(t) + \Delta V_y(t)$$

where:

$$\Delta V_y = A_y \times \Delta t$$

That is, the change in the (vertical) component of speed is simply the (vertical) acceleration multiplied by the time it acts over; and the new speed is simply the old speed plus this change.

You should test your program carefully. Explain, in your report, what tests you carry out, how the program behaves, and whether it has passed such tests.

### 6.5.2 Hints

#### The main() Function

Your main() function should declare variables to represent the values of all the relevant quantities. Think carefully about naming these variables—note, for example, that variable names cannot involve subscripts, or greek characters. Think carefully about the type of each variable: is it to be used to represent a numerical value? If so, will that value be strictly integral, or may it have a fractional part?

The overall structure of your main() function will consist of some initialisation (assigning initial values to appropriate variables), followed by a while loop which carries out the repeated calculations of the new $x$ and $y$ co-ordinate values. This loop should continue just so long as the $y$ value is greater than zero. The repetition, or iteration, in your program should be achieved using a while statement.

It might be a good idea to start off with a highly simplified version of the main() function, having only a variable to represent $y$ and which just (for example) decrements the $y$ value by a fixed amount each iteration. This would allow you to check, at this early stage, that you have correctly implemented the while loop and that it terminates when appropriate.

---

3One way of capturing program output in a file is described in detail in the section above on Command Line Redirection. Similarly, a method of graphing this data is described in the section on Plotting a Graph.

4Technically, under the various idealizing assumptions we have made, it can be shown that the trajectory will be parabolic.

5This change will be negative, because $A_y$ is negative; to put that another way, what goes up must come down!
Functional Decomposition

Try to use functional decomposition to break your program into smaller, more manageable pieces. This will make it easier to write, test, and debug.

For example, you might define a new function of your own, which deals with printing out the results, for one cycle of your calculations. This can then be simply called or invoked from within the while loop of your main() function—which should make the main() function shorter and easier to follow. Note that this function will have to accept two arguments (being the current values for the x and y co-ordinates).

6.6 Session 4: Letter Frequencies

In a world where there are code makers there will, inevitably, also be code breakers. One of the simplest tools in the armory of a code breaker is a program to count the relative frequency of each letter appearing in the cyphertext. For any given language there will be a characteristic distribution of letter frequencies in the uncoded message (the “plaintext”). The most commonly used letter in English is e, by a wide margin; t is in second place, with a and o nearly tied for third; i, n and r are also very commonly used.

If we know that a coded message uses any kind of simple substitution cypher (such as the Caesar cypher) then just counting the relative frequencies will allow us to make a fairly good guess as to the letters which have substituted for the most common English letters. Often this would be enough to allow the remaining substitutions to be easily guessed.

Of course, real cyphers are much more sophisticated and harder to break than this (how would you tackle a Vigenere cypher for example?). But letter frequency counts still form an essential tool for code breaking, albeit in conjunction with many other techniques.

6.6.1 Exercise 1: Counting A! (30%)

Your first task in this session is to write a program which will read a file, and count how many times the character A (or a) occurs. Note that you have to create the file and to introduce some words. The program should ignore the distinction between upper and lower case, and should ignore all characters other than A (and a). When the file has been completely read, the program should output the total number of A characters found.

Redirection should be used to cause the program to read the appropriate file via stdin.

As usual, you should test your program carefully, and document any problems etc. in your report.

6.6.2 Exercise 2: More Counting... (30%)

Now enhance your program to count how many times each separate alphabetic character occurs (ASCII character set is available in Appendix A). Again, it should ignore distinctions between upper and lower case, and should ignore all non-alphabetic characters. So it should generate 26 distinct count values, one for each letter of the alphabet, and not counting non-alphabetic characters at all.

Note that you can use a single array, with 26 elements (where the elements are of type int, say), to hold all the counts. You will need to think carefully about how to access the correct position or slot in this array to get the correct count for any given letter found in the file.

When the file has been completely read, the program should output a table showing the count for each of the letters A to Z.

Again, you should test your program carefully, and document any problems etc. in your report.

6.6.3 Exercise 3: Statistics (20%)

Next you are required to enhance the program further so that it will also calculate the total number of all alphabetic characters read, and output, for each character, both the actual number of times it occurred, and the percentage this represents of the total number of alphabetic characters encountered.

Again, test and document the program carefully.

6.6.4 Exercise 4: Plotting a Histogram (20%)

Finally: develop a further version of the program, which, instead of outputing the relative frequencies as numbers, outputs a histogram. This will be made up of lines of varying width (made up of asterisk characters) one line for each letter, from A to Z. The letter with the most occurrences should be scaled to give a line with, say, 60 *’s. Then the lines for all the others should be shorter in proportion. The display might look somewhat as follows:

A ************
B *****
C **
D ***
—and so on.

Note that histograms would normally be plotted with the independent variable oriented *vertically*, whereas I have described an *horizontal* orientation here. If you have time, try to alter your program again, to yield a histogram with a vertical orientation (this is a substantially more challenging task).

### 6.7 Session 5: Exploring Pointers

This exercise is concerned with exploring the use of pointers and structures in C. Note that in this session you are not required to develop any new programs of your own—though you may wish to modify the programs you are given. Furthermore, all of the programs introduced here are totally *contrived* or *artificial*. They do not do anything useful whatsoever in themselves; but they are useful as vehicles for you to experiment with C pointer variables and pointer operations.

You must review the chapter on pointers in advance of this lab.

#### 6.7.1 Exercise 1: Basic Pointer Manipulation (40%)

Take a copy of the file "ptr0.c" (Appendix B)

First analyse the program by hand, and predict what you think the outcome will be. Record this analysis in your report. Then execute it and test your predictions. Add extra `printf()` statements as you deem necessary in order to check the values of variables at different stages during execution. If the results differ from your predictions then test more carefully until you can explain the discrepancies. Note your results in your report.

Repeat the procedure described above for the file "ptr1.c"

#### 6.7.2 Exercise 2: Advanced Pointer Manipulation (60%)

Consider the file "ptr_net.c" This first introduces the declaration of the structure shape called `bucket_s`:

```c
struct bucket_s
{
    struct bucket_s *black, *white;
    int quantity;
};
```

This says, simply, that any structure of this shape will have three members, called *black*, *white* and *quantity*. The *quantity* member is simply of type *int*; but *black* and *white* are pointers to (other?) structures of the same shape. Of course, they do not automatically point at any particular other structures: but they *potentially* can, if suitable assignments are made.

Four actual variables of this structured type or shape are then declared:

```c
struct bucket_s n0, n1, n2, n3;
```

*Each of these will have the three separate members already described. They have been declared outside of any function so they are visible and accessible to all functions.*

The rest of the program then involves setting up particular linkages between these structures (via the *black* and *white* members) and using these linkages to *indirectly* access various members in different ways. I repeat that none of this achieves any useful purpose: the only purpose of the exercise is for you to test whether you have a sufficient grasp of what’s going on to be able to accurately predict what members will be accessed in each case.

First try to draw out on paper the relationships between the structures *n0, n1, n2 and n3* immediately after execution of the statement at line 77 (the invocation of the function `red_connections()`). Do this by *analysing* the program manually, *not* by *executing* it. The diagram should show graphically what each pointer member of each structure is pointing at.

Next attempt to predict the values of the *quantity* members of each structure immediately after execution of the statement at line 78 (the first invocation of the function `massage()`). Check your predictions by executing the program (you’ll have to add some `printf()` statements to print out the values you are interested in).

*Explain any discrepancies.*

Note that I really do mean that—do not go any further unless and until you can explain all discrepancies between your predictions and the actual results.

Now draw a new diagram showing the relationships between the structures immediately after execution of the statement at line 79 (the invocation of the function `blue_connections()`). Repeat this diagram one last time for the situation immediately after execution of the statement at line 80 (the invocation of the function `yellow_connections()`).
Finally, predict the values of the quantity members of each structure immediately after execution of the statement at line 81 (the second invocation of the function massage()). As before, check your predictions by executing the program, and explain any discrepancies.
Chapter 7

General Examination Instructions

7.1 Overview
These are general instructions applying to the Laboratory Exams associated with the module Software Engineering 2, for the session 2005/2006.

This module is assessed by way of a supervised Laboratory Examination, taking place at the end of the semester.

In the exam each student works separately on an individual PC. Instructions (the exam “paper”) are provided online. During the exam each student must prepare a report which is submitted electronically both via email and on diskette. These reports form the sole basis for marking the exam.

The class is too large to be accommodated in the laboratory in one session. Thus, the exam is administered in several separate sessions. The details of the exercises required in each session will be different.

These general examination instructions are made freely available to the students in advance.

The online version of these instructions provides links to the distinct, detailed, instructions for each session of the exam. But, in general, each such link will be operational only for the duration of that particular session.

Note that the exam is “open-book”. Students may bring in whatever notes, textbooks etc., which they wish. They may also access any online resources of their choice. They only restriction is that, during the exam session, they may not communicate with any other person, inside or outside the laboratory. This is supervised directly by the invigilators, and also indirectly by logging relevant system activities.

7.2 Regulations
These exams takes place under the full University Regulations. Note the following particular points in relation to breaches of the Regulations regarding examinations:

Serious academic offences (e.g., cheating) may, in addition to such reduction in marks or other action as the Committee may recommend to an Examination Board, be punished by suspension from the University in accordance with the provisions of this Code. Should the offence be repeated it shall result in expulsion.

7.3 Preparation
You must bring your DCU student card to the exam.

You will be provided with diskettes for submission of your final report. You will not be issued with any other materials (including writing paper) by the invigilator(s).

You may bring in such textbooks, manuals, notes, diskettes, writing paper, pens, calculator etc. as you wish. You will not be permitted to use or refer to materials brought in by any other student. You will not be permitted to leave the laboratory in order to get additional materials. It is entirely your own responsibility to provide all materials or resources which you wish to use in the exam.

7.4 Conduct of the Exam
You should be present in the Laboratory at least 10 minutes before the scheduled time for the exam. You will be assigned a computer by the invigilator. You may not subsequently move from this position without permission from the invigilator. You must
leave your student card in clear view throughout the exam.

You may not switch on or operate your computer until told to do so by the invigilator(s).

During the exam you may not communicate in any way with any other person, inside or outside the laboratory, by any means whatever—with the sole exception of the invigilator(s). Any attempt to engage in unauthorised communication will be regarded as a severe breach of the examination regulations and will be dealt with accordingly.

Each exam will be of three hours duration.

7.5 Equipment Failure

If you believe that the laboratory equipment (hardware or software) is not operating correctly then you should report this to an invigilator immediately, and follow whatever instructions you are then given.

If there is a general equipment failure—for example affecting the mains power, or the network—then you must remain seated, and silent, in your assigned position, until given directions by the invigilator(s).

7.6 Report Preparation

During the exam, you must prepare a report, in accordance with the Laboratory Report Guidelines.

It is recommended that you plan your allocation of time between the parts carefully in advance, and stick to this allocation during the exam. Ensure that you allow adequate time for finalising and proof-reading the report at the end.

7.7 Completion

When you have completed the exam, or when directed to do so by the invigilator, you must:

- Fill in the following information on the paper label on each of the two exam diskettes which you have been given:
  - The module code and title (i.e., EE105: Software Engineering 2).
  - Your name.
  - Your student ID number.
  - The date and time of your exam.
- Save a copy of the report, under the name report.txt in a directory called ee105 on each of the two exam diskettes which you will be given by the invigilator. Nothing else may be saved on these diskettes. Check carefully that the file has been correctly saved on both diskettes. These diskettes will be collected by the invigilator(s).
- Submit a copy of the report, by email. The email address for report submission is: ee105-exam@eeng.dcu.ie
  and will be given in the specific instructions for your session. If in doubt, ask the invigilator(s).
  The email subject heading must be EE105 Exam Report.
  It is recommended that you include yourself as an additional recipient for this message, and that you save a copy of the received message on your own diskette. However, this is not a required element of the exam, and will not be checked or assessed.

At the end of the exam you must remain seated at your assigned position in the laboratory until told you may leave by the invigilator(s). Please co-operate with the invigilator(s) in administering the exam. In particular, at the end of the exam, please remain silent (even after your diskettes have been collected) until you are told by the invigilator(s) that you may leave the laboratory.

7.8 Links to the Specific Exam Exercises

Links to the specific exam exercises for each session of each exam are provided in this section of the online version of the instructions. These links will be operational only for the duration of each exam session.
Copyright

This Hypermedia Document is copyright, © 1994-2000, by Dr. Barry McMullin © 2001-2003, by Dr. Noel E. O'Connor © 2003-2005, by Dr. Gabriel-Miro Muntean, © 2005-2006, by Dr. Cristina Hava Muntean. Permission is hereby granted to access, copy, or store this work, in whole or in part, for purposes of individual private study only. The work may not be accessed or copied, in whole or in part, for commercial purposes, except with the prior written permission of the authors.
Appendix A

The ASCII Character Set

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
! " # $ % & ' ( ) * + , - . /

48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
0 1 2 3 4 5 6 7 8 9 : ; < = > ?

64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
@ A B C D E F G H I J K L M N O

80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
P Q R S T U V W X Y Z [ \ ] ^ _

96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111
' a b c d e f g h i j k l m n o

112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
p q r s t u v w x y z { | } ~
Appendix B

Skeleton Report File

SOFTWARE ENGINEERING 2
ACADEMIC YEAR 2005/2006
LABORATORY REPORT

Student Name: Barry Murphy
ID Number: 54000001
Session Date: 18th February 2006
Session Number: 2
Session Title: Computation

Exercise 1: Area of a Circle

Plan

The aim of this lab session is to write a C program that calculates the area of a circle, to compile the program and to test it. A radius value is read from the user and used to calculate the area of the circle. The result is printed to the standard output (on the screen). Several tests are run to check if the program performs the right computation for different input values.

Development

The first step in the development of my program was to include the standard C libraries (stdio.h, stdlib.h, math.h) that contain the functions used in my program.

I also declared the variables that I will be used in the program, in this case two floats, the radius and the area of the circle. A constant variable called pi was also defined.
Then, I started to write the main function of the program.

The radius of a circle was requested from the keyboard using the "printf" statement and the value was read using the "scanf" statement. The area of the circle is calculated using this value.

The result is then printed to the screen, using the "printf" statement.

My development resulted in the following source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define pi 3.141592654

int main(void) {
    float radius, area;
    printf("Please enter the radius of your circle:	")
    scanf("\f", radius);
    area = pi * pow(radius2);
    printf("The area of the circle is:\t%f Units Squared", area);
    return 0;
}
```

Test
----
I compiled and tested the program using the bcc32 compiler.
I was presented with the following two errors:

Error E2379 area.c 11: Statement missing ; in function main
Error E2451 area.c 13: Undefined symbol 'radius2' in function main
Warning W8065 area.c 13: Call to function 'pow' with no prototype in function main
*** 2 errors in Compile ***

The above errors were corrected by inserting a ";" at the end of the printf statement on line 11. On line 13 an error was received due to incorrect use of the pow function that took radius2 as parameter instead of radius. The program was then recompiled (no errors received) and executed for runtime testing.

During program execution, a runtime error was encountered after the input data was introduced. On examination of the code, it was discovered that "\f" was omitted from the "scanf" statement, on line 12, causing the program to address a memory location that it did not have access to.

The program was now recompiled and tested with different values for the radius of a circle.

The results are presented next:

radius = 4.0 ---- Area = 50.265
radius = 30.25 ---- Area = 2912.89
radius = 0 ---- Area = 0.00

Conclusion
---------
During this lab session I learned about some of the functions contained in the "math.h" library. Also the errors I obtained helped me understand the difference between compile and runtime errors. The final version of the C source code for Exercise 1 is attached as "area.c" file.

---------

Exercise 2: Cryptanalysis.

Plan
----

Development
----------

Test
----

Conclusion
---------

---------

Exercise 3: The Vigenere Cypher.

Plan
----

Development
----------

Test
----

Conclusion
---------
Appendix C

Program Listings

C.1 sum0.c

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

int main(void)
{
    double x,y,sum;

    printf("Please enter a number: ");
    scanf("%lf", &x);
    printf("Please enter a number: ");
    scanf("%lf", &y);

    sum = x + y;

    printf("The sum of %f and %f is: %f\n", x, y, sum);
    printf("Goodbye!\n");

    return(EXIT_SUCCESS);
}
```
C.2 sum1.c

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

double get_input(void)
{
    double input;
    printf("Please enter a number: ");
    scanf("%lf", &input);
    return(input);
}

int main(void)
{
    double x, y, sum;
    x = get_input();
    y = get_input();
    sum = x + y;
    printf("The sum of \%f and \%f is: \%f\n", x, y, sum);
    printf("Goodbye!\n");
    return(EXIT_SUCCESS);
}
```
C.3  sum2.c

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

double get_input(void)
{
    double input;
    printf("Please enter a number: ");
    scanf("%lf",&input);
    return(input);
}

void put_output(double num1, double num2)
{
    double sum;
    sum = num1 + num2;
    printf("The sum of %f and %f is %f\n",num1,num2,sum);
    printf("Goodbye!\n");
    return;
}

int main(void)
{
    double x,y;
    x = get_input();
    y = get_input();
    put_output(x,y);
    return(EXIT_SUCCESS);
}
```
C.4  ptr0.c

1 /*
2 ptr0.c
3
4 B.McMullin
5 30th April 1996.
6 */
7
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <stdlib.h>
11
12 int main(void)
13 {
14    int i,j,k;
15    int *p, *q, *r;
16    int **s, **t, **u;
17    i = 7;
18    j = 23;
19    k = i * j;
20    p = &k;
21    q = &i;
22    r = p;
23    *r = k;
24    *p = (*p + *r) * 5;
25    i = *(&j);
26    s = &q;
27    u = &p;
28    t = s;
29    p = *t;
30    *p = **u * i;
31    **s = i + j + k;
32    j = 0;
33    r = p = &j;
34    s = &r;
35    t = s;
36    u = &p;
37    p = &k;
38    **u *= (*p * **s) * (i * j * k);
39    /* What are the final values of i, j, k? */
40    /*
41       printf("i = %d\n", i);
42       printf("j = %d\n", j);
43       printf("k = %d\n", k);
44    */
45    return(EXIT_SUCCESS);
46 }

48
C.5  ptr1.c

1 /*
2 ptr1.c
3 
4 B.McMullin
5 30th April 1996.
6 */
7
8 #include <stdio.h>
9 #include <stdlib.h>
10 
11 int main(void)
12 {
13  int i, j, k,
14    *p, *q, *r, *s,
15    **t, **u, **v;
16  
17    i = 0;
18    p = &k;
19    q = &i;
20    r = &j;
21    s = r;
22    t = &q;
23    u = &p;
24    v = &s;
25    j = *q;
26    k = j;
27    
28    while ((j <= 15) && (j >= -15))
29    {
30        j++;
31        s = r;
32        r = q;
33        q = p;
34        p = s;
35        v = u;
36        u = t;
37        t = v;
38        (**t)--;
39        (**u)++;
40    }
41    
42    /* What are the final values of i, j, k? */
43    
44    /*
45     printf("i = %d\n", i);
46     printf("j = %d\n", j);
47     printf("k = %d\n", k);
48    */
49    
50    return(EXIT_SUCCESS);
51 }
A test program illustrating structures and pointers.


#include <stdio.h>
#include <stdlib.h>

struct bucket_s
{
    struct bucket_s *black, *white;
    int quantity;
};

struct bucket_s n0, n1, n2, n3;

void red_connections(void)
{
    n0.black = &n1;
    n0.white = &n3;
    n3.black = n0.black;
    n3.white = &n0;
    n1.white = NULL;
    n1.black = &n2;
    n2.white = n3.black;
    n2.black = n1.white;
}

void blue_connections(void)
{
    n0.white = &n1;
    n1.black = &n0;
    n0.black = n0.white->black;
    n1.white = n1.black->white;
    n2.black = &n3;
    n3.black = &n2;
    n2.white = n2.black->black;
    n3.white = n3.black->white;
}

void yellow_connections(void)
{
    struct bucket_s *p;
    p = n0.white;
    n0.white = n1.white;
    n1.white = n2.white;
    n2.white = n3.white;
    n3.white = p;
    p = n0.black;
    n0.black = n1.black;
}
n1.black = n2.black;

n2.black = n3.black;

n3.black = p;

}

void massage(void)
{
 n0.black->quantity = 2 * n3.white->quantity;
 n1.black->quantity = 2 * n2.white->quantity;
 n0.white->quantity = 2 * n2.quantity;
}

int main(void)
{
 n0.quantity = 2;
 n1.quantity = -3;
 n2.quantity = 4;
 n3.quantity = -5;

 red_connections();
 massage();
 blue_connections();
 yellow_connections();
 massage();

 /*
 printf("n0.quantity: %d\n", n0.quantity);
 printf("n1.quantity: %d\n", n1.quantity);
 printf("n2.quantity: %d\n", n2.quantity);
 printf("n3.quantity: %d\n", n3.quantity);
 */

 return(EXIT_SUCCESS);
}
Appendix D

How to use Borland C++ Compiler

A step-by-step description on how to write a C program in a file (filename.c), to compile the program and to run the executable file is provided in this document.

**Step 1: How to edit a filename.c file using TextPad application**

Launch the TextPad text editor as follows:

*Start > Programs > TextPad*

Write the C code for your application in the opened file as presented in Figure 1.

![Figure 1. A file with C code lines that was edited using TextPad](image)

**Step 2: How to save the file**

In order to save the data, choose *File-* > *Save* option and indicate the name of the file (e.g. hello.c) as well as the folder of your choice in your home directory (e.g. h:’ee105’) (see Figure 2).
Step 3: How to open a command prompt window for running the Borland C++ Compiler

In order to compile and run the C source code from the hello.c file, you have to launch the Borland C++ compiler as follows:

Start > Programs > Borland C++ 5.5 > bcc55

A command prompt window will appear and will point to the temp directory on the C: drive (Figure 3).

You must now navigate to the directory where you have saved your file as follows:

> h:
> cd yourdirectory       (for example:  > cd ee105)
Step 5: How to compile the hello.c file using Borland C++ Compiler

You can compile a file with a C source code by typing “bcc32 filename.c” at the command prompt. This should result in the following output if everything has worked:

![Figure 5. Compile the hello.c file using the bcc32 command](image)

Step 6: How to run a C program

After the program was compiled and no errors were listed by the Borland C++ Compiler, an executable file is generated (e.g. hello.exe). This file is located in the same directory as the file with the source code (hello.c). You can run the executable file by typing filename.exe. (i.e. hello.exe) and the output of the program (if any) is listed on the screen.