

# DUBLIN CITY UNIVERSITY

**School of Electronic Engineering**

**Module: EE102**

*Software Engineering 1*

## **Laboratory Manual**

Academic Year 2005-2006: Semester 1

Module Co-ordinator: Dr. Gabriel-Miro Muntean  
Room: S326  
Telephone: (01) 700 7648  
Email: [ee102@eeng.dcu.ie](mailto:ee102@eeng.dcu.ie)

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	What is Software? . . . . .	4
1.2	Module Overview . . . . .	4
1.3	Instructors . . . . .	4
1.4	Textbook . . . . .	4
1.5	Computer Facilities . . . . .	5
1.6	Delivery . . . . .	5
1.7	Tutorial . . . . .	5
1.8	Assessment . . . . .	6
1.9	The Email Conference: <code>ee102-talk</code> . . . . .	6
1.10	Additional Resources . . . . .	6
<b>2</b>	<b>Laboratory Exercises</b>	<b>8</b>
2.1	General Comments on Lab Work . . . . .	8
2.1.1	Workload . . . . .	8
2.1.2	Logbook . . . . .	8
2.1.3	Report Guidelines . . . . .	8
2.2	Session 1: Introducing the <i>Internet</i> Computer . . . . .	8
2.2.1	Preliminaries . . . . .	8
2.2.2	Using email . . . . .	9
2.2.3	Winding Down . . . . .	11
2.2.4	Homework! . . . . .	11
2.3	Session 2: File Systems, Text Editing and E-Conferencing . . . . .	11
2.3.1	The <code>bash</code> Command Line Interface (shell) . . . . .	11
2.3.2	Files and Filesystems . . . . .	12
2.3.3	Exploring Filesystems . . . . .	13
2.3.4	Working With Text Files . . . . .	14
2.3.5	Creating and Editing Text: Using <code>PICO</code> . . . . .	15
2.3.6	Creating and Editing Text: Using <code>Textpad</code> . . . . .	16
2.3.7	Electronic Conferencing . . . . .	16
2.3.8	Homework . . . . .	16
2.4	Session 3: Introducing <code>C</code> . . . . .	17
2.4.1	Preparation . . . . .	17
2.4.2	Background . . . . .	17
2.4.3	Exercise 1: Hello World! . . . . .	17
2.4.4	Exercise 2: Branching out! . . . . .	18
2.4.5	Conclusion . . . . .	18

2.5	Session 4: Quiz Time . . . . .	19
2.5.1	Introduction . . . . .	19
2.5.2	Exercise 1: Analysis . . . . .	19
2.5.3	Exercise 2: Synthesis . . . . .	20
2.6	Session 5: Ever Decreasing Circles. . . . .	20
2.6.1	Introduction . . . . .	20
2.6.2	C Data Types and Operators . . . . .	20
2.6.3	Introducing the C Standard Library . . . . .	21
2.6.4	Exercise 1: Analysis . . . . .	22
2.6.5	Exercise 2: Synthesis . . . . .	22
<b>3</b>	<b>Laboratory Report Guidelines</b>	<b>23</b>
3.1	General . . . . .	23
3.2	Typing Up . . . . .	23
3.3	Structure . . . . .	24
3.3.1	Heading . . . . .	24
3.3.2	Plan . . . . .	24
3.3.3	Development and Test . . . . .	24
3.3.4	Conclusion . . . . .	25
3.4	Including Source Files . . . . .	25
3.5	Problems . . . . .	26
3.6	Presentation . . . . .	27
<b>4</b>	<b>Copyright</b>	<b>28</b>
<b>A</b>	<b>Program Listings</b>	<b>29</b>
A.1	mquiz0.c . . . . .	29
A.2	circle0.c . . . . .	30
<b>B</b>	<b>Skeleton Lab Report</b>	<b>31</b>

# Chapter 1

## Introduction

This is the top-level, online, hypermedia documentation for module EE102, *Software Engineering 1*, for the session 2005-2006.

This is a required module for all students enrolled for year 1 of the following academic programmes offered in Dublin City University:

- Common Entry into Engineering
- B.Eng. in Electronic Engineering
- B.Eng. in Information and Communications Engineering
- B.Eng. in Digital Media Engineering
- B.Eng. in Mechatronic Engineering
- Engineering Foundation Programme

These notes are also available online at:

<http://www.eeng.dcu.ie/~ee102/2005-2006/manual/>

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

### 1.1 What is Software?

The vast majority of modern technological products—from washing machines to communications satellites, CD players to industrial robots—include at least one, and often many more, embedded *microcontrollers*. A microcontroller is, in effect, a general purpose digital computer implemented as a single integrated circuit (IC). The specific behaviours or functionalities of the completed product are then realised by providing these microcontrollers with appropriate sequences of instructions, or *software*.

Software is thus a crucial foundation for virtually all modern technological products—and software development is an essential and pervasive skill for every engineering discipline. This is reflected in

the design of the engineering programmes in DCU, where Software Engineering takes a prominent role in all stages—beginning with this very important introductory module.

### 1.2 Module Overview

This module first develops basic computer literacy—keyboard skills, the pointer device (mouse), computer based text processing, electronic communication (email), and accessing electronic information resources (the Web).

This is followed by a rudimentary introduction to *Software Engineering*—which is an engineering discipline concerned about the development of *software* or *programs* for digital computers.

The module is 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.

### 1.3 Instructors

The Module Co-ordinator is Dr. Gabriel-Miro Muntean. There will also be two laboratory demonstrators who will be available to assist you during practical laboratory sessions.

### 1.4 Textbook

All students enrolled for this module are *encouraged* to purchase an introductory C programming textbook. A variety of these are available in the campus bookshop. Here are a few specific titles you might consider:

- **The C Programming Language**, *Kernighan and Ritchie*, Second Edition. Prentice Hall, 1988.

This is the book I personally rely on as a reference whenever I am doing any significant C programming work. Kernighan and Ritchie *invented* the C language, and they also give a very clear and precise exposition of it. I really like this book.

However: novices generally find this one a bit daunting. It *is* usable by beginners, but it is more oriented for advanced users. It's also rather expensive .

- **Applied C: An introduction and more**, Fischer, Eggert, Ross, McGraw-Hill, 2001.

This is a very comprehensive text which covers many aspects of the C programming language from fundamentals right up to advanced programming concepts. It contains many worked examples and exercises. Whilst this book may be expensive, I believe it constitutes a good investment, not just for first year, but also for virtually every other year of your degree programme as well. As such, I would recommend it.

- **C Program Design for Engineers**, Hanley, Koffman and Horvath, Addison-Wesley, 1995.

I quite like this text. I think it should be more accessible than K&R. It is also less expensive than K&R. It has good comprehensive discussions, is well organised, and provides *lots* of small self-assessment tests. It's also specifically oriented toward *engineering* applications of C programming. I'd recommend it.

- **Problem Solving and Program Design in C**, Hanley and Koffman, Second Edition, Addison-Wesley, 1996.

This is a more recent text, from two of the same authors as the previous one. It is a more generic book (not focused specifically on engineering); but it has also been updated and redesigned and that may be a plus.

- **Structured C for Engineering and Technology**, Adamson, Antonakos & Mansfield. Second Edition. Prentice-Hall, 1995.

This was specifically recommended by one of the students from the 1995/96 academic year.

I suggest holding off on making a decision on which text book to purchase for a number of weeks into Semester 1. At this stage you should be able to make an *informed decision* for the level of text you need, based on your understanding of the material covered in lectures and laboratory sessions.

Since I'm not a beginner, it is difficult for me to assess texts. So I'd appreciate it if members of the class could provide feedback about their own experiences with these, or other texts, as the module progresses.

In any case, as well as purchasing some specific text, you should consult relevant books in the library, and other materials on the Web, on an ongoing basis.

## 1.5 Computer Facilities

Labs associated with this module will take place in *Computer Aided Engineering* (CAE) laboratory of the School of Electronic Engineering, located in room S143/S144, in the Research and Engineering Building. Each student must attend a 3 hour lab session every second week. A detailed timetable will be made available on-line on the DCU website . Check the on-line version of the timetable for eventual changes during the semester.

For much of the time during semester, the CAE lab is scheduled exclusively, or on a priority basis, for specific class groups. Please respect this schedule, and yield your place in the lab if a student from a higher priority class group requests it. Apart from these classes you are welcome to use the facilities of the lab anytime and how long you like to.

Note that the School provides laser-quality printing facilities only by using the computer connected to the printer. However the service is not free, credit points having to be purchased in advance from a special machine placed in the lab that accepts only notes. You are required to indicate your student ID when both purchasing the credit points and login to the machine for printing.

## 1.6 Delivery

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

## 1.7 Tutorial

In addition to lectures and lab sessions, an informal one hour tutorial session will be offered each week. At least one of the instructors and/or demonstrators will be present at the tutorial. This is an excellent opportunity to look for help in completing any lab exercises which you have outstanding, particularly if

you are stuck with some problem you simply can't seem to overcome.

You can, of course, submit queries by e-mail at any time (see the information on the Email Conference associated with the module); but this may be impractical for various reasons (e.g., if your problem is that you can't get the email system to work!). In such cases, the tutorial is your opportunity to get the problem sorted out.

## 1.8 Assessment

The module will be continuously assessed over the entire semester. The assessment structure is as follows:

- Submission (by email) of formal reports on selected laboratory sessions (30%);
- A major final *assignment* at the end of the semester (70%).

**Caveat:** the module assessment structure is constantly under development and as such may change slightly during the initial weeks of the module. These changes (if any) will be announced in lectures and reflected in the on-line version of these notes.

To *Pass* the module you must achieve an aggregate score of not less than 40%.

If you *Fail* this module, you will be offered a single opportunity to make good over the Summer vacation period (before the Autumn diet of exams).

If you *still* do not achieve a *pass*, then you will *not* be eligible to progress to the next year of your programme. Instead you will be required to *re-enroll* for full attendance (paying full module fees) on the Software Engineering 1 module.

Thus, it is very important that you achieve a *Pass* grade in this module; and it is very much preferable to achieve this at the first attempt!

Detailed notes are available for each lab session in Chapter 2. Formal lab reports will be required for lab sessions 3, 4 and 5. You should consult the guidelines on report format in Chapter 3 for more information on what will be required in these formal reports.

The format of the final assignment, and its corresponding report, will be very similar to the format of the normal lab exercises. You will work alone. Instructions will be accessed 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 assignment.

Work on the assignment will be unsupervised. However, a sample of students may be required to attend an oral interview in which they must demonstrate that all work submitted is their own.

**If the examiner concludes that a student has submitted work which is not his or her own, then this will be classified as a case of *plagiarism* and dealt with through the University disciplinary procedures. Penalties for plagiarism are extremely severe.**

## 1.9 The Email Conference: ee102-talk

The module has an associated *email conference*, called ee102-talk. If you are not yet familiar with the idea of an email conference, please consult the document *Internet Email Conferences: A Brief Primer* at:

<http://www.eeng.dcu.ie/mailling/index.html>

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://maillist.eeng.dcu.ie/pipermail/ee102-talk/>

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

[ee102-talk@maillist.eeng.dcu.ie](mailto:ee102-talk@maillist.eeng.dcu.ie)

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 ee102-talk conference, you can address private email to me at:

[ee102@eeng.dcu.ie](mailto:ee102@eeng.dcu.ie)

## 1.10 Additional Resources

The following additional materials are also available:

- **C Source File Directory:**

<http://www.eeng.dcu.ie/~ee102/2005-2006/src/>

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

- **UNIXhelp:**

<http://www.eeng.dcu.ie/local-docs/UNIXhelp1.3/>

This is introductory documentation for using the *Unix* operating system. Although we will generally be using Windows 2000 rather than Unix, we will be using the **bash** Command Line Interface (CLI), which originated on the Unix system. Much of the **UNIXHelp** documentation is therefore quite useful and relevant even under Windows OS.

The **GNU C Compiler** (**gcc**) is recommended for use with this module. This is a high performance compiler which is available *free* for a wide variety of platforms.

In particular, for 32-bit PC Microsoft platforms (Windows-95, Windows-98, Windows NT, Windows 2000 or Windows XP), I recommend the **Cygnus GNU-Win32** port of **gcc**:

<http://www.cygnus.com>

**GNU-Win32** is a comprehensive package, with many additional tools over and above the compiler. For example, it includes the **bash** Command Line Interface (CLI).

Finally, for those who are a bit more adventurous, you may want to consider installing the (free) **Linux** operating system:

<http://www.redhat.com/>

This is a high performance, 32-bit operating system available for a variety of hardware platforms (including Intel based PCs). Standard distributions include the full **GNU** compiler and many other development tools. **Linux** is my own preferred environment for all serious software development work.

**Caveat:** installing Linux can be a major undertaking (particularly if you also want to keep your existing operating system) and the process should not to be taken lightly! Make sure to research this course of action thoroughly so that you know what you are doing in advance.

A very wide range of additional resources is available elsewhere online, on the general subjects of programming and the **C** language. Pointers to many of these are available through the Yahoo indexing service at URL:

<http://www.yahoo.com/>

In particular, check the section on “Computers and Internet: Programming Languages: C/C++”

Searching the web, especially using Google’s search engine is highly recommended. You can access it at the following URL:

<http://www.google.com/>

Additional useful Internet resources dealing with specific topics will be indicated via the econference facility as the semester progresses.

# Chapter 2

## Laboratory Exercises

### 2.1 General Comments on Lab Work

The laboratory exercises are crucial to this module: you will probably learn as much from the time you spend in the laboratory as from lectures and tutorials.<sup>1</sup>

#### 2.1.1 Workload

Each student attends one three-hour scheduled lab session every two weeks. *You should plan to spend at least three further hours of private study time working on a computer 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.

#### 2.1.2 Logbook

You should buy a new copybook or notebook for use as a logbook for this module. Bring this along to all module activities—lectures, labs, tutorials etc.—and use it to keep your own personal notes and questions relating to these activities. However, 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. This will be explained in more detail during the lab sessions.

---

<sup>1</sup>Of course, that does *not* mean that you can afford to miss lectures or tutorials. . .

### 2.1.3 Report Guidelines

You should consult Chapter 3 for advice on how to prepare lab reports. However, note that you will not be required to start submitting *formal* lab reports for this module until session 3.

### 2.2 Session 1: Introducing the *Internet Computer*

#### 2.2.1 Preliminaries

The first two lab sessions are concerned simply with introducing general purpose hardware and software tools which will be used in the remainder of the module. If you are reading these notes *online*, then you must have successfully completed the first phase of the exercise—congratulations!

Stop at this point and read back over what you have written in your logbook. Make sure you have a reasonably complete record of what you have done so far. As a guideline, ask yourself whether the notes you have made would be adequate to allow you to redo everything without further guidance or assistance.

In particular you should have some record of:

- The rules and the timetable applying to users of the computer laboratory.
- The *Acceptable Use Policy* regarding the School's computer and networking facilities.
- The component physical parts of the PCs—their names and functions.
- The procedure for “booting up” a PC, if it is powered down.
- Your personal *username* and *password* which you will use to access the School computer systems.

Note that the username/password information used for the School's computer systems is the same as those used to access the central DCU email system. They were given to you on a printed sheet of paper at registration. As such, you will need to remember the username and password as you need for accessing the School's computer facilities (i.e. actually logging in or gaining access to a machine) and for using email, once you are logged in.

Remember to treat your password carefully—ensuring both that you do not lose them, and that *nobody* else gains access to it.

If you mislay your password, or you suspect, for any other reason, that its secrecy may have been compromised, then contact the Computer Services Department or a technician *immediately* to have your account secured again.

If you mislay your DCU password, or you suspect that its secrecy has been compromised, then contact the Computer Services Department immediately.

- The procedure for *logging in* to the **Windows 2000** environment, in the standard configuration used in the School of Electronic Engineering. This also gives you access to the School *network*, including your own personal *home directory* (private file storage area). In general this area will be known as your H: drive.
- How to access the **Windows 2000 Help** facility, including searching and using hypertext links.
- How to launch (start, run, execute) and use the **KP Typing Tutor**.
- How to launch and use the **Web browser** (Microsoft Internet Explorer for example).
- How to access the *online* version of these hypermedia notes for the *Software Engineering 1* module. Does the online version have any advantages relative to the paper version? Any disadvantages?
- The procedure for *logging out* of the **Windows 2000** environment.

- The procedure for safely *powering down* the PC. However this is not required after each use of computers as other students may want to login on the same machine after you.

If your logbook fails to mention any of these things, take time *now* to make some additional notes.

## 2.2.2 Using email

Every student enrolled in DCU is provided with an email account. During the course of the semester, email is used to broadcast a variety of administrative information such as class changes, exam timetables etc. It may also be used for more general communication from your lecturers and other students relating to any of the modules you are following. You can also use your email account to *send* messages to other students and staff. This includes, for example, participation in the **ee102-talk** email conference associated with the **EE102** module.

Clearly, it is important that you master the basic techniques of sending and receiving email as soon as possible.

As a free bonus, your email account can also be used for personal communication with colleagues, friends and relations, within DCU and, indeed, anywhere else in the world. Of course, all such use is subject to the School's *Acceptable Use Policy* as already discussed; and the facility for external email can and will be withdrawn in the event of breaches of this policy . . .

The email system run the Computer Services Department is a web-based email facility. This means that you access your email using an Internet browser. The advantage of this is that you do not need a separate program to access email and the you can access your email from anywhere in the world, assuming you have an Internet-enabled computer.

The Internet browser we will use is called Internet Explorer. To run Internet Explorer find this program under the Start menu and click on the program name. You can also double-click with your mouse on the Internet Explorer icon from your desktop. When the browser starts it brings you to a default location on the world-wide web. This location has been set as the School of Electronic Engineering's home page. To access the DCU mail system you need to "point" your browser at the login page for the mail system:

<http://mail.dcu.ie>

This page can also be accessed under the Student Information section of the main DCU web page:

<http://www.dcu.ie>

At the login page, enter your **username and password** – this is the username and password you were provided with at registration – and click on the “Login” button.

You will then be presented with the DCU email interface. The interface is divided into a number of different sections each of which provides different functionalities.

The section in the centre is a work space which displays whatever function of the email system you are currently using. This defaults to a view of your InBox upon logging in. This is where all new emails will appear. By default, emails are listed by date of receipt with most recent first, however you can sort your InBox by sender, subject, or size. There are a number of buttons provided on this part of the interface allowing you to delete emails, forward emails and save emails to a particular folder – see below.

On the left hand side is a set of functionalities grouped by the labels Operations, Settings and Folders. Operations provides the functionalities of checking for new mail, composing an email to send, and logging out of the system. The Folders section provides functionality for managing your email by, for example, grouping all emails related to a certain subject in a single subject in one place called a folder. You can define your own folders to help you organise your email. For example, you might create a folder called EE102 which you could use to store all messages related to this module. The Settings section allows you set certain preferences for dealing with email. For example, in this section you can set up and add entries to an address book, so that you can keep track of email addresses of friends, family, and University colleagues.

As it happens, there should already be some incoming mail waiting for you. Select the message you wish to view by clicking on it in the InBox. The text of the message along with information about the sender and date of receipt etc should appear in the space below your InBox. Take some time now to read the emails in your InBox. Note that when you select a message for viewing, you can subsequently select to Reply to the message, to Edit the message, etc. Do not reply to any messages until you have completed the following step.

OK now you are ready to send some email. As already mentioned, the email facility allows you to

send messages to anyone, anywhere in the world, almost instantaneously... but we'll start on a slightly more modest scale. Your first email message will be sent to *yourself!*

This is not as silly an idea as it may appear: by sending a message to yourself, you will be able to establish both that you can send and receive email successfully. After all, there is no point in trying to communicate any further afield unless that much, at least, is working for you!

Your personal email address has the form:

`usernamenumber@mail.dcu.ie`

where, of course, you replace the word “username” with your own username and “number” with the one received at registration. If you are unsure of your email address, you can find it in the Settings section.

So: following the instructions of the lab demonstrator (or figuring it out yourself!) compose and send an email to yourself. Enter your own address in the **To:** field, and enter something short in the **Subject:** field (such as “**My first email!**”). Now type some text for your message proper—just enter any short text you like. It could be a verse of a song, a limerick, a joke, or just some nonsense text; but keep it to no more than 3-4 lines. Remember to “sign” your message at the bottom. Your name will automatically appear at the top of the message anyway, but it is still polite to sign off the message at the bottom too. Email messages tend to be a bit less formal than normal business letters. You might use something like this, for example:

That's all for now,

Cheers,

- John!

And finally: *send it.*

Now for the fun bit: check to see has it been delivered safely. Note that although the message should be delivered more or less instantly, it may not appear in your InBox for some time. You can force a new check of your InBox by using the “Check mail” option.

*It is really important that you succeed in sending and receiving at least this one email message during this first lab session. So, if you get into difficulties here, please get help from a demonstrator!*

Email will be used by staff to communicate with you about all aspects of your courses—from timetable changes to lab groups to scheduling tests or contacts with your personal tutor. So it is important that you *immediately* develop the habit of checking your email regularly.

*I recommend that you check your email at least three times each week; better yet, try to do it at least once each day—it only takes a few minutes.*

### 2.2.3 Winding Down

Well, that more or less completes this first rather exhausting lab exercise. Just tidy up a little before finishing. You can delete your first test message and if you have time, exchange a few emails with some of your neighbours in the lab.

Finally, *log out* of your email account, close the Web browser and and log out from Windows 2000.

### 2.2.4 Homework!

As indicated in the general information for the module, you are expected to put in roughly as much time again in private study as you spend in the lab—i.e., three hours over the next fortnight. Use this time for the following:

- Review again everything that was covered in this first lab session. Make sure that you are comfortable booting up the PC, logging in to Windows 2000, starting up the **web browser** (e.g. Internet Explorer), and finding the online version of these notes on the web. Explore a little of the other information that is available.
- Unless you are already a proficient typist, plan a programme of sessions using a text editor, over the next month, to gain a minimal degree of typing skill. This will benefit you repeatedly in the years to come!
- Check your email several times—at least 6 times before the next lab session!
- Exchange some email messages with other people in your class. If you know anybody outside DCU who has an Internet email address, exchange some emails with them also.
- If you run into problems with any of this, ask for help! You can do this immediately after any of the lectures, or by making an appointment.

- By the time you arrive for the next lab session it will be assumed that you are totally comfortable with *all* the material presented in this first session. It will *not* be reviewed again. It is crucial that you keep up with the labs at this early stage!
- Study the notes for the *next* lab session in advance. You will get *much* more benefit from the session proper if you have already read the notes in advance. If you have time you could even try to complete some of the exercises in advance (though you *must* still attend all of your scheduled sessions!).

## 2.3 Session 2: File Systems, Text Editing and E-Conferencing

### 2.3.1 The bash Command Line Interface (shell)

When we interact with a computer—when we direct it to carry out some task for us—we use what is technically called its *user interface*. This is just jargon for the hardware (keyboard, mouse, monitor etc.) and software which is directly “visible” to us, and which mediates our instructions. User interfaces are also commonly referred to as *shells*—because, in some sense, they encapsulate and shield us from the inner workings of the machine, so that all we see and interact with is this outer “shell”.

User interfaces to general purpose computers fall into two broad categories:

- Graphical User Interfaces (GUIs)
- Command Line Interfaces (CLIs)

When using a GUI, one’s main interactions are via the mouse pointer. So when you use the mouse to activate menus, to move and resize windows and so forth, you are using a GUI.

In contrast, CLI interactions are textual: they are primarily via the keyboard, and corresponding display of textual material.

GUIs are a more recent invention than CLIs. For that reason, they are sometimes seen as more “modern”, “user-friendly”, and even “superior” to CLIs; but this is actually completely mistaken. CLIs and GUIs are not alternatives, but are *complementary* to each other. Depending on the particular kinds of interaction one is engaging in, either a CLI or GUI

may be more appropriate (or there may be little to choose between them).

So: in the course of this introductory module, we will be using both GUI and CLI type interfaces at different times—as appropriate to the particular activities that we are engaged in. You will be expected to develop a reasonable degree of familiarity and competence with both kinds of interface; and to be able to sensibly choose which to use, and to switch between them as appropriate.

In fact, you have already been exposed to both a GUI (the standard Windows 2000 interface, with its task bar, **start** button, applications icons, application windows etc.). In today’s session we are going to explore the use of a CLI. For this we are going to use a CLI to interact with our local computer. Note that a CLI can also be used to interact with a remote computer. This is explained in more detail in lectures.

The CLI we will use is called **bash** (the GNU “Bourne-Again SHell”). Note that whilst this is only one CLI, the same skills and techniques you learn using this CLI will transfer to other CLIs when working with a local machine or indeed a remote server.

So: having logged in to Windows 2000 as usual, start up a **bash** shell (via the **Start** menu). We will now introduce a few general purpose commands you can use in **bash**. These are concerned with examining and navigating around your computer’s *filesystem*.

### 2.3.2 Files and Filesystems

So what’s a filesystem?

The information stored on a computer is organised into units called *files*. Roughly speaking you can think of a file as corresponding to a conventional, paper, *document*. Some files are small (like a short business letter), some are very long (like a book). Files can contain different “kinds” of information (text, images, even sound and video; and, of course, *computer programs*).<sup>2</sup>

Since computers can store so *much* information, we need some systematic way of organising it—similar to a filing cabinet for paper documents. A *filesystem* is such an organised collection of files. In fact, filesystems typically employ a *hierarchical* organisation: a group of related files is located in a “container”, called a *folder* or *directory*. These directories can, in turn, be grouped into higher level directories. And so on.

<sup>2</sup>Strictly, it is not the information that is of various “kinds”—at bottom, all the information stored in the computer is just long strings of 1’s and 0’s; what is different is the way we choose to *process* the information.

At the top of this hierarchy, we will have complete storage “devices”. A single disk drive (the diskette drive, or the CD-ROM drive, or the internal “hard” disk drive) will normally count as a distinct device.

Now, in order to actually make any use of this hierarchical structuring of file systems, we will need ways to label or refer to the different items (devices, directories, individual files). Which is to say, we will have to assign them *names*.

Different computer environments adopt quite different conventions on how to name the various objects in a file system; but since we are largely working in the Windows 2000 environment, we will adopt its conventions.

Starting at the topmost level of the hierarchy, storage *devices* are given simple alphabetic labels: A, B, C etc.<sup>3</sup>

By contrast, directories and files can have fairly arbitrary textual names, such as **Research**, **Teaching**, **cv-13.txt** etc. These names can contain any alphabetic characters, any decimal digit characters, the dash (-), underscore (\_) and period or full-stop characters. In general they can be as long as you like, though it gets a bit cumbersome if you use more than about 10–15 characters.

Conventionally, files (as opposed to directories) include a suffix consisting of a period followed by 1–4 characters which indicates the kind of information in the file. So a filename ending in **.txt** might contain simple text; one ending in **.c** might contain C language source code, and so on.

A complete specification of a file might then look something like:

```
//h/mail/2005-2006/personal/birthday.mbox
```

This is called a file *path*; or, more strictly, for reasons we will see in a moment, it is called an *absolute* file path. The slash character (/) is used to separate or delimit the the distinct directories and subdirectories.<sup>4</sup> The double slash at the start marks the following **h** as the name of a device, as opposed to a directory.<sup>5</sup>

<sup>3</sup>This is clearly pretty restrictive, since it imposes an absolute upper limit of 26 distinct devices. It’s also very cumbersome, in that these single device letters can’t offer any mnemonic clues as to what physical devices they refer to. These limitations made some sense on the tiny computers they were first used on, about 30 years ago. Now they are a fossil reminder of a bygone age...

<sup>4</sup>This is true within **bash**; other application programs may use the *backslash* character (\) rather than the slash character as the separator in paths.

<sup>5</sup>Another common convention is to use a colon *after* the device letter, rather than a double slash before it. Thus, both **h:** and **//h** may be used to refer to device **h**. **bash** can

This particular path identifies a file on the disk drive labelled `h`; `h` has a directory called `mail`, within which there is a subdirectory called `2005-2006`, and within that there is a further subdirectory called `personal`. The file in question is in this final subdirectory and is called `birthday.mbox`. The extension `.mbox` suggests that this file is a mailbox file—i.e., it contains a collection of email messages.

Note that a disk device need not be divided into directories or subdirectories. A perfectly valid file path might be:

```
//h/grandma.txt
```

We would then say that the file `grandma.txt` is in the *root* directory of the `h` device. We use this terminology because the complete hierarchy of directories, subdirectories and files making up a filesystem on a particular device can be thought of as like a *tree* growing from a single root; this root is the initial directory, denoted by `//h/`.

We can see that specifying a single unique file with an absolute file path can be quite complex and cumbersome. Not surprisingly, some shortcuts have been invented to make things a little handier. The most common one is that, within the context of any application (such as an editor or CLI), there will be a notion of a *working directory*. That is, you can specify a directory that all file references should be taken relative to, until further notice. Then you will be able to concisely refer to files in that directory (or even subdirectories of it), without having to spell out the complete path every time. So if we had set the working directory to:

```
//h/mail/2005-2006/personal
```

We could then simply refer to the file:

```
birthday.mbox
```

without further ado. The latter sort of shortened reference to a file, which is to be interpreted relative to some current working directory, is called a *relative file path*.

### 2.3.3 Exploring Filesystems

Note that the filesystem(s) on your PC are subject to a variety of *access controls*. That is, there will be devices, directories, and individual files where you will have only restricted access (e.g., you might be able to examine the contents of a file, but not change

generally accept both forms, but will use the `//` form when it is typing out paths. Many other programs can only handle the colon form.

it, etc.) Don't worry: the system will automatically catch any accidental attempt to do something you are not allowed to do, and warn you of the problem.

Every student in the School is allocated an area of private, personal storage area, as mentioned in the first lab session. When you log in under Windows 2000, this personal storage, also called your *home directory*, is made accessible to you as device `H`. Under `bash` (and *only* under `bash`!) this same area can also be equivalently referred to by the paths `//h` or `/user`.

This storage is physically located on a central server computer (`ee_olympus`), and is accessed over the local area network (LAN). This has the great advantage that you will be able to access your personal files etc., from whichever computer (within the EE School!) you log into. Naturally, you will have full access rights to all files within your own home directory.<sup>6</sup>

Below are a few commands which you will use very frequently in `bash`. Because some of them will require specific access rights in order to operate, you may have to make sure you use them within your own home directory. Take about 10 minutes to experiment and try out these commands for yourself.

- **pwd : Print Working Directory:** this will show you your current working directory.
- **cd : Change Directory:** This is used to set your current working directory in `bash`. For example:

```
cd //c/temp
```

- **ls : LiSt Files:** This will show a list of the files and/or subdirectories in the current working directory (if any). You can use the slightly more complex form:

```
ls -l
```

to get a “long”, or more detailed listing. The `-l` after the command is an example of a command *switch*—because it “switches” the behaviour into something slightly different.

- **touch filename :** This command has a few subtle effects which we will not elaborate here. For now it is enough that if a file called *filename* (where this denotes some arbitrary name of your own devising) does not already

<sup>6</sup>Note that your storage area is subject to a *quota*—a maximum amount of space which you are allowed use. This quota is currently set at 5 MByte per student.

exist, then, after executing this command, such a file will be created, though with no contents (it is of zero length). So, for example:

```
touch testing.ext
```

should result in the creation of a file called `testing.ext`, with zero length, in the current working directory. (How would you check if this worked as advertised?)

The *filename* here is called an *argument* to the command, just meaning some additional piece of information provided as input to the command. This particular argument can be a simple *relative* file path as shown, and be interpreted relative to the current working directory, or it could equally be an absolute file path, such as the following:

```
touch /user/testing.txt
```

In general, any time you see a command that can accept a “filename” or “directory” argument, this can be specified either by a relative or absolute path.

- **rm filename : ReMove:** This simply removes or deletes the named file. Use this with caution: once you remove a file, it’s gone for ever! For example:

```
rm testing.ext
```

- **mkdir dirname : MaKe DiRectory:** This command is used to create a new directory. Once a directory is created, you can create files “inside” it. Try something like the following sequence of commands for example:

```
mkdir testdir
cd testdir
touch even-more-testing-file
ls -l
```

- **rmdir dirname : ReMove DiRectory:** And not very surprisingly, this command removes or deletes a directory. However, note that you can only remove a directory if it is empty—i.e., it does not contain any files or subdirectories (even empty ones!).

At this point you should be satisfied that you know how to use commands in `bash` to examine

what directories and files are in a filesystem, to create and remove directories, and to create and remove (empty) files.

This has been a very brief survey of commands for exploring and manipulating filesystems using `bash`. These commands were historically first devised on the *Unix* operating system. You can get some additional tutorial information on them by referring to Unix documentation—for example, the `UNIXhelp` information available on the School Web site at:

<http://www.eeng.dcu.ie/local-docs/UNIXhelp1.3/>

Not all of this information applies in exactly the same way under the NT operating system as under Unix, but it is still a very useful reference.

### 2.3.4 Working With Text Files

This session introduces the tools you can use to create and manipulate *text* files on the PC. Text files are just textual documents. You might compose a letter or email message and save it in a text file. Subsequently you might edit it to correct mistakes or add more information. You might copy it from one diskette or memory stick to another for backup purposes (in case one media gets damaged or lost). You might print it out or email it. You might delete it off your diskette or memory stick when you no longer have any use for it.

In the first lab session we saw how the DCU email system could be used for the basic creation and editing of text messages to be sent as email. The same text editing capability are provided by a large range of applications such as `pico` - under Linux of `Cygwin` or `Textpad` - under Windows 2000.

We will be dealing with very basic so-called **Latin-1** text documents. This is a way of expressing text in electronic (binary) form, specified by the *International Organization for Standardization (ISO)*. It is derived from an earlier standard called **ASCII** (American Standard Code for Information Interchange), but supports additional characters used in a variety of European languages (Irish, French, German and others).

**Latin-1** essentially specifies a way of encoding each supported character as a binary number with up to 8 binary digits.<sup>7</sup> The supported characters comprise upper and lower case alphabets (**a** to **z** and **A** to **Z**), a selection of accented alphabets (**á**,

<sup>7</sup>A single binary digit is a single 1 or 0 symbol, and is also called a *bit* for short. A group of 8 bits—as used to represent an ASCII character—is called a *byte*

ÿ etc.), the decimal digit characters 0 to 9, a variety of punctuation characters such as ; : . ? ( ) [ ] etc., and finally a collection of “special” characters such as + & \* % etc.

**Latin-1** provides roughly the same capability for representing text as used to be available with an old fashioned typewriter. **Latin-1** does *not* support more advanced or complicated kinds of text attributes such as italics, underlining, bolding, varying sized characters, etc. As such, **Latin-1** might be regarded as pretty “primitive”.

The great advantage of **Latin-1** is that it really *is* an open, internationally recognised, standard. **Latin-1** encoded text files can be read and printed on virtually any computer system, and can be successfully transmitted through any email system. **Latin-1** is a sort of “Lowest Common Denominator” of text file formats.

We will use **Latin-1** files in this module for preparing lab reports, and for preparing program files (in the **C** programming language). **C** programs will, in effect, *be* **Latin-1** files, and will be manipulated with the same tools as we manipulate any other **Latin-1** files.

### 2.3.5 Creating and Editing Text: Using PICO

**pico** is a simple *text editor* application: it will allow you to enter text, save it in a file, edit it again, save it again, and so on.

You start a session with **pico** simply by giving the command **pico** in **bash**. By default, any files we create with **pico** will go into the same working directory as we have set (with **cd**) in **bash**. So, let’s create a directory just for whatever experimental files we create in this lab session. We’ll create it immediately under our home directory, make it current, and then start **pico**. So the sequence of commands should be something like:

```
cd /user
mkdir lab2
cd lab2
pico &
```

The **pico** command optionally accepts a filename, to identify the name of a file we’re going to edit; but since we haven’t got any pre-existing files to edit yet, just ignore that for now.

Notice the *ampersand* character (&) at the end of the **pico** command. This tells **bash** that it doesn’t have to wait for the command to finish before re-issuing its prompt to accept new commands. That is to say, we want our **pico** session to run in parallel

or concurrently with our **bash** session. In that way we can switch between editing and executing other **bash** commands as we go along.<sup>8</sup>

Explore and experiment with the use of **pico** for a while. In particular, read through the online help. Now carry out the following exercise:

- Type up a short text with the following topic:

```
Why did I choose School of
Electronic Engineering and
Dublin City University
```

Your text should be no more than 3-4 short paragraphs in length; but it should be neatly formatted, with correct grammar, punctuation and spelling.

- *Save* the text in a file called:

```
my_opinion.txt
```

- Exit from **pico**.
- Start up **pico** afresh.
- Open the file **my\_opinion.txt** for editing again.<sup>9</sup>
- Make some modifications to the file.

Note carefully that, while you are editing the file within **pico**, this is only changing a “copy” of the file loaded into so-called “main” or RAM memory. This is a kind of semiconductor memory, which is completely *volatile*—the information will be lost if there is a power failure or the machine is otherwise reset. The copy of the file on disk is *not* being modified. To update the disk copy, you *must* save the file from within **pico**. It is very important to get into the habit of *saving early and often*.

- Exit from **pico** again.
- Check that your file is present as expected, using the **ls** command under **bash**.

At this point you should be satisfied that, using **pico**, you know how to create, save, and re-edit **Latin-1** text files.

<sup>8</sup>This is a very general notion, which is not specific to running **pico**: any time you give a command to **bash** you can tell it to run concurrently by putting an & at the end of the command.

<sup>9</sup>If you wish, you can do this by including the filename as an argument to the **pico** command as already mentioned.

### 2.3.6 Creating and Editing Text: Using Textpad

Wordpad is another simple *text editor* application that will allow you to enter and process text files. Try to open the same file written using Pico and add another paragraph about what do you expect from the EE102 module. Save the file. Compare Pico and Textpad and use that one that you find more user friendly.

### 2.3.7 Electronic Conferencing

Now, find out (using the **Web browser**) as much as you can about the *Email Conference* facility operated by the School of Electronic Engineering. Check out the archives (if any) associated with this conference.

Now: compose (on paper) a short message (one paragraph) which summarises your opinion of this module (*Software Engineering 1*) so far.

Feel free to be informal, and critical; but please be polite—do not use language in email you send that you would not like to read in email you receive! Now may be a good time to review *The Net: User Guidelines and Netiquette* at:

<http://www.fau.edu/netiquette/>

for some good advice on how to use the Net effectively and without accidentally causing offense.

Send this email message to the address:

[ee102-talk@maillist.eeng.dcu.ie](mailto:ee102-talk@maillist.eeng.dcu.ie)

Include your own email address in the `cc:` field of the message header. `cc:` stands for “Carbon Copy”. The effect is just that a *copy* of the message will get mailed back to you also. This is very handy as a positive confirmation that your message got into the mail system successfully. Of course it does not guarantee that it has been successfully delivered to the intended recipient.

Check the archives of the conference again, to check whether your message was received and archived correctly. Wait a couple of minutes for this to happen; but if it still hasn’t arrived by then, ask for help from a colleague or the demonstrator.

You should also receive a second copy of the message back to your own mailbox (this is in addition to the copy you explicitly sent with the `cc:` header). This reflects the fact that every message you send to `ee102-talk` automatically gets sent to every other student registered for this module—and since you, yourself are registered, that means you get a copy too.

If your message *is* now visible in the archives, and you have also received the two copies back into your own mailbox—congratulations: the message you have posted to the `ee102-talk` conference is now—immediately—visible to the entire world wide Internet community!

Hopefully, you are now getting some idea of how an email conference actually works—all the participants receive copies of all the messages sent by all the others, so everybody is party to all the comments.

The idea of this *particular* conference (`ee102-talk`) is that anybody in the class can use it to ask questions or make comments on any aspect of this module; and anybody else in the class (and, of course, the instructors) can offer answers or further comments or questions. In this way, an email conference allows *everybody* in the class to benefit from all questions and comments. Take a few moments now to study the document *Internet Email Conferences: A Short Primer* at:

<http://www.eeng.dcu.ie/~majordom/primer.html>

to get a better idea of how this can work.

You may, of course, retain the messages you receive from a particular conference—they may build up into a useful reference. But it’s not really *necessary* to do so—because, as you already know, all the messages are automatically retained in the central archives anyway! Indeed, the only reason for copying the messages to all the individual participants is so that you will be aware, on an ongoing basis, of the submission of *new* messages; but once you have scanned a message, you may as well delete it (to avoid filling up your mailbox file(s) and, as a result, your disk space). You can always look it up again in the archive if you ever want to!

Conversely, to really benefit from the conference, you need to check your email regularly (remember: at least three times each week); and you must also be willing both to ask questions and to offer answers where you can. Finally, please remember that the address:

[ee102-talk@maillist.eeng.dcu.ie](mailto:ee102-talk@maillist.eeng.dcu.ie)

is intended *only* for comments and discussion relating to the *Software Engineering 1* module. Please do *not* use it for circulating information on other subjects.

### 2.3.8 Homework

There is no itemised homework this time. However I would like to receive your opinion about the choice

for EENG and DCU. There will be no marking for it. Send it via e-mail at the following e-mail address:

ee102-reports@maillist.eeng.dcu.ie

The basic advice is the same as for the previous lab session: take time to review everything that was done, and make sure you are completely comfortable with it. Also, *review the notes for the next session in advance*.

You should also be continuing your programme of typing practice with the `Textpad`.

Finally: if there is anything that has been introduced in the labs, or the lectures, that you are confused about, or you have any general questions or concerns about this module, make a point of sending a message to the `ee102-talk` conference about it. Alternatively, as you receive messages other people have contributed to the conference, take the time to add a reply—even if it is only very brief. In this way you will get a feel for the dynamics and psychology of the conference, and be able to use it effectively to aid your own study later on.

## 2.4 Session 3: Introducing C

### 2.4.1 Preparation

From this session onward it is *essential* that you study the notes for the session carefully *in advance*. The lab exercises are getting increasingly extensive and demanding. You will *not* be able to study the notes for first time *and* carry out the required exercises within three hours.

### 2.4.2 Background

In this session you will attempt, for the first time, to develop an actual *program*. You will be using the `C` language. This is a so-called “high-level” language, so you will need to translate the program into the “low-level” or *machine language* that the Central Processing Unit (CPU) of your computer can actually directly execute. You translate the program using a tool called a *compiler*.

There are a wide variety of `C` compilers available for the Windows 2000 environment, each with their own particular strengths and weaknesses. We will be using DJGPP which is a version of the **GNU C Compiler** or `gcc` for short. This is very widely used both on Windows and Unix platforms. It has the further advantage that it is free!

`C`, and other languages closely related to it, are the *de facto* industry standard programming languages for most general purpose applications. `C` is

a very sophisticated language in certain ways. This makes it a powerful programming tool for the experienced software engineer. Unfortunately, it can also make it a very dangerous and difficult tool for beginners.

This session is also the first to require you to prepare a *formal* lab report. Before going any further, please read carefully through Chapter 3 which provides the *Laboratory Report Guidelines*. This is the same format that will be required in the final *assignment* for the module. Now is your opportunity to practise writing reports according to this required format. You will *not* receive individual grades or feedback on each lab report; but they *will* contribute to your overall assessment result for the module.<sup>10</sup> Selected reports from these lab sessions *will* be reviewed during lectures, to give an indication of what is good or bad.

You must write the report *as you go along* in the lab session.

Start now. Create a working directory called (called, say) `ee102-lab3` in your home directory. Make it current. Now use `pico` to create a file called `report.txt` in this directory. Insert into this the required header text for the report, as specified in the guidelines. Keep the `text editor` running throughout the lab session, so that you can add to it as you go along. Remember to save every time you add any significant amount of text—otherwise you run the risk of losing this text completely if, for example, there is a power failure in the lab!

At the end of the session you will be required to finish the report and submit it by email.

*Allow 20 minutes at the end of the session to do this.*

Specifically, even if you have not completed the assigned exercises, stop working on them at that stage and submit the report anyway.

### 2.4.3 Exercise 1: Hello World!

Here is the minimal “Hello World” program discussed in the lectures:

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

int main(void)
{
    printf("Hello World!\n");
}
```

---

<sup>10</sup>currently 4% of the overall module marks for each of the three formal lab reports, making 12% in total (although this may be subject to review – see the description of the assessment structure).

```
    return(EXIT_SUCCESS);
}
```

Use the text editor - Pico or Wordpad - to create a file called `hello.c` in your working directory (`/user/ee102-lab2`) containing the program listed above.

Compile your program by giving the command:

```
gcc -Wall -o hello hello.c
```

The meanings of the various switches and arguments are as follows:

- `-Wall` : This instructs the compiler to *warn* you (W) about *all* kinds of questionable constructions it finds in your program. When you are a novice C programmer it is sensible to always use `-Wall`, so that you get as much help as possible from the compiler when you encounter problems.
- `-o hello` : This instructs the compiler to place the compiled, executable, instructions, into a file called `hello`. The `o` here is mnemonic for *output*.
- `hello.c` : This simply names `hello.c` as the file to be compiled.

Of course, `hello.c` *should* compile without any problems—if you have entered it correctly. But if any problems are encountered try to figure out what is going wrong and fix them. If you get stuck, seek help from the demonstrator.

Check (using `ls -l`) what files now exist in your directory. There should be a new file called `hello`. This is the executable file—the file containing the binary instruction codes that the CPU can directly execute. Make a note of the size of this file in your report.

To execute the program you simply give its name as a command in `bash`:

```
hello
```

Check that the file executes as you expected. If it does not, try to figure out why, and fix it. Again, if you get stuck, ask for help from the demonstrator.

If the program executes correctly, then congratulations—you have just entered, compiled, and tested your first C program!

## 2.4.4 Exercise 2: Branching out!

Here is the program `insult.c` introduced in the lectures:

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

int main(void)
{
    char name[100];

    printf("Welcome to INSULT ...\n");
    printf("Please enter your name:\n");
    gets(name);

    if(strcmp(name,"John") == 0)
        printf("Hi smarty pants!\n");
    else
        printf("I don't like you!\n");

    printf("Bye from INSULT ...\n");

    return(EXIT_SUCCESS);
}
```

Using this as a basis, or template, develop (plan, code, test, correct) a program called `sport.c` which will behave as follows:

1. Display a suitable welcome/identification message.
2. Ask the user to type in his or her favourite sport.
3. If the topic is “Tennis”, display the message “Tennis? I play Soccer!”.
4. If the topic is anything else, display the message “What a silly game!” instead.
5. Finally display some suitable closing message.

Record your experiences in developing this program—especially whatever problems you encounter, and how you attempt to solve them—in your report.

## 2.4.5 Conclusion

That completes the major work of the lab session. But, as previously explained, the formal lab report, which you have prepared during the session, must now be submitted by email.

Carry out the following procedure to submit your report. Allow at least the last 20 minutes of the lab session for this—i.e., 20 minutes before the end of the session, stop working on program development, *no matter what stage you are at*, and just concentrate on submitting the report.

You will be awarded the full marks for this lab report, provided only that you submit *something*, of even roughly the correct format, *within your scheduled lab session*. Note that your (email) submission will be automatically timestamped. You will be awarded *zero* marks if you submit the report late (regardless of how much more complete it might then be).

- Read quickly back through the report, correcting any obvious errors, and possibly adding some brief concluding remarks.
- Exit from the `text editor`, saving the final version of the report.
- Start up the DCU email system as usual. Compose a new message and use of the following address for this new email:

ee102-reports@maillist.eeng.dcu.ie

*Note this address carefully.* It is *not* the same address as the main email conference for the module (`ee102-talk@maillist.eeng.dcu.ie`); it is a completely separate “pseudo” conference, whose only purpose is to receive (and publically archive by lab group) the formal lab reports. *Be extremely careful not to send reports to the main email conference.* The rest of the class will not appreciate having their mailboxes clogged up by such extensive messages!

As usual, it’s a good idea to also include yourself as an additional recipient (in the `Cc:` field) so that you get positive feedback that the message successfully went into the mail system.

- Fill in the subject of the message as simply **"Session 3 Report"**.
- Now *include* your report file as the body of the message. To do this you can attach your report to the email to be sent, or you can simply insert the contents of your report into the body of the email message using the “cut and paste” facility (ask a demonstrator if you need help on this point).

- Send the message.
- Assuming that you have included yourself as an additional recipient, then you should receive a copy back. Check for this.
- This submitted report should be automatically archived, as usual. Check whether this has happened. Note that only reports appearing in the archives, within the correct deadline, qualify for the award of marks—so, if your report isn’t there, you won’t get any marks! To check that your report has safely arrived in the online archive by pointing your browser at the one of the following addresses:

<http://list.eeng.dcu.ie/pipermail/ee102-reports/>

## 2.5 Session 4: Quiz Time

### 2.5.1 Introduction

This session is concerned with developing and reinforcing the basic tools of `C` programming already introduced. The session does *not* involve any significant new concepts, but merely requires that you apply the techniques you have already learned in slightly more elaborate ways.

You are again required to write a formal lab report, and submit it by email, in exactly the same format and manner as in session 3. This session also again involves creating, editing and storing files on your PC (in suitable directories), and compiling `C` source files to produce executable files. If you are unsure of any aspect of doing these things, please review the session 3 notes again. The relevant instructions will *not* be repeated here.

*Do this ahead of time: there will not be time for revision during this lab session!*

This session has two exercises. In the first you are give a program and told what it ought to do. You are then asked to test whether the program behaves as specified; and, if not, to correct any defects you can identify. In the second exercise you are asked to develop a more ambitious program, based on the given program. *This is the same overall format as your final module assignment will take.*

### 2.5.2 Exercise 1: Analysis

Consider the program `mquiz0.c`. This should prompt for the user’s name; then print “Hello” followed by the user’s name; then repeatedly ask the

user what the best album of all time is, unless and until the answer “The Queen is Dead” is given.

Test whether this program behaves as specified.<sup>11</sup> If not, identify and correct any defects. Carefully record each such defect, and your correction, in your formal report.

### 2.5.3 Exercise 2: Synthesis

The program `mquiz0` will keep on repeating the question unless and until the user is inspired to give the correct answer (or is of refined musical tastes already!).

Develop an enhanced version of this program, called `mquiz1`. This should behave in the same way as `mquiz0`, except that, if the user gives a wrong answer, the program should then *ask* whether the user wants to try again. If the user indicates that he or she does *not* want to continue then the program should exit (remarking “Good idea - ‘cos you don’t know much about music, do you?”).

## 2.6 Session 5: Ever Decreasing Circles...

### 2.6.1 Introduction

This session is designed to explore more fully the **C** *data types* as well as providing a gentle introduction to the **C** *Standard Library*. You are again required to write a formal lab report in this session, and submit it by email, in exactly the same format and manner as in sessions 3 and 4. This session also again involves creating, editing and storing files on your PC (in suitable directories), and compiling **C** source files to produce executable files. If you are unsure of any aspect of doing these things, please review the session 3 notes again. The relevant instructions will *not* be repeated here.

The overall format of this session is very similar to session 4. It has two exercises. In the first you are given a program and told what it ought to do. You are then asked to test whether the program behaves as specified; and, if not, to correct any defects you can identify. In the second exercise you are asked to develop a more ambitious program, based on the given program.

<sup>11</sup>Note that, once you are viewing the program file in **Web browser**, you can use the `Save as...` option on the **File** menu of **Internet Explorer** or **Netscape Navigator** to save a copy of the program in some suitably named file in your home directory.

### 2.6.2 C Data Types and Operators

A *data type* is a format for storing and manipulating some particular “kind” of information in a **C** program. So far, we have used just two data types: `char` and `int`. In fact, we used an *array* of `chars` in order to represent *strings* of **Latin-1** characters. The length of a string is arbitrary, up a maximum length defined when the string is *declared*. Up to this point, we have used the `int` data type to store only two values – `TRUE` and `FALSE`, which correspond to 1 and 0 respectively. As we saw, using an integer variable to store `TRUE` and `FALSE` in conjunction with an `if` statement, is useful for controlling the flow of a program. This also applies to `while` and `for` statements.

In this session, we explore two data types used for representing, storing, and manipulating *numbers*. The `int` data type can be used to represent positive and negative whole numbers. The `float` data type, on the other hand, can represent fractional, or rational, numbers (positive or negative), using a form of scientific notation.

#### The Assignment Operator

The most basic operation or manipulation that can be performed with these data types is to assign a value to a variable of that type. For this, we use the *assignment operator* denoted by a single equal sign “`=`”. The assignment operator takes two operands. The left hand operand *must* be a variable. The value of the right hand operand is copied into, or assigned to, the variable named as the left hand operand.<sup>12</sup>

An example of the use of the assignment operator might be:

```
int the_answer;
float pi;

the_answer = 42;
pi = 3.141;
```

Note, incidentally, that the assignment operator *cannot* be used with strings. Thus, it would quite wrong to say something like:

```
char s[100];

s = "Hello world!";
```

<sup>12</sup>The assignment operator also technically generates an “evaluation result”, just as any of the other operators do. This evaluation result will be the value of the right hand operand—i.e., the same value as is stored in the variable named as the left hand operand. This allows an assignment operation to be nested within a more complex expression. That is *sometimes* useful, but its use is beyond the scope of the current session.

If you need to do something like that, use the function `strcpy()` from the **C** Standard Library (see next section) for this:

```
char s[100];

strcpy(s, "Hello world!");
```

## Numerical Operators

Both these numerical data types allow the four normal arithmetic operations:

+	Addition.
-	Subtraction.
*	Multiplication.
/	Division.

These operators all accept two operands, and evaluate with a result which is of the same data type as the operands. Note carefully that when you divide two values of type `int` the result is just the *quotient* and will *still* be of type `int`. *Any remainder will be discarded*. Thus, for example `5/3` would evaluate as `1`. You can use the special *remainder* operator, denoted by `%`, to separately calculate the remainder part of an `int` type division if you like. So `5%3` evaluates as the remainder after dividing 5 by 3, which is to say 2. The remainder operator is not applicable to `float` types (for obvious reasons?).

## Comparison and Boolean Operators

These numerical data types also allow various comparison operations:

<code>==</code>	Test for (exact) equality.
<code>&gt;</code>	Strictly greater than.
<code>&gt;=</code>	Greater than or equal to.
<code>&lt;</code>	Strictly less than.
<code>&lt;=</code>	Less than or equal to.

All of these comparison operators yield a result of either 1, indicating `TRUE`, or 0, indicating `FALSE`.

In fact, **C** provides a special suite of operators for manipulating `TRUE` and `FALSE` values:

<code>&amp;&amp;</code>	Boolean AND.
<code>  </code>	Boolean OR.
<code>!</code>	Boolean NOT.
<code>==</code>	Test for equality.

In more detail, the meaning of these operators is as follows:

- *Boolean AND operator*: Denoted by a double ampersand `&&` (with no space between them). Takes two operands. If both operands are `TRUE` it evaluates as `TRUE`; otherwise it evaluates as `FALSE`.
- *Boolean OR operator*: Denoted by a double vertical bar `||` (with no space between them). Takes two operands. If either operand is `TRUE` it evaluates as `TRUE`; otherwise it evaluates as `FALSE`.
- *Boolean NOT operator*: Denoted by an exclamation mark `!`. Takes one operand. If the operand is `TRUE` it evaluates as `FALSE`; if the operand is `FALSE` it evaluates as `TRUE`.
- *Test for equality operator*: Denoted by a double equal sign `==` (with no space between them). Takes two operands. If both operands have the same boolean value it evaluates as `TRUE`; otherwise it evaluates as `FALSE`. As explained previously, this operator can be applied to operands other than `TRUE` or `FALSE`, but its result is always either `TRUE` or `FALSE`.

## 2.6.3 Introducing the C Standard Library

Although it might not be immediately obvious, the programs we have developed so far make use of “pre-packed” software that is supplied with every **C** compiler. Collectively, this software is called the *C Standard Library* and it consists of *functions* for programming tasks we will want to perform very often. We access these functions in our programs by *including library header files* using the `#include` preprocessor directive. Header files are used to group functions which perform similar or related tasks. For example, the header file `stdio.h` provides access to a whole range of functions to do with inputting and outputting data from programs. Similarly, the header file `string.h` provides access to a range of functions dealing with strings (i.e. arrays of characters).

An example of a programming task we want to perform very often is getting text data into our programs for subsequent processing. So far, we have used the `gets()` function supplied in the standard input/output library and accessed via the `stdio.h` header file:

`gets(s)`: reads the next input line of text into character array `s` and replaces the terminating new line character with the `NULL` character.

Another example of a common programming task is outputting text data from our programs. For this, we have used the `printf()` function, also accessed via the `stdio.h` header file:

```
printf(char *format, ...): prints
formatted output on the output device.
The function takes a variable number
of arguments. format is a required argu-
ment which contains the text to be
printed as well as any required conver-
sion specifications. The remaining argu-
ments specify the data to be converted
to textual output for display.
```

The only *manipulation* of data we have performed in our programs so far is to compare two strings against each other. For this purpose, we have made use of the `strcmp()` function from the C Standard Library, accessed via the `string.h` standard header file:

- `strcmp(char *s, char *t)` : Allows two strings `s` and `t` to be compared. Generates a return value of 0 if the two strings are *exactly* identical, and 1 otherwise.

As you might guess, the C Standard Library contains many more functions than the small number we have encountered. Your primary task in learning to program in C will be to familiarise yourself as much as possible with the functions available. This should be one of your objectives for the remainder of this module (and indeed for the follow-on module – EE105: Software Engineering 2). For the purposes of this exercise we will simply explore more fully the use of two functions from the large number available to us: `printf()` and `scanf()`.

## 2.6.4 Exercise 1: Analysis

Consider the program `circle0.c` introduced in lectures. This should prompt for the user’s name; then print a welcome message containing the user’s name; then ask the user to enter the radius of a circle (in *m*); the program should then calculate and print out the circumference and area of the circle.<sup>13</sup>

The program is very poorly laid out and formatted. Reformat it in a way that makes it easier to read and understand.<sup>14</sup> Correct any “obvious” mistakes. Of course, you should note what changes your

<sup>13</sup>The circumference is given by the mathematical formula  $2\pi r$ ; the area by the formula  $\pi r^2$ .

<sup>14</sup>Note again that, once you are viewing the program file in **Navigator**, you can use the **Save as...** option on the **File** menu of **Navigator** to save a copy of the program in some suitably named file on your PC.

make, and include a copy of the reformatted program, in your formal report.

Now test whether the program behaves as specified. If not, identify and correct any defects. Carefully record each such defect, and your correction, in your formal report.

## 2.6.5 Exercise 2: Synthesis

The program `circle0.c` calculates the circumference and area of just one circle.

Develop an enhanced version of this program, called `circle1.c`. This should behave in the same way as `circle0.c`, except that it should repeat the sequence (of requesting a radius, then calculating and printing out the circumference and area) exactly *five* times. Each time the prompt should change (so the first time it should say “Enter radius 1:”, the second time “Enter radius 2:” and so forth).

## Chapter 3

# Laboratory Report Guidelines

It is an explicit objective of this module to develop your written communication skills. Therefore, in assessing the module, we shall 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 document carefully, and practice applying the guidelines given here in every relevant lab session. These guidelines also apply to the report on the final *assignment* (for which the bulk of the marks for the module are awarded).

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

`ee102-talk@maillist.eeng.dcu.ie`

### 3.1 General

The *Software Engineering 1* Lab and assignment 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 module also.

The central principle of writing the reports for *Software Engineering 1* is that they should explain *where your time went*.

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 ex-

plain, 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.

### 3.2 Typing Up

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

It is recommended that you use `pico` or `Textpad` to prepare the programs. Do *not* use any word-processing package such as `Microsoft Word` as they change the format the data is saved into from text into a particular other format. However when writing your report you can use any word-processing package at your choice.

It is recommended that you create the report file at the very start of the Lab session. Then keep a `text 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.

**Ensure that you save the report to disk every time you make any substantial change or addition to it. Otherwise, if there is a power failure, or the PC crashes, or even if you accidentally exit the editor, you will lose all your notes up to that point!**

## 3.3 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 purely sequential manner, in exactly the order you tackle the exercises during the session.

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

### 3.3.1 Heading

Each 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 associated with the lab exercise.

### 3.3.2 Plan

Given the statement of a particular exercise to be done, 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*).

The notes you are given 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 at most one 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 notes. Such restatement is a waste of time, and may well result in marks being deducted!**

### 3.3.3 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, in order to check that the output (or behaviour) conforms to what you expected.

**Test** involves the following kind of activity:

- Testing: This is providing suitable test data, to assess whether, or to what extent, the 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 goes straight through phases of coding, compilation, execution and test in one single pass. 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* and *Test* sections 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 ‘Unrecognized 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 to happen and what actually happened. Distinguish between problems that are *compile-time*—i.e., they are indicated by some message from the compiler—and those that are *run-time*—i.e., they are manifested only when the program is executed.

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* and *Test* sections is that there should be enough information so that another person *could* reconstruct what you did more or less exactly.

### 3.3.4 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 *Test* 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. Marks here go for clarity, and honesty, not actual achievement!

## 3.4 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* and/or *Test* 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 prints
a birthday greeting.
```

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

#include <stdio.h>

int main(void)
{
    printf("Happy Birthday!!!");
}
-----
```

Note that, using **pico** or **Textpad** to edit the report, you can easily use the **Copy** and **Paste** facilities to copy from one location to another.

Where you have created a new source file from scratch 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 you progress through an exercise (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.

### 3.5 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!

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.

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 of problems for which we can give more detailed advice.

- **Problems of Understanding:** For example, “I don’t understand what a *library* is”, “I don’t know what a *floating point type* is”, “I don’t understand what *while* 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 online notes, looking for information in the online documentation, 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 problems. 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.

### 3.6 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., “. . . and then the bl\*#@#dy computer crashed again!!!”).

Take careful note of the following more specific points, which represent the most frequent presentational 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 (:) and full stop.
- Think of a left bracket as part of the following word; think of a right bracket as part of the preceding 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 Latin-1 file is to surround it with asterisks, like this:

This is how to show *\*emphasis\**.

## Chapter 4

# Copyright

This Hypermedia Document is copyright-ed,  
© 1994–1998, by Barry McMullin  
© 1998–2003, by Noel E. O’Connor  
© 2003–2006, by Gabriel-Miro 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 author.

# Appendix A

## Program Listings

### A.1 mquiz0.c

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

#define TRUE 1
#define FALSE 0

int main(void)
{
    char album[100];
    char name[100];
    int asking;

    printf("Welcome to mquiz0!\n");
    printf("What is your name?\n");
    gets(name);
    printf("Hello %s", name);

    asking = TRUE;
    while (asking)
    {
        printf("Here is your question:\n");
        printf("What was the best album of all time?\n");
        gets(albums);
        if (strcmp(album, "The Queen is Dead") == 0)
            printf("Yes - you are obviously an expert!\n");
            ask = FALSE;
        else
            printf("No - don't you know *anything*?\n");
    }
    return(EXIT_SUCCESS);
}
```

## A.2 circle0.c

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

int main(void)
{
    float radius, circ, area;
    char name[100];

    printf("What is your name? "); gets(name);
    printf("Hello %s and welcome to the circle program!\n",name);
    printf("Please enter a radius: ");
        scanf("%f",&radius);

    circ == 2 * radius * 3.14159;
    area = radius * 3.14159;

    printf("The circumference is: %f meter(s)\n",circ);
    printf("The area is: %f sq. meter(s)\n");

    return(EXIT SUCCESS);}

```

# Appendix B

## Skeleton Lab Report

---

EE102: SOFTWARE ENGINEERING 1

ACADEMIC YEAR 2005/2006

LABORATORY REPORT

---

Student Name: A. McStudent  
ID Number: 55123456

Session Date: 17th October 2005  
Session Number: 3  
Session Title: Introducing C

---

Exercise 1: Hello World!

---

Plan

---

Development

---

Test

---

Conclusion

---

---

Exercise 2: Branching Out.

---

Plan

---

Development

---

Test

---

Conclusion

---