Tricks of the Trade for Novice Programmers

Vreda Pieterse
# Contents

Preface iii

About the cover iv

Foreword v

## 1 Netiquette 1

1.1 Introduction ................................................. 1
1.2 General rules ........................................... 2
1.3 E-mail ................................................ 3
1.4 Online forum ........................................ 6
1.5 Style and formatting ..................................... 9
1.6 A final note .............................................. 11

## 2 Coding standards 13

2.1 Introduction ............................................. 13
2.2 Style .................................................. 14
2.3 Clarity ................................................ 16
2.4 Flexibility ............................................. 19
2.5 Reliability ............................................. 20
2.6 Efficiency ............................................. 22
2.7 A final note ........................................... 22

## 3 Academic integrity 25

3.1 Introduction ............................................. 25
3.2 Practical assignment goals ............................ 26
3.3 Working with existing code ........................... 26
3.4 Plagiarism ............................................. 29
3.5 Regulations ........................................... 30
3.6 Working together or assisting a classmate .......... 34
3.7 A final note ........................................... 34
## Contents

### 4 Installing Linux
- 4.1 Introduction .............................................. 37
- 4.2 Linux distributions ....................................... 38
- 4.3 Install options ........................................... 39
- 4.4 Installing the OS ......................................... 42
- 4.5 Installing software ........................................ 43
- 4.6 A final note ................................................ 45

### 5 Operating system utilities
- 5.1 Introduction .............................................. 47
- 5.2 Directories and navigation ................................. 48
- 5.3 Files and their contents ................................... 50
- 5.4 Wild card characters ...................................... 54
- 5.5 Serialising and compression .............................. 55
- 5.6 Become efficient ........................................... 58
- 5.7 A final note ................................................ 60

### 6 Compiler utilities
- 6.1 Introduction .............................................. 63
- 6.2 Command line compiling and linking .................... 64
- 6.3 Automating the build process ............................ 70
- 6.4 Reducing the size of a makefile .......................... 78
- 6.5 A final note ................................................ 81

### 7 Software testing
- 7.1 Introduction .............................................. 85
- 7.2 Black box testing ........................................... 87
- 7.3 White box testing .......................................... 90
- 7.4 Debugging tools ........................................... 95
- 7.5 A final note ................................................ 96

### 8 Conquering technology
- 8.1 Introduction .............................................. 99
- 8.2 Active participation ........................................ 101
- 8.3 Read more .................................................. 102
- 8.4 Write code .................................................. 104
- 8.5 Productive learning ....................................... 105
- 8.6 A final note ................................................ 107

### Bibliography
- Bibliography .................................................. 109

### Index
- Index ............................................................ 115
Preface

People in today’s world have to use technology in practically every job. For this reason, many degree programmes recommend that an introductory programming module should be part of the curriculum. This does not imply that everyone should become programmers. The idea is to educate students to have smarter relationships with technology, and computers.

You have to achieve a reasonable level of programming confidence so that you can be comfortable with technology. You should also acquire substantial detailed knowledge of the programming language you use and the technicalities relating to how computers work. Some studies indicate that it takes about ten years to transform a programming novice into an expert [49]. Ericsson et al. [8] support this view; they claim that approximately ten thousand hours of deliberate practice is required to become an expert in an area. Successful programmers have to engage in specific activities which can improve aspects of their programming performance. It is of prime importance to write as many programs as possible and then put the programs you write to the test. Do not unquestioningly accept what you are taught. Experiment with the examples by making small changes and see if those changes have the effect you expected. Students who succeed in the end are those who are able to deal with shortcomings in the notes, the lectures and the administration of a module and who take responsibility for their own learning.

The C++ programming language is widely used. For this reason it is the programming language of choice for many introductory programming courses. Such courses require you to write, compile and run programs written in this language. To this end, you need certain skills and knowledge. This book contains additional notes and guidelines which we have developed over the years to assist the students in our first-year introductory course in programming. Though the focus is on the scaffolding students need in the course, the advice has a far wider scope.
Each chapter concludes with a final note, divided into three subsections. The first of these subsections summarises common errors. It is intended to emphasise the pitfalls that students should avoid. The next subsection is called the survival kit. It is the essence of the chapter and can be used as a quick reference. The concluding subsection gives general guidelines to help students to conquer the topic of the chapter.

Special thanks go to the many students who have attended the introductory programming module presented by the Department of Computer Science at the University of Pretoria over the years. They have shaped my thoughts about how they can be helped to master the art of programming. I have been fortunate in having the help of many colleagues who have worked with me while presenting this module. Derrick Kourie was instrumental in defining the teaching style and course structure, and the emphasis on value-based teaching. Linda Marshall, Isabel van Rooyen, Hugo Breedt, Christoph Stallman, Hannes Janse van Vuuren and Daniël Louw have collaborated on research into the presentation of the module that helped improve the quality of our teaching, and contributed to the notes we developed to support our teaching. Christoph Enslin, Justin Kourie, Isabel van Rooyen, Christopher Cleghorn, Christoph Stallmann, Anthony Burgess and Madoda Nxumalo have written instructions and notes that are incorporated into this book. Neels van Rooyen has been a tremendous help in ensuring that the material is technically correct.

I could not have completed this book without the moral support of my husband, Ben Pieterse, and the encouragement of my head of department, Andries Engelbrecht. The final version of this book benefits from the skills of Gillian de Jager, who devoted many hours to the language editing and continues to teach me to write better English. Derrick Kourie was deeply involved in creating this book, giving critique and inspiring confidence along the way, and writing the foreword to complete this book.

Vreda Pieterse
January 2016

About the cover

The cover designed by Marie Pieterse symbolises the function of this book. The book provides scaffolding for novice programmers. The scaffolding helps them to understand the environment in which they learn how to program. The scaffolding in this book can be compared to the working platforms of construction scaffolds from which they can, conveniently and at an ideal angle, reach the learning contents.
Foreword

The first few weeks at university are inevitably daunting to new students. You are in a completely unfamiliar place. You may know a few people, or perhaps no one, in the throngs of people milling around. You have to learn about all sorts of unknowns: the layout of the campus; the names of the buildings; the names of new subjects and lecturers; the times and venues of strangely named events such as “practicals” and “tutorials”; etc.

One of the scariest unknowns you face is whether you have the ability to cope with what is to come. Almost everyone will be filled with self-doubt. “Will I make it?”, you ask yourself. “Do I have what it takes to get this degree? Is this the right degree for me to take?” As you look around, you see friends chatting, couples holding hands, people who seem to know exactly what is going on, while you are battling to keep yourself together. Everything is new and you feel frightened, lonely and inadequate.

These first few weeks can be critical. The choices you make and the habits you form can quite literally determine the success or failure of the rest of your university life and, therefore, the entire course of your life after university. This Foreword to the first volume of the Tricks of the Trade series is intended to orient you mentally in a direction which will give you the best chance of success at university, both academically and socially. Indeed, if you fully internalise the core ideas mentioned here, you will fundamentally change the way you think about yourself, about the world and about other people.

I’m OK – you’re OK

As an aid to help you reflect on your basic orientation to the world around you, consider what is called the OK Corral in Figure 0.1. This diagram is derived from a best-selling self-help book, I’m OK, you’re OK1 by Thomas Anthony Harris, published in 1969.

1https://en.wikipedia.org/wiki/I%27m_OK,_You%27re_OK
The figure indicates four basic life positions that characterise our relationship to others, namely:

1. **I'm OK–you're OK**: this is a healthy win-win life position. I get along with the people around me and I am generally happy. I feel neither superior nor inferior to others. Because I am comfortable with people, I am comfortable about assertively putting forward my opinions, but I also listen with an open mind to the opinions of others.

2. **I'm OK–you're not OK**: this is a one-up I-win-you-lose life position. I want to get rid of the people around me and my general tone is one of anger. I feel superior to those around me, I criticise their behaviour and I am aggressive about putting forward my opinions. In extreme form, this is the life position of the psychopath.

3. **I'm not OK–you're OK**: this is a one-down I-lose-you-win life position. I want to get away from the people around me because, for the most part, I feel helpless and inadequate. Because of my low self-esteem and feelings of inferiority, I am passive about expressing opinions. I prefer to keep a low profile and remain silent. In an extreme form, this is the life position of the paranoid.

4. **I'm not OK–you're not OK**: this is a hopeless I-lose-you-lose life position. It is my expectation that I will get nowhere with my life. I therefore see no point in acting or being enthusiastic, because I have lost hope that things can change. I feel too confused to express any opinions. In an extreme form, this is the life position of the suicide.

![Figure 0.1: The OK Corral](image)

Where do you think you generally belong? Though each of us has a favoured life position, no one is locked into only one position. Instead, we tend to go in various directions as our circumstances change.
The pressures and anxiety that students feel during the unfamiliar and challenging first few weeks at university, will incline most people to the “I’m not OK–you’re OK” life position. The highest achievers, a small minority, may escape much of the anxiety. Some might be not only quietly confident of success but also comfortable with others, so they belong in life position 1. Other high achievers may have an arrogant disdain for others; they belong in life position 2. A small number of students might be in life position 4. Typically such people do not really want to be at university but have come to study because of social pressure or to please their parents.

Clearly, the ideal is to be in the “I’m OK–you’re OK” life position. The next section presents some key ideas for shifting your life position towards this ideal.

Responsibility

Self-awareness is pivotal if you want to change or improve your situation in a rational way. Once you have admitted the reality of your situation to yourself, it is much easier to make and control the changes you desire. For this reason, it is a good idea to know which position in Figure 0.1 you tend to occupy.

Take care, though, because your self-awareness risks being superficial if you do not analyse yourself deeply. To be able to give a rational and powerful response to your situation – i.e. to be response-able – means to be fully and whole-heartedly responsible for who and what you are. You have to take responsibility for yourself in every possible way: for your words, your actions and how you deal with your feelings. When you do this, you empower yourself to respond to the circumstances of your life and to develop positive relationships with other people. When you do not do this, you give up the power available to you to take control of your life and you jeopardise your power to form meaningful, whole, honest and integrated relationships with other people.

Taking responsibility for yourself is the key to living a whole, integrated, honest and powerful human life.

You may ask how you can do this? How do you take responsibility for yourself? There is no magic formula. It is a “head thing”, something you cultivate in your mind. You do it by realising that the alternative is to be phoney and dishonest. So you decide to align yourself with the reality that you are solely responsible for yourself.
Monitoring the language you use is a good way to determine the extent to which you take responsibility for your life. Words and phrases such as “I have to”, “I have no choice”, “I can’t”, “I can’t help it”, “that’s just how I am”, “it’s just that . . .” — these are all excuses or cop-outs! They push away responsibility and are almost always lies. In most circumstances, you do not “have to”, you “choose to”. You do in fact have a choice. When you start to say “I choose to” or “I decide not to” then you begin to speak with integrity, and you begin to take responsibility for your choices, your actions and your life.

So you choose to smoke or not to smoke. You choose to experiment with drugs or do not choose this. If you become addicted, you choose to stay an addict, instead of giving up the drugs or alcohol. You choose how you express your sexuality. You choose to overeat or to control your eating. You choose to keep to the speed limit when driving or do not choose to do so. You choose to live with and honour your parents or you do not. You choose to waste hours on FaceBook / WhatsApp / YouTube / electronic games or you do not. In all these matters, you are the one who decides. It may be difficult — perhaps even extremely difficult — to make sensible and appropriate choices, but it is not impossible. Every choice you make has consequences, and when you make choices, you choose the consequences as well.

Of course, there are practical limits to what you can choose. You cannot choose your skin colour, your gender, your sexual preferences, your country of origin, your level of intelligence, congenital health conditions, etc. But we do have a range of choices which is generally much larger than we care to admit.

Recognise how radically free you are to make choices. Decide to take responsibility for your choices. This will be the most liberating gift you can give yourself. It will also be the key to your academic success.

Here are a few facts to consider about your life. They relate to your status as a first-year student, but they apply equally as much to all other circumstances in your life.

- You are at university doing this particular degree because you have chosen to do this. You have not been forced to be at university, even though you might have many social pressures, for example from your parents. Such pressure is not irresistible. You can say “no”, in which case there will be certain consequences and these may be unpleasant. So you have chosen to be here, perhaps reluctantly, perhaps anxiously, but you and you alone have the final say.
• It is your choice to attend classes / practicals / tutorials regularly or not. It is your choice to complete or to skip your assignments. It is your choice to revise each night the material covered in class during the day. If you choose instead to spend time hanging out with friends or drinking in a pub or playing computer games, there will inevitably be negative academic consequences. Take personal responsibility for choosing those consequences. Do not whine about the difficult subject or the boring lecturer.

• It is your choice to follow a healthy lifestyle so that you can cope optimally with the demands of academic life. It is your choice to ensure that you have enough sleep, that you eat appropriate food and that you exercise appropriately. It is your choice to abstain from smoking, excessive alcohol and experimentation with drugs. Alternatively, you choose the consequences of an undisciplined lifestyle: tiredness, poor health, lack of concentration, potential addiction, etc.

• You have choices about the friendships you make and the company you keep. These choices will inevitably have positive or negative academic consequences. You probably do not have much of a choice about whether or not you fall in love. One’s hormones are powerful and can play havoc with one’s life. But you can choose to be the master, not the victim, of your sexuality. Ultimately, you are the one who chooses how constructive the friendships you form will be, and how all-consuming and unrealistic you will allow romantic relationships to become.

Although we have far more choice than we care to acknowledge about our actions and decisions, we do not have much of a choice about the emotions we feel. We all feel these emotions, much as we feel the temperature. Just as you cannot choose to feel hot or cold, you cannot choose to feel happy or sad or scared or angry.

What you can do is choose how to respond to these feelings. You can also choose to take action that will increase the chances of having good or bad feelings in future. You can talk yourself into feeling downhearted and inadequate. You can make stupid choices which are likely to lead to bad feelings; or you can make intelligent choices which are likely to result in good feelings. The next section focuses on the fact that when we do not take responsibility for our actions, we end up playing psychological “games” with one another, and inevitably end up with bad and negative feelings.
Drama triangle: the basis of games

Most good drama, whether in literature or films or plays, is about the interaction among three different roles: the victim, the persecutor and the rescuer (see Figure 0.2). From our childhood on, we delight in stories where these roles are played out: the wicked witch (persecutor) who fools the beautiful princess (victim) into eating the poisoned apple but who is rescued just in time by the dashing prince (rescuer). We do not have to belabour the point. You can easily identify these roles in your favourite films and even in your favourite computer games.

![Drama Triangle Diagram]

Figure 0.2: The Karpman Drama Triangle

When you do not take responsibility for your actions and decisions, you end up positioning yourself as a victim in a pseudo-drama. You imagine that you are being persecuted by something or someone, and you wait for someone or something to rescue you from your plight (usually in vain). In fact, it is even worse than that. You may find you are swapping roles in many relationships, so that though you may start off in the role of victim, you end up being the persecutor, perhaps later in the rescuer role. Here are a few fictitious examples of how this might negatively play out in the context of a university career.

- Though Thabo is a bright young man, his disadvantaged background has always weighed heavily on him and he sees himself as a victim of society (the persecutor). As a result, he does not have high expectations of himself. Instead, he expects to be rescued from his plight by the government (the rescuer). Though his final marks at school are not impressive, he gains admission to university. Instead of aiming for top marks—something that is well within his abilities—his victim-mentality holds him back from even trying. Instead, he lazes around, makes bad friends and blames the system (he effectively becomes the persecutor of the system) for failing his exams.
• Sarah is a high achiever who had eight distinctions in matric. While at school, her parents constantly pushed her to achieve top marks. As a result, she did not have the leisure time to be simply a normal child and have fun. As she began to reach maturity, she became increasingly resentful of all the pressure on her. She came to see herself as a victim of her persecuting parents. And so at university, she started making irresponsible choices, cheating in her exams, drinking irresponsibly and failing her tests. Her and her parent’s roles had changed. She had become the persecutor and her parents, the victims. She allows herself to be “rescued” from her misery and confusion by a hard-drinking boyfriend, falls pregnant and drops out of university.

• Selo is an intelligent happy chap who just loves playing computer games. In fact, he is pretty much a game addict. Because his parents cannot scold him at university, he chooses to miss class, skip assignments and indulge himself by playing his favourite computer game for long stretches of time. His academic performance soon suffers and he starts to fail exams for the first time in his life. He persuades himself that he is a victim. Life is unfair. Lecturers expect too much of students. The amount of work assigned is just too much for normal human beings. The university and its staff members become the persecutors. In his mind, his life is so tough that he believes he needs relief from all the pressure. “I need to play computer games to cope with life,” he says. “I can’t help it. I have no choice but to be a computer game addict.”

• Zanzi is a pleasant young lady who likes to dress fashionably. Even though she did reasonably well at school, she believes that this was a matter of luck rather than ability. Because of the negative messages about women in her family background, she sees herself as being a victim of her gender. She regards universities as mainly male-dominated places. Women are not supposed to do well. Consequently, she feels justified in skipping classes and assignments for the slightest excuse. When her semester marks are critically low, she pretties herself up and visits the course’s male lecturer. She hopes to use her feminine charms to persuade him to rescue her somehow. When he refuses to bend the rules for her, she sees him as a persecutor.

In all these fictitious but typical cases, the root cause of the problem is that these people have defined themselves as victims. The solution for Thabo, Sarah, Selo, Zanzi and others like them, is to take personal

---

2This is not an entirely fictitious story.
responsibility for their actions—to know that they are not victims. That
does not necessarily mean that they will always make the best choices or
that they will necessarily be successful in their university careers. It does
mean that they will be integrated, honest, empowered people with far
more energy and the capacity to make good choices, to enter into whole
relationships, and to succeed in their lives, academically and otherwise.

Software development values

The goal of any university course is to influence decisively your skills
and your values. The various chapters of this book will introduce you to
important and useful skills needed in introductory software development
courses. These skills are the *Tricks of the Trade*—the *trade* being software
development. However, when professional software developers use these
tricks, they do so inside a certain professional cultural milieu.

Responsibility, integrity and enthusiasm are highly valued at the heart
of this cultural milieu. The meaning of responsibility has already been
discussed above. Responsibility naturally leads to integrity and enthusiasm.
Here are some of the ways to check whether you measure up to this value
system as you engage in a software development course. Your basic life
position of I’m OK – you’re OK means that you have a non-victim mentality,
leading you to –

- take responsibility for your own work;
- strive for excellence;
- manage your time well;
- keep up to date with your revision;
- hand in your assignments on time;
- diligently attend lectures and practical classes;
- collaborate with fellow students when instructed to do so;
- follow the instructions for processes and procedures;
- request assistance and give feedback when appropriate; and
- treat fellow students and staff members with dignity and respect.

If you *choose* to behave in this way, you will do well in your university
courses. Your success will help you enjoy your university career. But if,
through lack of motivation and self-discipline, you choose to behave in the
opposite way, then you are likely to fail. In fact, you will then be unlikely to
be happy and successful as a software developer and you should consider a different career.

Here are some of the signs of victim-like behaviour which is irresponsible, without integrity and lacks enthusiasm. Victims who are not OK choose to be people who –

- continuously rely on others;
- are content with mediocrity;
- procrastinate and leave revision and assignments to the last minute;
- hope to pass with the least possible effort;
- chronically skip lectures and practical classes;
- lack focus and habitually ignore or get confused about instructions;
- try to buck the system by copying assignments;
- do not take responsibility for their actions but blame others (staff members, the course, other students) for their misfortunes;
- exaggerate and whine about the work load;
- are disrespectful to staff members and fellow students.

The fact that you have been admitted into a university degree programme is strong, objective evidence that you have the ability to complete successfully the degree. Nevertheless, far too many students in our country drop out of university after floundering around and trying to pass all the prescribed courses in the permitted time. This is both sad and unnecessary. Of course, there are genuine personal reasons for failure: financial difficulties, family problems, illness, etc. However, most course failures are due to the student’s life position of I’m not OK – you’re OK. Such people tend to behave like victims. They find all sorts of persecutors and tend to drift, passively hoping that somehow they will be rescued.

I hope that those who read this reflection will not fall for the lie of being a victim but will, instead, take up the challenge of personal responsibility. The buck stops with you!

Finally, I congratulate Vreda Pieterse for producing this excellent little book. It is a testimony to her own can-do sense of responsibility for accepting the challenge to fill a gap which has been present for many years.

Derrick G. Kourie
January 2016
Chapter 1

Netiquette

1.1 Introduction

Netiquette is a term derived from merging the words network and etiquette. The term refers to a set of social conventions that facilitate interaction over networks, including blogs, forums, wikis, Facebook and twitter\(^1\). The rules are made to avoid misunderstanding. Ultimately, the netiquette rules should support reasonable behaviour and good will while preserving the spirit of flexibility, supporting the objectives of sharing and creating knowledge.

One follows netiquette rules as a matter of courtesy to other users of the same online community. It reduces noise and consequently contributes to the quality and findability of the knowledge that the online community creates. Professional unity is formed when a community adheres to an agreed-upon code of conduct. Consequently, the members of such a community find that their forum is a pleasant place to search for information. Ideally, all members should show a sense of professionalism as this leads to mutual respect among the members. This attitude is conducive to creating higher levels of knowledge and achieving deeper levels of learning.

This chapter discusses general conventions as well as some specific rules. You are encouraged to adhere to these when posting on the forum of your introductory programming module or submitting queries to your instructors by e-mail. This may help you to have a rewarding online experience.

\(^1\)[http://en.wikipedia.org/w/index.php?title=Etiquette_(technology)]
1.2 General rules

Watching what others do is a good strategy for newcomers to an established online community. It allows the newcomers to judge the tone of the community before launching in, and so avoid causing offence, being ridiculed or humiliated. You should be able to distinguish between experienced users and novices by viewing the profiles of the participants. You should avoid following the example of newcomers to the forum and repeating their blunders.

The online forum of an introductory programming module is likely to have more newcomers than experienced users. When participating in such an immature forum, you should learn about the expectations outlined in this chapter. This will enable you to become a valued participant who sets a good example for others to follow.

Many attempts have been made to identify and publish common standards of netiquette. It seems unlikely that a single global netiquette will be established to govern all online behaviour. Scheuermann and Taylor [37] identify the following most frequently cited specific suggestions for online users:

- think first;
- write in upper and lower case;
- avoid abbreviations;
- be concise;
- avoid smileys;
- don’t flame;
- don’t take offence easily;
- don’t evangelise;
- know the audience.

These guidelines should be kept in mind in all situations where you participate in online communication. The premise is that most people would rather make friends than enemies, and that if you follow a few basic rules, you are less likely to make the kind of mistakes that will prevent you from making friends.
1.3  E-mail

When sending e-mail messages, you should keep in mind that, although the communication is assumed to be private, the content is often stored on insecure mail servers. Do not write anything that you are not willing to say in public.

We discuss a few rules for enhancing the readability of your e-mail messages. The rules should reduce the likelihood of unwittingly annoying the recipients of your messages.

Subject line

An e-mail message should have a descriptive subject line to represent the content of the message. People who receive large numbers of e-mail messages often rely on the subject line to determine the importance and tone of the e-mail and even use it to pre-sort their inbox so that they can manage their e-mail. If the subject line is meaningless or misleading, the recipient of the e-mail may not notice the message.

It is particularly important to formulate the subject line carefully when sending e-mail messages to someone who may not recognise your name, for example your lecturer or a tutor. If the subject line is accurate, the recipient is likely to identify it and give you a speedy response. If the subject line is incorrect, however, it can easily be overlooked.

When one hits the reply button, the current subject line is used in the subject line of the reply. You are advised to edit the subject line of the return e-mail if the default subject line no longer reflects the contents of the new e-mail.

Body

Always start with a greeting line, including the name of the person to whom the mail is sent. Address the person, as if the communication is face-to-face. Always end by signing your name and giving your full contact details, i.e. include your student number, e-mail address and cell phone number. When writing the message, be concise and include all relevant factual details. For example, if you want to know about the marks you were given for a practical assignment, the following message would be unacceptable:

*My mark for the prac is wrong.*

Members of the teaching staff cannot attend to such a query unless they are prepared to do detective work to figure out the relevant details.
The following is an example of a query which includes all the relevant information and uses the proper netiquette:

Ms Pieterse

On the mark sheet for COS 132 which was published on 2013-05-17 on the CS website, my marks for practical 2 are 23/50. However, my results from fitchfork were reported to be 5/5 for task 1, 15/30 for task 2, and 15/15 for task 3. This adds up to 35. I would be most grateful if this could be corrected.

Thank you in advance,
John Doe
14555555
083 166 9999

Attachments

Avoid attaching documents to an e-mail message. Never send an e-mail without any body text and only an attachment. It often happens that such an e-mail is identified as spam and consequently not delivered to the recipient. Large attachments might annoy high-bandwidth users but could upset users with low-bandwidth connections [34]. Large files should rather be shared by using cloud applications instead of sending them via e-mail.

Wherever possible, attachments should be independent of platform and preferably in a format which your recipient can probably read. The pdf format is one such format which is fairly safe to assume is readable. Avoid sending documents which require proprietary applications to view them.

Avoid phishing and spam

Be cautious of e-mail messages that seem to represent trusted organisations requesting personal or financial information from you. They may appear to be legitimate, but are more likely to be malicious attempts by criminals to collect your information for the purpose of committing fraud or identity theft. For the same reason you should not send your solutions to practical assignments when someone requests this via e-mail.

Avoid clicking on links in an e-mail message. All too often the links that seem to be harmless redirect you to sites where the perpetrator can infect your computer with a virus or collect information from your computer.
1.3. E-MAIL

Responding to e-mail

You should read the WikiHow\textsuperscript{2} article on e-mail etiquette. It contains several guidelines worth following when writing e-mail messages. The essence of e-mail etiquette when writing an e-mail is to pay attention when you read your e-mail and think about what you are sending and to whom you are sending it. When you are unable to answer an e-mail within a reasonable time, it is good manners to acknowledge that you have received it and to state an estimated period when you will be able to respond.

Correspondence in request-tracking systems

Online service providers and software houses usually have to deal with large quantities of customer requests. They often use request-tracking software to assist them. Although the management of e-mails is automated when using such a system, most of the responses are still provided by people. Automatic responses are usually limited to acknowledging the receipt of the request. The following is an example of a general message you might receive from the request-tracking software we use for managing student queries:

\begin{quote}
We acknowledge receipt of your enquiry.
Your ticket has been assigned an ID of [shodan.cs.up.ac.za #1638]
Please note that it may take up to three working days for a person to respond to your enquiry.
\end{quote}

A unique issue number is assigned to the case when you initiate an e-mail to the system. All conversation about this issue should contain this number in the subject line. Therefore, if you want to send a repeat request about the same issue, you should include this number in the subject line. To do so, you simply have to reply to the auto-generated message.

Note that the answers that are provided via this system are written by staff members — real people who spend time doing this. Please wait patiently for an answer. Sending more messages will not make them answer you sooner, it might only slow them down, as they have to inspect each of the multiple requests you have made.

If you want to initiate a new issue, you should send a new message. Do not do this by replying to an old message. If the ID number of a previous issue is in the subject line, the system will link it with the old issue. Such message could easily be overlooked because it will not be flagged as a new issue.

\textsuperscript{2}http://www.wikihow.com/Improve-Your-Email-Etiquette
You might get a response such as the following when an issue is deemed resolved.

According to our records, your request has been resolved. NTN.
Only respond to this message to re-open the ticket.

This means that the staff members have completed all the actions required by of your request and have now signed off. NTN is an abbreviation for no thanks needed. You should honour such a request — really!

The next sentence states that you should only use the reply feature of your e-mail system if you DISAGREE with the decision that the case is now closed. If you were to ignore the NTN request and respond by simply saying Thank you, the ticket is automatically re-opened, although it should have remained closed.

When sending an e-mail to an address that uses a request-tracking system, you are advised to follow these rules:

- Include your name and contact details in the body of the e-mail.
- Provide a short and descriptive subject line.
- Write separate e-mails if you want to make more than one request. Requests about different topics should not be made in the same e-mail. For example, do not enquire about marks and also about a mentor session in the same e-mail message.
- Do not create a new e-mail when continuing correspondence on a topic. This will help to link the e-mail messages about an issue.
- Write a new e-mail if you have a new query (e.g. do not reply to a ticket which was resolved).
- Your first sentence should be a brief description of your enquiry, followed by a new paragraph explaining your problem.
- Write in a formal style and be respectful.
- Check that the spelling and grammar are correct before sending the e-mail message.

### 1.4 Online forum

An online forum is a web-based application where users can write messages and expect to see responses to their messages. Other names which are often
used to refer to online forums include discussion group, discussion forum, message board and discussion board.

The purpose of an online forum is to provide an opportunity for its users to share ideas, help one another and create knowledge. You can benefit by reading the information shown on the forum because it informs you about current issues and possible different opinions. A forum is a place where students who struggle may find solutions to their problems. It also creates an opportunity for students who are able to answer some of the questions, to support their fellow students and to share their knowledge to the benefit of the online community.

It may become difficult for users to find information on a forum if the forum is not well organised. Participants help ensure that information is easily accessible by adhering to the specific guidelines given in this chapter when they participate.

Threads

A thread is a collection of related posts. A thread includes its starting post as well as the tree of replies and comments to such a post. When posting a message, you should find the most suitable thread for your post. If your comment or question does not belong to any of the existing threads on a forum, you should create a new thread. When you start a thread, you should make its scope broad enough to generate adequate participation but also narrow enough to allow a clear focus. You are advised to create a new thread with the utmost discretion, according to the following guidelines:

- Check and see whether a similar thread already exists before creating a new thread.
- Every thread should have an appropriately descriptive heading. It should be short but descriptive enough for everyone to understand the scope of the thread. This scope should strike a good balance and be neither too general nor too specific.
- The body of the starting post that introduces a new thread should have a paragraph describing the general theme of the thread and providing adequate background on the question or intended discussion. This message should demarcate the context of the new thread. It should end with an open-ended question to invite participation in the thread.
- A thread should focus on one (and only one) idea. Threads with more than one focus will lead to inconsistent replies and make it difficult for students to find certain replies and information. Rather post several threads, each with its own focus.
CHAPTER 1. NETIQUETTE

Replies

A reply, sometimes called a post, is a comment written by a participant as a reply to any other message on the forum. Use the following guidelines when writing replies:

- Keep replies short. Shorter messages are easier to read. Readers tend to skim over long-winded arguments and may miss the essence. One hundred words per message is a good norm. It is of the utmost importance for you to be clear and concise in your replies.

- Stick to the topic of a thread. Valuable information posted in an unrelated thread is often lost because people tend to look for the information where they expect to find it.

- It is not necessary, nor advisable, to start a post by introducing yourself or to add your name or contact details at the end of a post. The necessary information (e.g. name and photo) is usually part of your profile and automatically appears with your posts.

- Check the spelling and grammar of your post before submitting it. Most modern browsers have a built-in spell checker. If your browser does not have one, simply copy your text into a word-processing program which has one and do a spell check in the word-processing program.

Noise

Three types of noise occur in online forums: off-topic messages, redundant messages and zero contribution messages.

A reply is considered off-topic if any part of its content is not related to the descriptive heading of the thread where it is posted. If a post does not coincide with the descriptive heading of the thread in which it is posted, it is considered off-topic even if it is related to some other reply in the thread. It is usually not a good idea to reply to a post that is off-topic because it contributes to noise.

A thread or post is redundant if its content is already available on the forum. There is no need to say exactly the same thing several times. Sometimes a discussion may include different views on the same theme, which is acceptable. When participating, you should try to contribute by adding new ideas or examples instead of simply repeating what has already been said.

The worst form of noise is zero contribution. Talking for the sake of talking is nice at a party but posting for the sake of posting clutters the
forum, making it difficult for users to find useful information among all the noise. When posting, you should think before you hit the post button. If the answer to the following question is “no”, consider not posting.

Will the other users of this forum find my post interesting or will they benefit in some way from what I am saying / asking?

1.5 Style and formatting

In face-to-face conversation, body language and context play important roles in communicating the message. Printed matter uses images, layout, fonts and styles to portray more meaning and to promote correct interpretation. Inappropriate formatting often makes messages difficult to read. The following rules should be kept in mind when formatting your messages:

- Avoid using special keys such as the tab character. Although it might look fine while you are editing your post, it may be unintelligible when it has been posted.

- Only use the features that the message editor provides. For example, if the editor does not have the functionality to insert a character such as è you may omit the caret or use a plain text alternative such as ^e.

- Type your message by using the normal capitalisation rules. Do not use all caps or all lower case. USING CAPITALS WHEN YOU WRITE IS LIKE SHOUTING WHEN YOU SPEAK! By contrast, it is considered disrespectful if you do not use capitals at all, but type everything in lower case.

Writing style

When participating in an academic forum, use a professional writing style. Occasionally, less formal comments may be appropriate. Learning is serious, but you should also have fun. Keeping a sense of humour supports a good quality of life. Our sense of humour gives us the ability to find delight, feel joy and release tension. Horowitz [16] remarks that humour should always be used appropriately and sensitively.

A professional writing style and carefully selecting the words you use can improve the quality of your contribution and also make your message
clearer. Keep in mind that some participants may not be fluent in English. Be forgiving in the interpretation of other’s posts.

The use of SMS language in a professional forum is unacceptable. You should avoid using uncommon abbreviations so that you can prevent your message from being misinterpreted. Find out what the exact meaning of an abbreviation is before you use it and use abbreviations sparingly. Some commonly used abbreviations are listed in Table 1.1.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASAP</td>
<td>As soon as possible</td>
</tr>
<tr>
<td>BTW</td>
<td>By the way</td>
</tr>
<tr>
<td>F2F</td>
<td>Face-to-face i.e. in person</td>
</tr>
<tr>
<td>FAQ</td>
<td>Frequently asked questions</td>
</tr>
<tr>
<td>FYI</td>
<td>For your information</td>
</tr>
<tr>
<td>IMHO</td>
<td>In my humble opinion</td>
</tr>
<tr>
<td>LOL</td>
<td>Laugh out loud</td>
</tr>
<tr>
<td>NTN</td>
<td>No thanks needed</td>
</tr>
<tr>
<td>TY or Tx</td>
<td>Thank you</td>
</tr>
<tr>
<td>VG</td>
<td>Very good</td>
</tr>
</tbody>
</table>

**Emoticons**

The online community has developed a set of text symbols, called emoticons, to use in text messages to solve the problem of not having communication tools such as your tone of voice and facial expression when you are limited to using plain text to communicate. For example, if you look at the character string : - ) as if the colon is the top, it is a picture of a smiling face (two eyes, a nose and a smiling mouth). Further capitalising on the human ability to see a face with the aid of minimal information, the character string 0. 0 can easily be seen as two wide eyes and a straight mouth, symbolising a surprised face, and \o/ is a small head between two arms which are thrown up in the air. Table 1.2 lists a number of commonly used emoticons, some requiring a bit more imagination than others. In many editors, symbols like these are automatically turned into graphic images. You may include emoticons in your messages to convey some emotion which in turn can contribute to the correct interpretation of your message. Emoticons are special abbreviations.
The guidelines for using abbreviations apply equally to the use of emoticons, i.e. use them sparingly and only when you know what they mean.

<table>
<thead>
<tr>
<th>Emoticon</th>
<th>Meaning</th>
<th>Emoticon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:-)</td>
<td>the basic smiley</td>
<td>:- (</td>
<td>a sad face</td>
</tr>
<tr>
<td>::</td>
<td>a stern face</td>
<td>:: /</td>
<td>annoyed</td>
</tr>
<tr>
<td>:: X</td>
<td>embarrassed</td>
<td>:: 0</td>
<td>a wide smile</td>
</tr>
<tr>
<td>; - )</td>
<td>winking smiley</td>
<td>; - P</td>
<td>tongue sticking out</td>
</tr>
<tr>
<td>: - &amp;</td>
<td>I am tongue-tied</td>
<td>: - o</td>
<td>I am bored (yawn)</td>
</tr>
<tr>
<td>&gt; : [</td>
<td>a frown</td>
<td>0 . 0</td>
<td>surprised</td>
</tr>
<tr>
<td>\ o /</td>
<td>excited</td>
<td>o / \ o</td>
<td>high five</td>
</tr>
<tr>
<td>* \ O / *</td>
<td>cheer leader</td>
<td>( ... )</td>
<td>disappointed</td>
</tr>
<tr>
<td>( ^ o ^ )</td>
<td>laughing</td>
<td>( _ ^ ' )</td>
<td>nervous</td>
</tr>
</tbody>
</table>

1.6 A final note

Common errors

Inappropriate participation in online communication is often caused by one or more of the following:

- Inattentive reading – when you skim through a message, you might misinterpret it.
- Wrong assumptions – when you base your interpretation of a message on a wrong assumption, it is bound to be skewed. Read more about wrong assumptions on goodread.4
- Flamed response – when you write while you are annoyed, you are likely to write things you may regret later.

Survival kit

If you keep the following in mind when posting, it is likely that others will appreciate your participation:

- Use a formal writing style. Do not use SMS language, and minimise the use of abbreviations and emoticons.

4http://www.goodreads.com/quotes/tag/assumptions
• Do not use foul language or be disrespectful to anyone.
• Use the general rules of capitalisation and punctuation. You should not use ALL UPPER CASE or only lower case for an entire message.
• Do not start a thread on a topic which is already covered by another thread, and do not answer off-topic posts.
• Make sure that your post is of interest to the readers of the forum.
• Keep your headings/titles short and descriptive.
• Check the spelling and grammar on your post before submitting it.

Put netiquette in your pocket

Netiquette can help novice users to avoid mistakes, but being too wrapped up in rules may mean that you lose your perspective. There are times when it is better to ignore a rule. If a contribution violates the precise wording of a rule, it might still be a good contribution. Similarly, just because something is not explicitly forbidden, it does not mean it is a good idea. The principle of the rules is more important than the letter of the law.

Participation in forums may give you a great learning experience. You will gain knowledge of the content discussed on a forum and may develop life skills such as etiquette, communication skills and the ability to analyse situations. Most of all, you may be rewarded by the pleasure of being able to help and support your peers.
Chapter 2

Coding standards

2.1 Introduction

A coding standard can be broadly defined as a set of programming styles and practices to which a group of people adhere, in the belief that such adherence contributes to the overall efficiency of producing high-quality code that is understandable and maintainable.

The benefits of having uniform coding styles and standards are widely discussed. Scott Ambler, the Practice Leader Agile Development at IBM Corporation in the IBM Methods group, once said:

*Inexperienced developers, and cowboys who do not know any better, will often fight having to follow standards. They claim they can code faster if they do it their own way. Pure hogwash. They MIGHT be able to get code out the door faster, but I doubt it. Cowboy programmers get hung up during testing when several difficult-to-find bugs crop up, and when their code needs to be enhanced it often leads to a major rewrite by them because they’re the only ones who understand their code. Is this the way that you want to operate? I certainly do not.*

Coding standards are laid down to achieve robust and error-free code that is easy to use, understand and maintain. Style is a crucial component of professionalism in software development. Adhering to styles and standards based on good programming practices is beneficial for sharing and understanding code. Clean code that follows stylistic conventions is easier to read, maintain and share with colleagues. When a consistent style is used throughout a project, it makes it easier for the developers working on the project to understand one another’s code.

1http://www.ambysoft.com/downloads/javaCodingStandards.pdf
Oman and Cook [31] found through empirical studies that the style used when writing or maintaining a program has a direct impact upon the quality of the software and the comprehensibility and maintainability of a program.

Coding standards and guidelines can be classified into the following five broad categories [32]:

**Style** – guidelines and standards in this category deal with layout issues.

**Clarity** – this is about the measures, besides typographic rules, that can make code easier to read and understand.

**Flexibility** – in the software engineering industry, it is paramount for software developers to be agile. These standards are guidelines to enhance the adaptability and portability of the code.

**Reliability** – this concerns the production of user-friendly, robust and error-free code. These guidelines and practices are aimed at reducing the chances of making common, silly, programming errors and reducing the chances of program malfunction due to user actions.

**Efficiency** – this is about writing elegant code that uses its resources sparingly. Standards aimed at efficiency are rules which should be followed to use resources such as memory, CPU time and disk space efficiently, without compromising other resources such as programmer effort and money.

In the following sections, each of these categories is discussed in more detail. Particular standards are specified. Many of these have been adapted from the specifications for coding standards by Horstmann [17]. You are advised to adhere faithfully to these standards while learning to program. In any commercial enterprise where software development is important, you will find a similar set of coding standards, probably differing in some of the details, but addressing the categories mentioned below.

### 2.2 Style

Standards related to style prescribe typographical requirements. The purpose of these standards is to improve consistency and neatness in the appearance of the code. Adherence to these standards enhances program readability and reduces the likelihood of syntax errors.
2.2. **STYLE**

**Naming conventions**
- Use **ALL_CAPS** for named constants, and camelCase for all other identifier names.
- Identifiers of variables, functions and methods should start with a lowercase letter.
- Class names should be capitalised (start with an initial capital letter).

**Layout rules**
- Use blank lines freely to separate the parts of a function or method that are logically distinct.
- Use a blank space around binary operations.
- Leave a blank space after (but not before) each comma, colon or semicolon.
- Lines of code should never extend beyond the right-hand side of a reasonable window width on the screen. Limiting each line of code to 80 columns will ensure that this is achieved.
- If a hard copy of a program list is made, insert page breaks to prevent code blocks from spanning over page breaks.
- Use indentation and blank lines to reveal the subordinate nature of blocks of code. Each line which is part of the body of a control structure (if, while, dowhile, for, switch) is indented one tab stop from the margin of its controlling line. The same rule applies to function, struct or union definitions, and aggregate initialisers.
- We do not require the specific placement of opening and closing braces. We do, however, require consistency. Sutter and Alexandrescu [45] state that a professional programmer will not have difficulty in reading any of the following styles. Choose one of them, or anything similar, and use it consistently throughout all the code of one project.

```cpp
int main()
{
    cout << "Programming is great fun!" << endl;
    return 0;
}
```
int main()
{
    cout << "Programming is great fun!" << endl;
    return 0;
}

int main()
{
    cout << "Programming is great fun!" << endl;
    return 0;
}

2.3 Clarity

Clarity is about the measures, besides typographic rules, that make code easier to read and understand. The organisation and order of presentation, the careful selection of identifier names, and the content and writing style of comments play an important role in the clarity of code.

Organisation and order of presentation

- When the main function calls other functions, they may be defined in the same file. In this case, you should list all the function prototypes above the definition of the main function. Their definitions should follow the main function in the same order that their prototypes are listed.

- Functions that are called in the main function may be defined in a different file. In this case, you should list the function prototypes in a header file which is included in the file for the main function. Their definitions should be included in a separate source file in the same order as the list of prototypes in the header file.

- In a class definition, you should list all its public members, then all its protected members and lastly all its private members. List methods before instance variables.

- For each class, place the class definition in a header file which is included in the source file that implements the methods of the class. The implementation of methods should be presented in the order in which their prototypes are listed in the header file.

- Program sections should be listed and grouped in a logical order which will enhance comprehension.
2.3. **CLARITY**

- The grouping of program sections should maximise the cohesion of groups, and minimise coupling between groups.

- The beginning and end of a program block should fit on one screen. Long code sections can always be defined in terms of a number of smaller functions. A good norm is between seven and 15 lines of code in a block. Do not exceed 30 lines of code in one block.

**Selection of identifier names**

Although the compiler only needs a unique character string to identify an entity, programmers also rely on their meaning. Identifier names are intended for the convenience of readers, and should not be a shortcut for the writer.

- Use nouns to name classes and variables.

- Use verbs to name functions and methods.

- Apart from being of the correct word type (noun or verb), an identifier name should be reasonably long and describe the purpose of the identifier in the program.

- Avoid the use of names that are too general.

- Avoid the use of abbreviations (e.g. calc for calculate). Use dictionary words\(^2\). Exceptions to this rule include using single characters or cryptic variable names for loop counters and for the parameters of a constructor, provided that these variables are used in an accompanying initialiser list.

- Never use the single characters “O”, “o” or “é” (which normally show on displays as “1”), and avoid using them as the last character in an identifier—these two characters can easily be confused with 0 and 1.

**Commenting practices**

Comments are included in code to clarify code and give the additional information that cannot be included in the code. The principle is rather to write self-documenting code than to over-comment. It is important to realise that comments cannot rectify bad code. As Oman and Cook [31] put it, “Don’t comment bad code - rewrite it”.

\(^2\)Note that this standard may be violated in handwritten code snippets.
Comments are used to enhance the clarity of automatically generated documentation. For this reason, comments which are embedded in the code should follow the syntax specified by the documentation generator of your choice.

- Avoid redundancy and duplication of what is already clear in the code. This rule is often violated by programmers who have the impression that the mere presence of comments serves a purpose. More often than not, extra comments obscure more than they clarify.

- Make sure that comments and code agree. Often programmers change code without updating the accompanying comments. This is unacceptable. Inaccurate comments are worse than no comments at all.

- Use a formal writing style to state facts in full sentences that are concise and to the point. Writing concise explanations is often trickier than writing code!

- Every function definition should be preceded by comments that briefly describe what the function does. Bear in mind that because the documentation generator of your choice uses these comments when generating the documentation, you should use the proper tags the generator requires. The best way of specifying what a function does, is to provide the following:

  - Give the function's *precondition*. Do this by describing each function parameter. For each parameter, indicate any restrictions on the values it may assume in order to guarantee that the function will work correctly. For example, if an integer parameter called *income* has to be positive for the function to work properly, this should be stated explicitly.

  - Similarly, give the function's *postcondition*. Do this by describing what the function will return, or how the function will change the state of the code.

The pre- and postconditions can be seen as a contract between the person who wrote the function and the person who uses the function in a program: if the programmer who uses the function ensures that the parameters comply with the preconditions, then the writer of the function guarantees that the function will comply with its postcondition.

---

3) Javadoc can be used when writing Java code and Doxygen is suitable for C and C++ code.
Control structure style

It is extremely important for other programmers (and you after a while) to be able to follow the program flow of your code. To aid in this respect, you will find it useful to adhere to the following conventions which are aimed at the simplicity and clarity of program flow:

- A function or method should be a pure accessor or a pure mutator. Avoid using reference parameters in value-returning functions or methods.

- Use jump statements responsibly. These are goto, break, continue and return statements that are intended to short-circuit a loop or to leave a function or structure at a point other than its end. Minimise the use of break statements in loops, and avoid the use of continue and goto statements altogether. It is preferable to avoid using return to break out of a loop—rather terminate the loop gracefully (by having a suitable loop condition) and then return after the loop’s termination.

- Give preference to the use of while loops. Use for-loops only when a variable runs from somewhere to somewhere with some constant increment/decrement.

- Avoid confusing programming tricks [30].

- Avoid deep nesting of loops and conditionals [30].

2.4 Flexibility

Flexibility standards are guidelines to assist programmers with building adaptable and portable code. If code is adaptable, it can easily be changed and reused. If code is portable, it can easily be moved to another platform or environment.

- Avoid the use of “magic numbers”. A magic number is a numeric constant embedded in code. Rather introduce a named constant. An example of the use of a magic numbers is when you hard-code 3.14 where \( \pi \) is used in a formula. Rather use the named constant \( \text{M_PI} \) defined in the math library for \( \pi \). Another example is when you have the same hard-coded end value in several loops in the program. In such a case, rather introduce a named constant such as \( \text{int MAX = 100;} \) and use \( \text{MAX} \) instead of the hard-coded value in each of the loops.
• Write your programs in a modular fashion. Functions should be used to split up functionality. This splitting should be done logically by grouping together similar functionalities (which will make sense as a unit). Avoid having too few or too many functions. Try to maximise cohesion within a function and minimise coupling between functions.

• Apply the object-oriented programming principles, such as modularity, encapsulation and independence.

• Apply the appropriate design patterns wherever possible.

2.5 Reliability

Code that is written to be flexible is usually also more reliable. This is because when you enhance flexibility (according to the guidelines below) you also contribute to localisation — i.e. the information relevant to particular parts of the code is close together, making it less likely to have syntax errors. It is also easier to isolate possible logical errors, making the code less prone to error. Furthermore, code that adheres to standards which are aimed at eliminating human error and enhancing usability will contribute to its user-friendliness and will therefore be more robust and reliable.

Scope and accessibility

Wikipedia\(^4\) describes a side-effect as follows:

...a function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world.

Side-effects are unexpected behaviour originating from unintended changes in the values of variables in the program. This should be avoided as far as possible.

• All non-final variables (interim values) should be private.

• Keep accessibility as private as possible. Avoid global variables and minimise the use of static variables.

• All features should be explicitly tagged public, protected or private — avoid using the default visibility.

\(^4\)http://en.wikipedia.org/w/index.php?title=Side_effect_(computer_science)
2.5. RELIABILITY

- Define each variable just before it is used, rather than defining all variables at the beginning of a block.
- Avoid having instance variables of a class which could have been defined as non-final variables in the implementation.

User orientation

The robustness of a program depends on how well the user understands its use. For this reason, one should strive to write user-friendly code.

- Avoid clutter on the screen.

- When prompting a user:
  - be as exact and complete as possible about what would be acceptable input.
  - In command line prompts, end the prompt string with a colon and a space and do not use a new line. The user input should be typed on the same line as the prompt.

- When displaying results:
  - display the result in a complete, grammatically correct, sentence;
  - be as exact and complete as possible with regard to the meaning of the result.
  - If applicable, include the input values that contributed to the result in the output.

- When displaying an error message:
  - be consistent in the appearance of different error messages throughout the program;
  - be as exact and complete as possible about what went wrong. For example, if a file could not be opened, you should include in the error message the name of the file that could not be opened.

Avoiding logical and runtime errors

The following good habits may lead to code that is less likely to contain logical errors. It may also help to reduce common runtime errors, such as reference to uninitialised objects (segmentation faults) and overflow or underflow.
• Make sure that variables are initialised before they are used. It is best to provide default values upon declaration [28].

• Test your program with data which includes all possible extreme cases as well as all conceivable user misinterpretation.

• Take compiler warnings seriously [28]. It is important to make sure that before you dismiss a warning, you understand exactly what the warning is trying to tell you.

• For every class with dynamic instance variables, you should explicitly declare a default constructor, copy constructor, assignment operator and destructor.

2.6 Efficiency

Efficiency is about writing code that is elegant and at the same time being aware of resource usage. You are advised to read Effective C++ [28] for a comprehensive discussion of specific ways to improve the effectiveness of your code. Adhering to these guidelines will greatly improve your code. One should realise that even when following the guidelines to the letter, this might not guarantee good code. Efficiency is based mainly on algorithms and is inherently situational. Here we mention only a few of the prominent ways to avoid gross inefficiency:

• Never declare variables that are not used.

• For each variable, use the smallest data type that will comfortably hold the expected extreme values.

• Avoid the need to apply type casting.

• If the same (or very similar) code appears in more than one place in the program, put it in a function which can be called more than once.

• If a number of consecutive lines of code are the same (or very similar), find a way to specify the operation performed by the code, using a loop structure.

2.7 A final note

Common errors

It is quite common to have the wrong impression that coding standards are mainly about layout rules and naming conventions. Once you have
grown to understand how compliance with coding standards improves the quality of your code and makes your life easier, you will develop your own programming style which can be adapted to comply with any reasonable coding standard expected in the workplace.

**Survival kit**

If you keep the following items in mind when writing code, your code will probably comply with coding standards.

- Do **NOT** write comments which repeat what is already clear in the code.
- Use descriptive names which are not too general. Avoid abbreviations.
- Use consistent indentation and blank lines to enhance readability.
- Declare named constants to remove magic numbers.
- Use the smallest data-type that is big enough to hold all the anticipated values. Also be mindful of possible overflow.
- Give complete and grammatically correct information in prompts and error messages.
- Use loops and functions to avoid code duplication.
- Avoid confusing programming tricks.
- Avoid deep nesting of loops and conditionals.
- Focus on border cases and extreme values when testing your code.

**Put coding standards in your pocket**

The purpose of these coding standards is to introduce students to a representative set of coding standards which are typical of professional programming practices, and to help students develop the habit of good coding style, which is necessary to complete large programs successfully.

Compliance with coding standards is a vital professional skill required by the software industry. A coding standard is aimed at enhancing code quality and uniformity. When complying with a standard, code becomes easier for everyone to read, and simpler for other people to analyse, debug and maintain.

We hope that that the contents of this chapter will not only convince students of the benefits of compliance with coding standards but will also provide guidelines to enhance the quality of the code they write and to inspire them to strive for excellence.
Chapter 3

Academic integrity

3.1 Introduction

To ensure academic integrity, students are encouraged to participate in all academic activities with the positive attitude advocated in the foreword and in Chapter 8. They should avoid any practices which may give them a short-term unfair academic advantage. In this context, academic integrity is deemed to be the opposite of plagiarism.

This chapter is intended to educate students about the aims of practical assignments and the role of academic integrity in achieving these goals. It clarifies the meaning of concepts such as intellectual property and plagiarism. The consequences of academic dishonesty and the futility of cheating are highlighted. It is important for students to learn about software engineering techniques so as to enhance code quality and speed up development. These include collaborative development and code reuse. When students have a deeper understanding of academic integrity, it is more likely that they will be able to develop their skills in constructive collaborative work and proper code reuse, without fear of being suspected of cheating or plagiarising.

Section 3.3 describes ways in which programmers work with existing code. Judgement errors that may hamper the achievement of the learning goals of practical assignments are illustrated in Section 3.4. These may lead to counter-productive or even dishonest behaviour.

Students should illustrate their understanding and honesty when submitting their code for assessment. They can do this by acknowledging the resources they use, along with a description of their own contribution to the code base that they have submitted for assessment. The required documentation is discussed in more detail in Section 3.5 while Section 3.6
addresses collaboration.

3.2 Practical assignment goals

The purpose of practical assignments is to serve as an exercise so that students can gain a deeper understanding of specific programming concepts or techniques. Programming assignments are primarily given to create an opportunity for students to apply a specific concept so as to gain experience in the application of the concept. The marks that are awarded for an assignment should never be seen as a goal on its own. You should regard marks merely as a way to gauge the level of your understanding. Students who understand and subscribe to this reason for doing assignments, will also understand the futility of submitting work which is not the outcome of their own learning experience.

When tackling an assignment, the goals of the assignment are best achieved when the student has identified the learning objectives that can be associated with a task. If the student does not do this, the result might be to create a solution to the problem by using an alternative method which may not help the student to gain the intended knowledge and skills. Always try to solve the given problems in a way that has the best potential to provide the practical experience for gaining a deeper understanding of the content that is taught at the time when the assignment is given.

Nothing prevents you from opting for shallow learning so that you can gather marks while avoiding every opportunity to expand your knowledge – but in the long run, you will have greater difficulty in keeping up with acquiring in-depth knowledge to expand your programming proficiency.

3.3 Working with existing code

Hacking

Many students seem to have the impression that they are reusing the code when they alter existing code to suit a specific need which is similar to the intention of the original code. As explained in Section 3.3, this is not the case. Doing this is sometimes called hacking\(^1\). In the sense that the word hack is used here, it describes the actions of an inexperienced programmer who uses trial and error to change sections of code without a proper understanding. This is probably the most ineffective way to learn how to program.

---

\(^1\)Hacking is more often used to refer to the action of secretly gaining access to a computer system.
A student gave the following advice on the COS132 course discussion board in 2013 to students to explain how one can honestly learn by using existing code. He also points out that dishonesty is futile.

*Study the code and understand what it does. Then code your own solution; this makes you better at coding and eliminates the need to plagiarise. If you just copy code all the time what do you learn?*

If you were to hack an assignment which was created by someone else, and submit the altered code as your own, this would be deemed plagiarism. It should be clear that hacked code is not a form of proper code reuse.

**Refactoring**

Fowler [9] defines refactoring as follows:

*Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which is “too small to be worth doing”. However the cumulative effect of each of these transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring – which allows you to gradually refactor a system over an extended period of time.*

In large legacy systems, refactoring may be applied to gradually improve the design of the system instead of replacing it. Refactoring can also be applied on a smaller scale to improve the readability, clarity, reliability, user-friendliness, flexibility, portability, effectiveness and efficiency of small portions of code. Sadly, dishonest students use simple refactorings in an attempt to hide the fact that they have plagiarised their code.

Martins et al. [25] categorise the refactorings that students commonly use to avoid the detection of their copied code by software for identifying code similarities. Table 3.1 was compiled using Martin’s categorisation as basis. It defines seven refactorings applicable to the kind of programs that students are expected to write for introductory programming assignments.

Code to which refactorings were applied, such as those listed in Table 3.1, can easily be mistaken for plagiarised code unless the original source is properly cited and is accompanied by a valid change log, justifying each of the refactorings in terms of an improvement, as suggested in the coding standards described in Chapter 2.
Table 3.1: Simple refactorings

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change comments</td>
<td>Edit, remove or add comments.</td>
</tr>
<tr>
<td>Change identifiers</td>
<td>Rename, introduce or eliminate variables or constants or rename functions.</td>
</tr>
<tr>
<td>Change scope</td>
<td>Change the scope of variables, for example, move local variables to global scope and <em>vice versa</em>; or move variables local to the body of a loop, out of the loop and <em>vice versa</em>.</td>
</tr>
<tr>
<td>Use alternative</td>
<td>Use a different construct to achieve the same results, for example, use a different mathematical expression in a conditional statement; use a different loop type; use a different data structure or data type; or use an alternative function for I/O.</td>
</tr>
<tr>
<td>Change order</td>
<td>Swap statements that may be executed or listed in a different order, for example, change the order of variable declarations, parameter lists or function definitions.</td>
</tr>
<tr>
<td>Change modules</td>
<td>Change the modular design of a program, for example, move the body of a function to the position where the function was called; move a block of code into the body of a function; or move a function from one class to another.</td>
</tr>
</tbody>
</table>

**Code reuse**

It is important in software engineering to write code in ways which will maximise code reuse. To this end, software engineers should know where to find reusable code and how to incorporate it into their software. The quality of software can be improved through the efficient use of the expertise of experienced programmers. Software reuse has the potential to increase productivity by reducing development time [47, 21].

Code reuse, also called software reuse, is the use of existing software, or software knowledge, to build new software [10]. The key idea in code reuse is that parts of a computer program written at one time could or should be used in the construction of other programs written at a later time. To this end, programmers define templates, functions and procedures that are generic enough to be used in new situations without the need to change the original code. These are often published in software libraries.

3.4. PLAGIARISM

The C++ language was designed to encourage reuse [43]. It is likely that you reused code which is in the *iostream* library when you coded your very first program. When using a library, you simply specify that the library is included and then the code that is in the library may be used as if it were native to the source code. You will soon learn how to use more libraries and later how to build your own.

Software reuse includes the design and implementation of software libraries to extend the code base of reusable code, domain engineering methods to promote the development and deployment of reusable code and enhance the reusability of code in systems, as well as the development of tools to support software reuse.

Proper code reuse avoids “copy-and-paste programming”. The following are basic forms of code reuse in a small program. The code that is reused is written in a general fashion in order to reuse it in different specific ways.

- If the same (or very similar) code appears in more than one place in the program, put it in a function that can be called more than once.
- If a number of consecutive lines of code are the same (or very similar), find a way to specify the operation performed by the code using a loop structure.

3.4 Plagiarism

Definition

The use of code that originates from sources such as textbooks, lecture notes, online tutorials, online code repositories or even peers, does not necessarily constitute plagiarism.

Code plagiarism is defined as using someone else’s code and passing it off as your own. It does not matter if you copied the code from someone else’s work without consent, copied it from a friend’s work (even with consent) or copied it from other sources, such as textbooks or the Internet. In some cases, reusing your own code may be seen as a form of plagiarism called self-plagiarism [4].

Note that the use of someone else’s code is not condemned. Using other people’s work and ideas is fundamental to learning and should be encouraged [11]. The difference between honest code reuse and code plagiarism lies in the integrity of the person reusing the code. In the case of plagiarism, the person submitting the code pretends that the work is his/her own. It is dishonest to reuse code with the criminal intention of getting marks (or money or fame) for work that someone else did.
When is code plagiarised and when not?

When one looks only at the end product, it is almost impossible to distinguish whether or not the code was plagiarised. Only the person who created the submitted version of the code will know whether the work done can be classified as an honest action of learning (this is not plagiarism) or as an action to get marks with minimal effort (this borders on plagiarism).

Many students seem to think that as long as the work they submit cannot be clearly identified as plagiarism, then it is acceptable. Some students may feel that in certain situations, copying code or hacking code with the wrong intentions may be justified. This contrasts sharply with the value-based attitude towards learning discussed in the foreword, an attitude we hope to instil.

For this reason, students are encouraged to write documentation in their code files to provide additional information to state their own intent and to enable assessors to determine the truth. We are not blind to the fact that students may lie in their documentation to deceive the assessor – at least the onus is on them to tell the truth. Students who lie and cheat are unlikely to succeed.

3.5 Regulations

We suggest that the regulations described in this section should be used when submitting an assignment for assessment. They are aimed at ensuring that the correct learning has taken place.

Adherence to these regulations is intended to allow students to be open and honest about their work and is likely to result in a deeper understanding of academic integrity. It also creates an opportunity for students to apply their knowledge to improve the quality of their code in terms of compliance with the coding standards discussed in Chapter 2. These coding standards are fairly generic and quite comprehensive. They include guidelines on improving various quality attributes such as readability, clarity, reliability, user-friendliness, flexibility, portability, effectiveness and efficiency [32].

Preamble

A preamble should be included in each file. It is a comment block at the top of the file and contains the following information:

- The name(s) and student number(s) of the author(s).
- The date of the last edit.
• A short paragraph, describing the purpose of the code in the file.

• If applicable, a description of how this file relates to other files in the submission with the inclusion of the file names of related files.

• A declaration.

Use the proper tags for author and date that are specified by the documentation generator of your choice. We recommend Javadoc\(^3\) for Java code and Doxygen\(^4\) when writing C or C++ code. Most of the tags for these documentation generators are standardised across different generators.

If you completed the assignment on your own without any assistance, the declaration should state that the content of the file is the author’s own work.

If you received guidance from someone or used resources such as online tutorials or your textbook, you should mention this in the declaration. These resources should be properly cited in the manner described in the next paragraph.

If the file contains code that is a variation of existing code, the declaration should state this clearly. The original resource should be properly cited. In addition, the submission must be accompanied by a file containing a change log. The declaration must state the file name of the required change log.

If you reuse code libraries other than those in the standard template library, you have to cite the URL where the libraries can be found or include the libraries in your submission.

Citation of resources

The aim of citing resources is to provide enough information for the readers to locate the original resource. Citations should preferably be given in a standard style such as APA\(^5\) or Harvard\(^6\). The following specifies the details that must to be included for each of the different types of resources:

**Person** – the name and role of the person. The role specifies the relation between you and the person, for example your tutor, friend or dad. If the person is someone in the class, his/her student number is required. Also give the specific date(s) when this person assisted you.

\(^3\)http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html
\(^4\)http://www.stack.nl/~dimitri/doxygen/index.html
\(^5\)www.apastyle.org/
\(^6\)openjournals.net/files/Ref/HARVARD2009%20Reference%20guide.pdf
Own code base – details about when, where and why the code was originally written. Ideally the code should be made publicly available, for example on your blog or Facebook page.

Online resource – the author, the agency, the year of publication, the title of the page and the complete URL. If unknown, either the author or the agency may be omitted, but you may not omit both.

Book – the title of the book, the author, year of publication, publisher, place of publication and the page numbers.

Journal article – the title of the article, the author, year of publication, journal name, volume number, issue number and page numbers.

Conference paper – the title of the paper, the author, year of publication, conference name and abbreviation, and page numbers.

If some of the required facts are unknown, this should be stated explicitly. For example if the year of publication is unknown, you should write “No date” in the citation.

Change log

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refactor</td>
<td>Apply one or more of the refactorings described in Table 3.1 to improve the quality of the code in terms of the guidelines specified in our coding standards.</td>
</tr>
<tr>
<td>Add functionality</td>
<td>Insert new functionality to extend the source to suit the requirements of an assignment.</td>
</tr>
<tr>
<td>Remove functionality</td>
<td>Remove redundant functionality if the source code contains code that is not required for the assignment.</td>
</tr>
</tbody>
</table>

A change log must be included so that the assessor can evaluate the level of understanding achieved by a student who has applied some changes to existing code.

The aim of the log is to enable the identification of the portions of the code and algorithms that are original. It helps the assessor to gauge the amount and quality of the programming work that was done by the student who submitted the code. Instead of determining the value of the code base as a whole and awarding a mark for the correctness and quality of the
complete code base, the assessor may choose to assess only the student’s contribution to the code base.

A change log should describe the changes that have been made and give a reason for making each of the alterations. Possible changes are shown in Table 3.2.

The following are examples of the reasons that may be given for applying some of the refactorings mentioned in Table 3.1:

- Add or reword comments to enhance clarity.
- Change identifier names to make them more descriptive of their purpose.
- Remove redundant comments as the code is clear enough without these comments.
- Change the scope of a global variable to make it local to the function to improve modularity.
- Simplify an expression to remove complexity and/or improve performance.
- Replace a for-loop with an equivalent while-loop to enhance the elegance of the solution.
- Change the order of function prototypes to match the logical order in which the functions are implemented in the program to enhance the clarity of the code.

The change log should be specified in a separate file. The file name of the change log should match the file name specified for it in the preamble to the submitted code.

For each of the changes:

- Describe the details of the change. For example, when renaming identifiers, you are required to give a list, stating the old name and the new name for each renamed identifier.
- Locate the change — i.e. specify the line numbers in the code where the changed code can be found.
- Justify the change — i.e. link the change to your adherence to an item in our coding standards or give full details why the change was needed.
3.6 Working together or assisting a classmate

When students are required to work in teams or pairs, the team is expected to submit only one code base for assessment. The names and student numbers of all participants should appear in the preamble. The same requirements apply as those for submitting an individual task.

If students had worked together, all the students who were involved should acknowledge one another’s involvement in their preambles. In this case, an extended declaration is required. The declaration should stipulate the level of co-operation and describe the participation of each of the students involved.

If an assignment is meant to be completed individually, working in teams or pairs may be a problem, especially when the contribution of the individuals who participate is not equal. It is, however, not prohibited. We encourage our students to help one another. The value of co-operative learning is widely accepted [19, 40, 46]. Rohani [36] reports that students may have a feeling of greater competence when instructed by a peer than when instructed by a teacher. Moreover, pair programming is acknowledged to have the potential to enhance learning [27, 12].

If you received assistance while working on an individual assignment, you are required to state this in your declaration in the preamble.

3.7 A final note

Common errors

It is a common misconception among students that, as long as code similarity checks are unable to identify their code as potentially plagiarised, it has not been plagiarised. Students with this misconception are likely to find ways to fool their assessors. These students do not realise that these actions are not only unethical but also detrimental to their own learning and final success.

Survival kit

The essence of academic integrity is to consider the purpose of a practical assignment when tackling the assignment. Complete the assignment with the intention of achieving the learning goals of the assignment. Always try to solve the given problems in a way that has the best potential to provide the practical experience needed for gaining a deeper understanding of the content being taught at the time when the assignment is given.
Illustrate your understanding and honesty when submitting your code for assessment by acknowledging the resources you used, and describing your own contribution to the code base that you submit for assessment.

Put academic integrity in your pocket

It is important to eradicate plagiarism in some way or other because it undermines learning\(^7\). Shaw et al. [39] are convinced that it is better to prevent cheating than to spend the effort needed to establish that cheating has occurred and take action against students who did indeed plagiarise. Prevention entails reducing the opportunities to cheat, minimising the temptations to cheat as well as fulfilling our educational responsibility to create a learning culture and instil ethical values. Kourie [22] emphasises that curricula should be aimed at transmitting an agreed-upon value system, using skills training as the context to achieve this. Lecturers should constantly reinforce values relating to professionalism, responsibility, ethics, etc. by making good use of opportunities to assimilate these values. The core message that should be brought to our students is aptly described in the words of Scott Alexander:

\[
\text{All good is hard. All evil is easy. Dying, losing, cheating, and mediocrity is easy. Stay away from easy.} \quad \text{\cite{Alexander}}
\]

\(^7\)http://www.cs.utsa.edu/~wagner/pubs/plagiarism.html

\(^8\)http://en.thinkexist.com/quotations/all-good-is-hard-all-evil-is-easy-dying-losing/324937.html
Chapter 4

Installing Linux

4.1 Introduction

An operating system (OS) is the low-level software that supports a device’s basic functions, such as scheduling tasks and controlling peripherals. Widely used OSs are Windows, Linux, Mac OS and Android. Users generally interact only with application software (apps) which in turn uses the functionality of the OS to perform the required operations. As a programmer, you are sometimes required to interact directly with the OS by means of a set of commands. A part of the OS, called the command processor or command line interpreter, accepts and executes the commands to perform such operations. Programmers use the command interpreter of the OS to manage their files and to perform basic tasks, such as compiling and testing their programs. Some of the typical operations you may want to perform include moving files, copying files, changing the names of files, compiling a program and executing a program.

Linux is the preferred OS for our introductory programming module. The first Linux kernel was released in October 1991 by the Finnish software engineer, Linus Torvalds\(^1\) from whose name the word Linux was derived (Linus + Unix = Linux). Arch Linux is installed on the computers in the labs on campus. We use this environment in the lectures to show practical examples. This provides a simple environment which is suitable for introducing novice programmers to the process of software development. You are advised to create a similar programming environment on your own computer so that you can do your programming assignments at home in an environment similar to the environment in the labs on campus.

\(^1\)https://en.wikipedia.org/wiki/Linus_Torvalds
This chapter introduces the Linux OS and describes the different options for installing a Linux environment on your own computer. You may choose which one of these options will best suit your situation. The chapter also introduces the basic OS commands you will use while learning to program.

### 4.2 Linux distributions

Many organisations, companies and communities are developing a multitude of different Linux systems. These systems are called “distributions” or “distros” and are all built on top of the original Linux kernel. The different distributions handle internal operations differently and are often developed for a specific purpose. Here we briefly discuss only five popular distros.

**Ubuntu**

Ubuntu\(^2\) is a popular desktop and server distribution derived from Debian, maintained by the British company Canonical Ltd. It is designed for inexperienced Linux desktop users. The latest release is 15.10 (nicknamed Wily Werewolf) and its next build 16.04 will be the last to be based on the Unity 7 desktop. It includes tools for browsing the web, accessing e-mails, playing multimedia files and working with office documents. It also has a large software repository with a wide variety of apps.

**Lubuntu**

Lubuntu\(^3\) is a lightweight variant of Ubuntu. You can use it on older machines or systems with limited resources. Version 10.10 of Lubuntu uses the LXDE window manager and requires a minimum of 128 MB of RAM and a Pentium II processor. It focuses on being fast and energy efficient. It features a plethora of office, internet, multimedia and graphics apps along with a wide assortment of useful tools and utilities. Lubuntu is compatible with Ubuntu repositories, giving users access to thousands of additional packages which can be easily installed.

**Arch Linux**

Arch Linux\(^4\) is a rolling release distribution targeted at experienced Linux users and maintained by a volunteer community. It works well even on

\(^2\)http://www.ubuntu.com
\(^3\)http://lubuntu.net/
\(^4\)https://www.archlinux.org/
minimal and old hardware, provided that you do not install a resource-hungry desktop environment. Install a lightweight desktop environment such as LDXE. If you are brave, you may even use it without a desktop environment. The Arch Linux installer deploys only the bare minimum and relies on the user to install manually what he/she wants. It requires some skill to add features and to customise it, but once it is set up it is the easiest to maintain. Owing to its upgrade policy, it always has the latest kernels in their repos days or weeks ahead of other distros.

OpenSUSE
OpenSUSE\(^5\) is a community distribution, sponsored mainly by the German company SUSE. The latest version – OpenSUSE LEAF 42.1 – takes its version number from Douglas Adams’s answer to “life, the universe and everything”. Its all-in-one management tool YaST (Yet another Setup Tool) can handle software installation as well as system configuration. It uses the KDE desktop and offers GNOME as an alternative desktop.

Fedora
Fedora\(^6\) is a community distribution sponsored by the American company Red Hat. It aims to be a technology testbed for Red Hat’s commercial Linux offering, where new open-source software is prototyped, developed and tested in a communal setting, before maturing into Red Hat Enterprise Linux. The distribution RedHat is intended to serve the engineering environment, specialising in computation-intensive operations.

4.3 Install options

You can create your own environment for writing, compiling and testing your programs written in C++ in several ways. Each of these ways has its advantages and disadvantages. Consider these when deciding which option will work best for you.

Native operating system

Install Linux as the native OS on your computer. When you apply this option it is like starting from scratch with a new computer. The new OS will delete the current OS and replace the current apps and data on your

\(^{5}\)http://www.opensuse.org
\(^{6}\)http://fedoraproject.org
computer. After installing, you can copy the backup of your data, which you have made before installation, to your new machine. You will also have to re-install the Linux versions all your apps on your new machine. Some apps, such as MS Office, do not have Linux versions. You will have to find alternative apps instead. Note that replacement apps might be unable to read your old data files unless you have used your old apps to save them in a format that the destination apps can read.

If you have a computer with limited resources, you might not have enough disk space to have more than one OS installed, so this may be your only option. In such case, you have to select your primary hard drive as the destination for installing Linux. Select the option for replacing your old OS or erasing the entire disk in the installer wizard.

**Dual boot**

Install Linux along with another OS and use a dual-boot boot-loader. The lab machines are set up in this way. When you apply this option, your computer can load the OS of your choice every time you boot your computer. The installer will give you the option to indicate that you want to install Linux alongside your old OS. It will automatically install a boot-loader which supports dual-boot and will require you to configure the boot-loader. You have to have multiple physical hard drives for this option. Alternatively, you can create multiple partitions on one physical hard drive. Each partition appears to be a separate device – called virtual hard drives. If the installer does not offer to partition the hard drive, you should do it before installation, using the tools of your current OS. When installing Linux, select an empty physical or virtual hard drive as the destination for installing Linux.

When selecting this option, you are advised to set up a shared drive where you can save and read data using either of the OSs running on your machine. You will find instructions on how to do this on the Arch Linux wiki\(^7\) and many other online forums.

**Virtual machine**

Install Linux on a virtual machine on top of the native OS on your computer. A virtual machine (VM) is an emulation of a real computer inside another OS. You first have to install a VM host app. Depending on what native OS is installed, you can choose from several VM apps. We recommend that you use VirtualBox\(^8\), which is cross-platform free open-source software that can be used on any OS to host VMs.

\(^7\)https://wiki.archlinux.org/index.php/Samba#Adding.a.Share.to.fstab

\(^8\)https://www.virtualbox.org
After installing a VM app, you can use it to create a new virtual machine. The VM app should have a wizard, which will take you through the steps to create a VM and install the OS of your choice on it. Allocate about a quarter of your physical RAM or less to the VM. VirtualBox includes a wizard to install Ubuntu. If you want to install a different distro, you should consult the distribution’s website or other relevant online forums.

When you choose this option, Linux runs like an app on your computer. This is probably the most convenient way of having a functional Linux while keeping your old OS, apps and data intact. Unfortunately, it is only viable if you have enough RAM and hard disk space. It is not recommended that you install a VM if you have less than 1 GiB RAM.

When selecting this option, you should set up the VM to use the peripheral resources of the host, such as the keyboard, mouse, printer and WiFi connection. For convenience, you may set up a shared drive where you can save and read data by using the file manager of the VM as well as the host’s file manager.

**Portable device**

Create a bootable USB stick (flash drive) for running Linux. You can use an app such as Universal-USB-Installer.exe. It is a wizard which allows you to select the OS of your choice so that you can install the OS on a specified flash drive. Because the operations this program must perform are potentially harmful, it pops up as a User Account Control (UAC) prompt. You have to approve at the prompt before the wizard will guide you through the installation process. It creates a bootable flash drive that will boot into a fully capable Linux installation. Any changes you make will persist the next time you boot it up.

When selecting this option, you should follow the procedure to boot from the flash drive every time you want to use it. The procedure for booting from something other than your primary hard disk, is described in the section discussing how to boot from installation media when wanting to launch a Linux installation. You can customise your Linux installation to your liking. With this option, you are advised to mount a directory of the host computer in the Linux you keep on your flash drive. This will allow you to transfer the work you have done to other devices, if necessary.

**Fake it**

Since you can write and compile C++ programs on any OS, it is not essential to have Linux so that you can learn to program in C++ or do the assignments required in our introductory programming module.
When using MacOS, you may use a C++ compiler such as the one included in Xcode. You may use the Xcode editor or any simple text editor. You can also get SciTE for your Mac, although this is not free of charge. Since MacOS is Unix-based, the command line interpreter uses the same commands as Linux.

When using Windows, you are advised to install the MinGW\(^9\) C++ compiler. It will behave in almost the same way as the GCC compiler on Linux. You will also be able to install SciTE\(^10\). You can use it to edit and save your programs in the same way as you would in Linux. You can use the Windows command prompt (in the Windows System Menu) in the same way as you would use the terminal in Linux to compile your program. The compiler will create an .exe file, which you can execute simply by typing its name. Note that the Windows command prompt does not use the same commands as Linux. If you wish, you can install an app such as Cygwin\(^11\) to provide a command window which supports Linux commands.

If you choose this option, you should compile your homework assignments in Linux before submitting them for assessment. Furthermore, it is recommended that you should spend enough time working in the labs to gain enough experience in using the native Linux environment so that you will be proficient when you have to complete your practical exam on Linux.

### 4.4 Installing the OS

If you want to install Linux, you have to acquire an installation disk for the distro of your choice. You will find a link on the distribution’s website, which you can use to download the appropriate ISO file matching your hardware and current OS. An ISO file has .iso as extension. It is a copy of a disk image. You have to burn it on a bootable DVD or CD or copy it to a bootable flash drive. If your OS does not have the tools to do this, you can download an app to do it. Some free apps to burn CDs include Free ISO Burn\(^12\) and ImgBurn \(^13\). We recommend UNetbootin\(^14\) or Linux Live USB Creator\(^15\) for making a bootable flash drive containing the ISO file.

---

\(^9\) http://www.mingw.org/wiki/howto_install_the_mingw_gcc_compiler_suite

\(^10\) http://www.scintilla.org/SciTE.html

\(^11\) https://www.cygwin.com/

\(^12\) http://www.freeisoburner.com

\(^13\) http://www.imgburn.com

\(^14\) http://unetbootin.sourceforge.net

\(^15\) http://www.linuxliveusb.com
4.5. INSTALLING SOFTWARE

To install Linux, you have to boot from the installation medium i.e. the DVD, CD or flash drive containing the ISO file you downloaded. To do this, switch off the computer, insert the CD/DVD or the USB flash drive and switch the computer back on. When the first screen appears, you have to press your boot option key. Most computers display a message to inform you which key or key combination you should press to invoke the boot options. If not, read your computer’s manual (or google) to find it. If successful, you should see a small screen listing the drives you currently have available. Select the installation medium.

You may also enter the BIOS options from the start-up screen. Here you can specify that the machine should always boot from the external medium. Look at the top of the screen for a menu which relates to booting. This menu should give an option such as “Boot priority”, “Boot order” or something similar. You can change the setting to list your installation medium first. Save the settings and exit BIOS. If you use BIOS instead of the boot menu, make sure you change your boot options back to its previous state (probably booting from your hard drive) once you have finished the installation. The installer wizard will start automatically. Answer the prompts according to your choice of how you want to install Linux.

We do not give you a step-by-step guide for installing the software you need. You are advised to get the information for the specific distro you are installing fresh from the Internet. Apart from the comprehensive instructions that are usually provided on the distribution’s website, you will find many forums where you can ask questions and share your experience with installing and configuring your environment. Take note of the following general advice when installing Linux:

- Always make a backup of your system before starting any installation. Although some of the options are not supposed to destroy any of your data, it is better to be safe than sorry.
- If you have a laptop, plug in the charger. If the laptop loses power during the installation, it might corrupt your entire drive.
- Some distributions automatically download updates during installation. If you do not want to waste bandwidth, unplug your network cable and do not connect your WiFi during the installation.

4.5 Installing software

Each Linux distro has its own tools to support the installation of apps. You can find the information about how to do the installation of the apps you need, on the distribution’s website.
The Arch Linux package manager, called pacman\textsuperscript{16}, is an app for installing and removing apps. It allows you to manage packages from the official repositories or from your own builds. If you want to install a single package or list of packages, give a command that complies with the following template at the command prompt:

\begin{verbatim}
pacman -S <package name 1> <package name 2> ...
\end{verbatim}

To use Linux for programming with C++, as we do in our introductory programming module, you need the following:

- A command line interpreter – Arch Linux comes with Bash
- A desktop environment – we recommend LXDE
- A web browser – for Arch Linux we recommend Firefox
- A text editor – we recommend SciTE
- A compiler – we recommend GCC

If you installed the Arch Linux distro, give the following commands at the command prompt to install the required software.

\begin{verbatim}
sudo pacman -S gamin LXDE
sudo pacman -S firefox firefox-AdBlock-plus
sudo pacman -S scite
sudo pacman -S gcc base-devel
\end{verbatim}

In each case follow the prompts and answer them carefully.

After you have installed LXDE you will have to edit the \texttt{.xinitrc} file to activate the environment. You will find commented instructions in this file for loading different window managers, for example \texttt{# exec startkde}. Insert or uncomment the instruction \texttt{exec startlxde}. Find more help on configuring LXDE in the HowtoForge Linux Tutorial\textsuperscript{17}.

SciTE reads its configuration settings from \texttt{~/.SciTEUser.properties} (user-specific) and \texttt{/usr/share/scite/SciTEGlobal.properties} (global). You can access both of these files from the options menu in SciTE and change them to your liking. The Frequently Asked Questions (FAQ) page\textsuperscript{18} has tips you may find useful.

\textsuperscript{16}https://wiki.archlinux.org/index.php/Pacman
\textsuperscript{17}https://www.howtoforge.com/tutorial/arch-linux-lxde-desktop/
\textsuperscript{18}http://www.scintilla.org/SciTEFAQ.html
4.6 A final note

Common errors

A universal problem people encounter when having to install software is a lack of confidence. If you are prepared and understand the issues, the installation software is likely to guide you gently through the process – just do it!

The converse may also be a stumbling block. If you are over-confident you may be inattentive while going through the motions. This may be disastrous. Remember the prompts in the wizards are there for a reason. If the default was suitable in all cases, there would be no need to ask the user to specify which option has to be applied.

Survival kit

If you keep to the following advice, you will probably install and configure your software with the least possible blunders.

- Read about the software you want to install and learn about its quirks before diving in.
- Before you install a program, back up everything you do not want to risk losing.
- Use the provided installation wizard carefully. Read every prompt and respond only when you really understand the consequences of each option.
- Do not ignore alert messages during the installation. Investigate the consequences and take appropriate action.
- Keep a record of the actions you have taken along with any relevant notes or observations – this may come in handy when troubleshooting later.

Put installation of software in your pocket

You can minimise the risk of installing software wrongly by knowing about the software you want to install and about the system on which you have to install it. But it is not enough to know where you are going and how you can get there is. You have to give yourself the safety net of being able to roll back when something goes wrong. There is no guarantee that nothing will go wrong, even if it says so on the box. For this reason, you should be in the habit of keeping recent backups of everything important.
Being able to install software is the easiest way to impress the non-geeks out there. It is like baking a cake with pre-mixed ingredients. You do not have to be a top-notch baker to do this, but you may look like one if you follow the instructions carefully and succeed.
Chapter 5

Operating system utilities

5.1 Introduction

In this chapter we discuss the Linux OS utilities, which programmers often use. Commands to perform such operations are accepted and executed by a part of the OS called the command processor or command line interpreter. Programmers use the command interpreter to manage their files and perform basic tasks, such as compiling and testing their programs. The command interpreter is also called the terminal or console. As you become more familiar with programming, you may develop an appreciation for the simplicity and power of using it.

Figure 5.1: The terminal
Press `ctrl + alt + t` or click on the `icon` on the desktop to open the terminal. A window similar to the one shown in Figure 5.1 should open. The text shown in the terminal window is called the command prompt. It complies with the following template:

```
<username>@<host computer>:<directory> $
```

`<username>` indicates the user who is currently logged in. On the lab computer, it is your student number. `<host computer>` indicates the name of the current computer where you are sitting. `<directory>` indicates the current directory you have opened. By default, it is `~` which is your home directory.

To give a command, you have to type the command in the terminal and press the Enter key. Note that Linux terminal commands are case-sensitive. In general, Linux commands are all lower-case letters.

Documents on your disk are organised into directories and files. A file is simply a container holding the data created by an app, such as a word processor or a text editor. You will save the source code of your programs in text files. A directory, often also called a folder, is a container that holds files and other directories.

### 5.2 Directories and navigation

#### The `mkdir` command

Use the `mkdir` (short for make directory) command to create a new directory. The command has the following format:

```
mkdir <name of directory to create>
```

For example, use the following command to create a directory called `modules`:

```
mkdir modules
```

When giving the command, the specified directory is created in the current working directory. If you want to create a directory in a different place from the current working directory, you may specify the path where the new directory should be created. For example, if your current working directory contains a directory called `modules`, you can create a subdirectory called `WTW114` in the `modules` directory by giving the following command:

```
mkdir modules/WTW114
```

Note the naming rule that directory and file names are case-sensitive and should not have any spaces.
The cd command

Use the cd (short for change directory) command to change the current working directory to the path specified in the parameter. This command has the following format:

\[ cd \textless\text{name of directory to make working directory}\r\n\]

For example, if your current working directory contains a directory called modules, you can create a subdirectory called COS101 in the modules directory by giving the following commands:

\begin{verbatim}
  cd modules
  mkdir COS101
\end{verbatim}

After completing the above examples, the directory tree structure can be visualised as:

\begin{verbatim}
  ~
  \|-- modules
  \|   \|-- COS101
  \|   \|-- WTW114
\end{verbatim}

The command prompt shows the current working directory. Before executing the cd command it was the home directory (~). After executing the cd command, the current working directory displayed in the command prompt has changed to modules instead of ~.

The cd command is used to navigate in the directory tree and the command prompt shows the current working directory. When specifying a directory, you may use the following special characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Current working directory</td>
</tr>
<tr>
<td>..</td>
<td>Parent of the current working directory</td>
</tr>
<tr>
<td>~</td>
<td>Root directory</td>
</tr>
<tr>
<td>–</td>
<td>Previous working directory</td>
</tr>
</tbody>
</table>

Note that the command interpreter will not recognise the command if you omit the space between the cd command and its argument (the dots). For example, if you type cd .. without the space between cd and the dots, the command interpreter will interpret the string as a whole as a command and produce a message such as the following:

\begin{verbatim}
  -bash: cd..: command not found
\end{verbatim}

You can correct this error by typing cd .. instead. Note the space between cd and the dots.
As an illustrative example, assume that the working directory is the directory called COS101 in the above existing structure and you want to add more directories to create the following structure:

To do this, you can give the following commands:

```bash
mkdir practical1 practical2 practical3
cd practical1
mkdir task1
mkdir task2
cd ..\practical2
mkdir task1
cd ~
mkdir webPages
```

### 5.3 Files and their contents

#### The `more` command

Use the `more` command to display the contents of a file. When executed, it will display the contents of the specified file, one page at a time. The command has the following format:

`more <name of file>`

While the contents of the file are being displayed, use the following keys for navigation:

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>Proceed to the next page</td>
</tr>
<tr>
<td>↑</td>
<td>Move up</td>
</tr>
<tr>
<td>↓</td>
<td>Move down</td>
</tr>
<tr>
<td>q</td>
<td>Quit</td>
</tr>
</tbody>
</table>
The `ls` command

Use the `ls` (short for list) command to list the contents of a directory. As an illustrative example, assume that the working directory is the directory called `practical2` and it contains the files and directories shown here:

```
practical2
    └── Prac2.pdf
        ├── task1
        │    └── income.cpp
        │    └── income.out
        │    └── income.tgz
        ├── task2
        │    └── convert.cpp
        │    └── convert.out
        └── task3
            └── rainfall.cpp
```

The following example shows how the output of `ls` command is displayed on the screen if the command is given without arguments:

```
cstallmann@b04Spc182:Practical2$ ls
Prac2.pdf task1 task2 task3
cstallmann@b04Spc182:Practical2$
```

Note that the command is typed next to the command prompt and that this prompt is displayed after the output of the command.

If you want to show the contents of a subdirectory, you can pass the directory name as argument. The following example shows how this will be displayed on the screen:

```
cstallmann@b04Spc182:Practical2$ ls task2
convert.cpp convert.out
```

If you would like to show the contents using one entry per line, you can pass the numeric digit “one” as argument. The following example shows how this will be displayed on the screen:

```
cstallmann@b04Spc182:Practical2$ ls -l
Prac2.pdf
  task1
  task2
  task3
cstallmann@b04Spc182:Practical2$
```
Pass the -R flag as argument to the ls command to list the contents of all the subdirectories. R is an abbreviation for recursive, which means that the command calls itself and passes its output as arguments to these recursive calls. The following example shows how this will be displayed on the screen:

```
cstallmann@b04Spc182:Practical2$ ls -R
Prac2.pdf  task1  task2  task3
./task1:  
income.cpp  income.out  income.tgz
./task2:  
convert.cpp  convert.out
./task3:
rainfall.cpp
```

If you would like to show more information about the files and directories, you can pass the -l flag as argument. The following example shows how this will be displayed on the screen:

```
cstallmann@b04Spc182:Practical2$ cd task1
cstallmann@b04Spc182:task1$ ls -l
-rw-r--r-- 1 cstallmann root 473 Feb 8 2015 income.cpp
-rw-r--r-- 1 cstallmann root 2923 Feb 8 2015 income.out
-rw-r--r-- 1 cstallmann root 625 Feb 8 2015 income.tgz
```

The number in the fifth column indicates the size of the file in bytes. If a file is empty, the size is zero. Other information which is also shown includes read and write permissions and the date and time that the file was last modified.

**The cp command**

Use the cp (short for copy) command to make copies of files or directory structures. The command has the following format:

```
cp <name of file to copy>  <name of the copy>
```

The following command creates a copy of income.cpp. The copy is called incomeNew.cpp:

```
 cp income.cpp incomeNew.cpp
```
As is the case when using the `cd` and `mkdir` commands, you may include path names for both the source and the target when making copies across directories. For example, give the following commands while the current working directory is `task1` to create a new directory called `task4` at the same level as `task1` and make a copy of `income.cpp` called `insurance.cpp` in the new directory:

```
mkdir ../task4
cp income.cpp ../task4/insurance.cpp
```

You can use the `cp` command to copy directories and their contents by passing the `-R` flag as argument. For example, give the following commands to change the current working directory to `practical2` and then create a new directory called `task5` that is a copy of `task2`:

```
cd ..
cp -R task2 task5
```

The following is a visualisation of the directory called `practical2` after making the above copies:

```
practical2
  └── Prac2.pdf
      ├── task1
      │   ├── income.cpp
      │   ├── income.out
      │   └── income.tgz
      │       └── incomeNew.cpp
      └── task2
          └── convert.cpp
          └── convert.out
      └── task3
          └── rainfall.cpp
      └── task4
          └── insurance.cpp
      └── task5
          ├── convert.cpp
          └── convert.out
```

**The `rm` command**

Use the `rm` (short for `remove`) command to delete files and directories. The command has the following format:

```
rm <name of file to delete>
```
Be careful when you use this command, because deleted files cannot be recovered. They are not automatically moved to a recycle bin as they would have been if you deleted the files by using the GUI. The file called `income.cpp` can be deleted by giving the following command:

```
rm income.cpp
```

You may include path names when specifying the files to be deleted when you use the `rm` command. For example, give the following command while the current working directory is `task1` to delete the file called `convert.out` in the `task5` directory:

```
rm ../task5/convert.out
```

You can use the `rm` command to delete directories and their contents by passing the `-R` flag as argument. For example, assume the current working directory is `practical2` in our running example. Give the following command to delete the directory called `task3` and its contents:

```
rm -R task3
```

### The `mv` command

Use the `mv` (short for `move`) command to move files or directory structures. The command has mutatis mutandis\(^1\) the same format as the `cp` command. It can be used in the same way and it performs the same operations. The only difference is that when using `mv` instead of `cp`, the source is deleted after the copy has been made. Because of the similarity of these two commands, we do not give detailed examples of how to use the `mv` command.

You can use the `mv` command to rename a file. Give the following command to change the name of the file called `task3.cpp` to `incomeTax.cpp`:

```
mv task3.cpp incomeTax.cpp
```

### 5.4 Wild card characters

When specifying file names or paths, use the `*`-character as a substitute for unspecified characters. Any name that has zero or more characters instead of the asterisk matches this pattern. Similarly, use the `?`-character as a substitute for one unspecified character. Ranges of characters enclosed in square brackets ([ and ]) serve as a substitute for any of the characters in the specified ranges; for example, any single capitalised or lower-case letter can be substituted where the pattern `[A-Za-z]` is required.

\(^1\)Mutatis mutandis is a Latin phrase meaning “the necessary changes having been made.”
The following example shows how these substitutes can be used to write
generic commands. This command lists only the .cpp files in the current
directory, which have a lower-case or upper-case k as the third character in
the file name:

```bash
ls ??[kK]*.cpp
```

The use of naming conventions, for example starting a specific subset of
filenames with the same prefix, allows you to refer to this subset of files in
terms of a wild-card pattern.

### 5.5 Serialising and compression

#### The tar command

The tar (short for tape archive) file format was created to write hierarchical
file structures to sequential I/O devices on an early main-frame OS called
UNIX. A file in tar format is not a compressed file. It is the serialisation
of files and their structure into a single file. Linux kernel developers ported
the tar utility to Linux. The command has the following format:

```
tar -c <files to be archived> -f <target file name>
```

The tar command can be executed in one of several modes. The mode
of execution is specified by using the appropriate flag. The following table
shows a selection of flags and the modes they specify:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Create</td>
</tr>
<tr>
<td>r</td>
<td>Append</td>
</tr>
<tr>
<td>u</td>
<td>Update</td>
</tr>
<tr>
<td>x</td>
<td>Extract</td>
</tr>
</tbody>
</table>

#### Create a tar archive

The -c and -f flags stand for create and for target file respectively. The
following command concatenates the source files called Account.cpp, Ac-
count.h and main.cpp into a single target file called Bank.tar. The -f flag
must be followed by a space and the filename of the target file.

```
tar -c Account.cpp Account.h main.cpp -f Bank.tar
```

The following three commands are valid variations of the above command:

```
tar -f Bank.tar -c Account.cpp Account.h main.cpp
```
```
tar -f Bank.tar Account.cpp Account.h main.cpp -c
```
```
tar -cf Bank.tar Account.cpp Account.h main.cpp
```

Note that the flags may be presented in any order, provided that their applicable parameters are specified properly. Note how the flags may be concatenated as shown in the last variation. It is assumed that these files reside in the directory where the command is given. If this is not the case, the filenames may be specified with their complete paths. Likewise, the archive in the above command is created in the current directory, but could equally well be created in any valid location specified in the command.

Wild cards (see Section 5.4) may be used to specify the source files for the `tar` command. For example, the following command will create a new target archive called `Project.tar` that contains all files with `.cpp` or `.h` extensions in the current directory:

```
tar -c *.*cpp -*.*h -f Project.tar
```

When using wild cards in a `tar` command, it is advisable to add the `-v` flag. This flag means verbose. This will make the command give a list of the files which are processed. For example, the following command will create exactly the same archive as the above command, but it will list the files on the standard output.

```
tar -cv *.*cpp -*.*h -f Project.tar
```

### Append or update a tar archive

The `tar` command is executed in append mode when passing the `-r` flag instead of the `-c` flag as argument. The following command adds the source files called `Customer.cpp` and `Customer.h` to the existing target archive called `Bank.tar` which was created in the previous example.

```
tar -r Customer.cpp Customer.h -f Bank.tar
```

The `tar` command is executed in update mode when passing the `update` flag. This mode is used when you want to replace a file in an archive with a newer version of that file without changing other files in the archive. For example the following command will alter the existing target archive called `Bank.tar` to contain the newer version of the source file, called `Customer.cpp`. If the original target did not contain a file called `Customer.cpp`, this file will be added:

```
tar -u Customer.cpp -f Bank.tar
```
Extract a tar archive

You might expect that there is a utility such as `untar` to extract a tar archive, but there is no such command. Instead, the extract mode of the tar utility is used to extract the target file. For example, the following command will verbosely extract the entire archive called *Project.tar* in the current directory:

```
tar -xvf Project.tar
```

If the current directory already contains files with the same names as those in the archive, they will be overwritten without a warning. For this reason, it is important for you to move or copy the versions of these files that you would like to keep, to other directories before extracting the archive.

One can selectively extract specified files from an archive. For example the following command will only extract the header files in the given archive:

```
tar -xf archive.tar *.h
```

Compress a tar archive

The a compression utility called `gzip` was created by Jean-Loup Gailly and Mark Adler and publicly released on October 31, 1992\(^3\). The g in gzip stands for “gratis”, meaning free. The gzip utility compresses a single file. One important thing to remember about gzip is that, unlike tar, it replaces your original file with a compressed version.

You can create a compressed file bundle by combining `tar` and `gzip`. You use `tar` first to create one file which is a serialisation of the files. Then you apply `gzip` to compress it. The compressed tarfile with the extension `.tar.gz` or `.tgz` created by combining `tar` and `gzip`, is usually called a tarball.

The first step is to use `tar` to combine the collection of files into a single uncompressed file. The second step is to use `gzip` to compress the resulting single file. You can do this by giving commands such as the following in succession:

```
tar -cv *.cpp *.h -f Project.tar
gzip Project.tar
```

The first of the above two commands creates a tar archive, consisting of all the `.cpp` and `.h` files in the current directory, and the second command uses `gzip` to compress this archive. The result will be a file called `Project.tar.gz` which is a compressed tar file containing the specified files.

\(^3\)http://en.wikipedia.org/w/index.php?title=Gzip&oldid=369888563
CHAPTER 5. OPERATING SYSTEM UTILITIES

The `tar` command can be executed with the `-z` flag (short for gzip). This performs both these steps in a single command. The following command will create the same resulting `.tar.gz` file as the above-mentioned two commands would have:

```
tar -czv *.*.cpp *.h -f Project.tar.gz
```

**Extract a compressed tar archive**

You have to extract a tarball to gain access to the files inside the tarball. Similar to when creating a tarball, the user can perform the operation in two steps by first unzipping the file with the `gunzip` command and then extracting it. For example, use the following commands to decompress and then extract the tarball called `Project.tar.gz`.

```
gunzip Project.tar.gz
```

```
tar -xvf Project.tar
```

You can perform both these actions in a single command by adding the `z` switch when extracting with the `tar` utility. For example:

```
tar -xzvf Project.tar.gz
```

### 5.6 Become efficient

**Read the man pages**

Use the `man` (short for *man*ual) command to display brief documentation about the use of OS commands and other applications installed in Linux. These manuals are called the man pages. You can close a man page by pressing the `q` key. The `man` command has the following format:

```
man <name of command>
```

For example, use the following command to find more information about the `ls` command:

```
man ls
```

When reading the man pages of the commands we have discussed here, you will realise that we have only touched the surface. You should read the man pages of the commands you use to refresh your memory about how to use them and to find more powerful ways to use them.

---

4 Pronounce as *gee-unzip*
Apply asynchronous execution

Asynchronous execution means having multiple windows open at the same time. While you are programming, you frequently have to move between typing code, compiling the saved code and executing the compiled code. You should have the tools for all these actions available at the same time and be able to switch between them with ease.

If an app is invoked from the command line, that app usually has to be closed before the command interpreter accepts more commands. This behaviour can be changed by starting the app asynchronously. When doing so, the app will be executed in the background and the command interpreter will be active immediately after it has been invoked. The command has the following format:

```
<command> &
```

For example, use the following command to start SciTE asynchronously:

```
scite &
```

Use hot keys

A hot key is a key or a combination of keys that gives quick access to a particular function in a computer program. It is worthwhile memorising certain hot keys for actions you have to perform regularly. The use of hot keys helps you to perform certain tasks more quickly, allowing you to do more in less time. Table 5.1 lists a few hot keys you may find useful.

The `tab` key can be used to complete your current command. Type a few characters and then press the `tab` key. If unambiguous, the command will automatically be completed. If the terminal is unable to auto-complete, simply press the `tab` key again to list all possibilities and select the one you want.

The `!` key can be used to recall a previous command. Type `!` followed by one or more of the starting characters of a previous command. The first event whose starting characters match the given string, when searching all previous commands from the latest to the oldest, will be executed. This will save time because you do not have to scroll back with the up arrow if the command you want is far back in history.

You will find a practical selection of Linux shortcuts and commands in a concise form in the `Linux Newbie Administrators Guide`.

[^http://lnag.sourceforge.net/lnag.html/node5.html]
### Table 5.1: Hot keys

<table>
<thead>
<tr>
<th>Hot key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctrl + alt + arrow</td>
<td>Go to desktop (direction)</td>
</tr>
<tr>
<td>alt + f4</td>
<td>Close window</td>
</tr>
<tr>
<td>alt + space</td>
<td>Window menu</td>
</tr>
<tr>
<td>alt + tab</td>
<td>Next window</td>
</tr>
<tr>
<td>alt + shift + tab</td>
<td>Previous window</td>
</tr>
<tr>
<td>f11</td>
<td>Toggle full-screen mode</td>
</tr>
<tr>
<td>ctrl + alt + t</td>
<td>Open terminal</td>
</tr>
<tr>
<td>Ctrl + ℥</td>
<td>Clear the terminal screen</td>
</tr>
<tr>
<td>tab</td>
<td>Auto-complete command in the terminal</td>
</tr>
<tr>
<td>!X</td>
<td>Execute the last command you ran that started with X</td>
</tr>
<tr>
<td>up arrow</td>
<td>Recall the previously used command in the terminal</td>
</tr>
<tr>
<td>Esc + .</td>
<td>Insert the last argument of the previous command</td>
</tr>
</tbody>
</table>

### 5.7 A final note

#### Common errors

It is common to make typing errors when typing commands. Because the commands are cryptic and they have to adhere exactly to the required syntax, a simple space or capitalisation difference may result in an invalid, or a very different command. Keep the following in mind to avoid typing errors of this kind:

- If a command has an argument, there should be a space between the command and the argument. A classic example is forgetting the space between the `cd` command and its argument if the argument is `~` or `..` or something similar.

- If a command has more than one argument, there should be a space (but not a comma) between each two arguments. It may be difficult at first to avoid typing commas to separate arguments.

- Commands and command options are case-sensitive.

- Options should be preceded by the hyphen character. Do not separate the hyphen and the option by inserting a space character.

Students who are new to the command interpreter often find it difficult to understand the concept of a working directory and to visualise the
tree structure of directories and files. It may help to draw the directory hierarchy on paper and to place a physical token, such as a coin, on the working directory in the drawing. Every time you use the `cd` command, move the token to its new position. Once you have done this for a while, you may find it easier to maintain a mental picture of your tree structure of directories and files while you are working.

**Survival kit**

Memorise the names of the commands you need, and use the man pages to find out which options and arguments you should use. Table 5.2 is a summary of the commands discussed in this chapter.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mkdir</code></td>
<td>Make directory</td>
</tr>
<tr>
<td><code>cd</code></td>
<td>Change directory</td>
</tr>
<tr>
<td><code>ls</code></td>
<td>List files and directories</td>
</tr>
<tr>
<td><code>more</code></td>
<td>Show the contents of a file</td>
</tr>
<tr>
<td><code>cp</code></td>
<td>Copy files and directories</td>
</tr>
<tr>
<td><code>rm</code></td>
<td>Remove files and directories</td>
</tr>
<tr>
<td><code>mv</code></td>
<td>Move files and directories</td>
</tr>
<tr>
<td><code>tar</code></td>
<td>Serialise and extract a file structure</td>
</tr>
<tr>
<td><code>gzip</code></td>
<td>Compress a file</td>
</tr>
<tr>
<td><code>gunzip</code></td>
<td>Decompress a compressed file</td>
</tr>
<tr>
<td><code>man</code></td>
<td>Show documentation of a command</td>
</tr>
</tbody>
</table>

**Put OS commands in your pocket**

Most OS operations can be performed using graphical user interfaces (GUIs). For example you click the icon for the program to start a program. Many editors have a menu option to compile source code. When using the command interpreter, you have to type the name of the program you want to execute and will probably have to specify a number of arguments. Using the command interpreter might seem to be less efficient than using a GUI. You may ask why it is necessary to know how to use the command line? The answer is that with practice it is often faster to type a few commands in the terminal than to perform a series of clicks.
Expert computer users tend to rely heavily on using command line commands, probably because many how-to instructions on the web are given as command line commands. This in turn is probably done because it is much easier to give and explain a command than to describe how to find the menu options and configure the dialogues in GUIs.

Once you have discovered the flexibility and power of the command interpreter, you will grow fond of it. The learning curve is steep but you will soon see it is worth the effort. You may soon have a better grasp of the more advanced uses of the command interpreter, which will open a world of opportunities to automate many boring and repetitive tasks. This chapter discusses only the most basic commands and gives a few examples. Now it is up to you to learn more about these commands and other commands on your own.
Chapter 6

Compiler utilities

6.1 Introduction

The diagram in Figure 6.1 shows the process you have to follow to complete each programming assignment. Use a text editor such as SciTE to create and edit your code, use a command interpreter such as bash to compile, test and archive your code, and use a browser such as Firefox to download assignments and upload solutions.

![Figure 6.1: The programming process](image)

In this chapter we discuss the practical tricks and tools you need to compile programs. You will have to master these tools so that you can experiment with programming examples. You will also have to use these tools when writing your own programs.
6.2 Command line compiling and linking

Compiling a single C++ file

When writing a C++ program, you have to compile the source code before it can be executed. When the code of the whole system is included in a single .cpp file, the process is trivial. You simply have to use the g++ command to invoke the GCC compiler and pass the .cpp file name as argument to the command.

If your source code file is called HelloWorld.cpp, use the following command to compile the program:

```
g++ HelloWorld.cpp
```

This command will create the executable file with the default name (a.out). Use the following command to execute this program:

```
./a.out
```

Although only a single command is given to perform the compilation of the program, the following three steps are automatically performed to obtain the final executable program:

1. **Compiler stage:** all C++ language code in the .cpp file is converted into a lower-level language called Assembly language.

2. **Assembler stage:** the assembly language code created in the previous stage is converted into object code. These are fragments of code that the OS of the computer can execute. An object code file is usually saved with .o as its file extension.

3. **Linker stage:** the final stage is to link the object code fragments with code libraries which contain “built-in” functions. This stage produces an executable program, which is called a.out by default.

You can specify a name for the output file using the `-o` flag followed by the name for the output file. If your source code file is called HelloWorld.cpp, you can compile the program with the following command:

```
g++ HelloWorld.cpp -o MyWorld.out
```

This command will create the executable file with the specified name: MyWorld.out. Use the following command to execute this program:

```
./MyWorld.out
```

Note that you can give the executable file whatever name you want. It need not have the same name as the .cpp file.
6.2. COMMAND LINE COMPILING AND LINKING

Header files

When writing code in terms of classes (using the object-oriented programming paradigm), it is customary to save the code of a given class in two files:

1. **The header file:** this file should contain forward declarations (prototypes / signatures) of classes, member functions and instance variables. A header file is usually saved with .h as its file extension.

2. **The implementation file:** this file should contain the complete definitions of the member functions which are listed in the header file. The implementation file is usually saved with .cpp as its file extension.

The code for such a class has to be combined into a single document before it can be compiled. This is done by using a pre-processor directive to insert the header file in the .cpp file before compiling the combined content of the two files. This is specified by using the `#include` pre-processor directive at an appropriate position in the implementation file.

If a header file is called `Account.h`, it can be included in any .cpp file which uses the classes, functions or variables declared in this header file, by adding the following line of code in the .cpp file:

```
#include "Account.h"
```

All the code in the header files that are included in a .cpp file is compiled at the time when the code in the .cpp file is compiled. If the file called `Account.cpp` includes the file `Account.h` with the above compiler directive, the constructed file that is compiled by the following command consists of the code contained in both these files:

```
g++ Account.cpp -o Account.out
```

Multiple header files can be included in a single .cpp file. A single header file may also be included in multiple .cpp files. Note that .cpp files should not be included in header files.

To avoid multiple declarations when including a given header file in multiple .cpp files in the same compilation, it is customary to guard the content of header files as a defined block (with a unique name) by using pre-processor directives. These directives tell the compiler to compile the inclusion only on condition that the defined block was not previously encountered during the current compilation.
The following illustrates how the contents of a header file can be guarded:

```c
#ifndef H_GUARD
#define H_GUARD

/* The forward declarations of classes, functions,
 variables, and other identifiers comes here */
#endif
```

The contents of the header file are given a unique name with the directive `#define H_GUARD`. To simplify naming, this name is usually chosen to correlate with the header file's name. The entire contents of the header file are placed in a conditional block starting with the directive `#ifndef H_GUARD` and ending with the directive `#endif`. This conditional statement specifies that the body of this statement should only be considered if the given name is not defined. Because the specified name is defined inside this block, the contents of the block will be compiled only once, namely the first time it is encountered during the compilation process.

### Compiling multiple source files

When a system becomes larger, it makes sense to divide the source code into separate easily manageable .cpp files. Even if a system consists of many files, it can be compiled by giving a single compile command.

If a system is made up of two .cpp files called `Bike.cpp` and `Tricycle.cpp` respectively and a single common .h file called `Wheel.h`, then the command to compile all, assuming `Wheel.h` is properly included in both .cpp files, is as follows:

```
g++ Bike.cpp Tricycle.cpp
```

The first two steps taken to compile these files are identical to the previous procedure when compiling a single .cpp file. In the third step, the linker stage, the two compiled files are linked to create one executable program. Because the name of the output file was not specified in this command, the output file will be called `a.out` in this case.

### Compiling without linking

The single command that performs the three stages mentioned above to create the executable program, can be separated into two commands. The one command is given to complete the compiling and assembly stages without performing the linking stage. The second command is given to complete the
linking stage. The first command should compile and assemble all .cpp files and store its result (the object code) in .o files. The second command should link the object code in these .o files to produce the executable program.

Use the -c flag with the g++ command to create the object (.o) files of the specified .cpp files. The command to compile both the above-mentioned .cpp files, without linking them, is the following:

\[ g++ -c Bike.cpp Tricycle.cpp \]

When executing this command, the compiler will stop after the assembler stage. Because the names of the output files were not specified in this command, the object code is written to disk in two .o files with default names corresponding with the names of the .cpp files.

**Linking compiled code**

The compiler can create one executable file by using any number of files that are either in .cpp format or .o format as input. Give the g++ command (without the -c flag) to do this. For example you can use the following command to link the compiled code that was created in the previous section:

\[ g++ Bike.o Tricycle.o \]

The following example shows how the compiled program in the above example can be called something other than a.out. All that is needed is to specify the desired executable file name with the -o flag. Here the name of the executable file of the above-mentioned system is GoRide.out:

\[ g++ Bike.o Tricycle.o -o GoRide.out \]

Assume you have written a system consisting of the following files:

<table>
<thead>
<tr>
<th>C++ file</th>
<th>Includes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bike.cpp</td>
<td>Wheel.h, Bike.h</td>
</tr>
<tr>
<td>Tricycle.cpp</td>
<td>Wheel.h, Tricycle.h</td>
</tr>
<tr>
<td>main.cpp</td>
<td>Bike.h, Tricycle.h</td>
</tr>
</tbody>
</table>

When giving the following commands, the intermediate .o files for this system will be created on disk; after this they are linked and a single executable file called GoRide.out\(^1\) is created.

\[ g++ -c Bike.cpp Tricycle.cpp main.cpp \]
\[ g++ Bike.o Tricycle.o main.o -o GoRide.out \]

\(^1\)The names given to files are arbitrary and should be chosen to be descriptive of their purpose.
Recompiling after a small change

If you have changed one of the source files in the above-mentioned system, it is not necessary to recompile all the files. Only the files that will be changed as a result of changing its source file will have to be recompiled. Considering how the system is designed, you will realise that Tricycle.o and main.o are not affected by changes in Bike.cpp.

If you change only Bike.cpp, you can achieve a complete rebuild of GoRide.out incorporating the change by giving the following commands:

```
g++ -c Bike.cpp
```
```
g++ Bike.o Tricycle.o main.o -o GoRide.out
```

Similarly, if you change only Tricycle.h, you will only have to recompile the files which include Tricycle.h. Use the following commands to rebuild the system:

```
g++ -c Tricycle.cpp main.cpp
```
```
g++ Bike.o Tricycle.o main.o -o GoRide.out
```

We have shown how to save compile time in situations where a complete rebuild is not needed. We do admit that in this small system it would not make much difference if you omit the creation of one or two .o files. However, when developing very large systems containing scores, or even hundreds of files, it often saves a lot of time if only the appropriate files are updated.

The compile time that is saved comes at the cost of extra work done by the programmer. The programmer has to give more commands. These commands also required some thought to get them right. Section 6.3 explains how this compile time can be saved without this cost when this work is automated.

Compiler flags

When compiling, you can give the compile command with flags. Most compilers have a host of compiler flags but they are not standardised across compilers. Table 6.1 lists some of the useful flags available when using the GCC compiler.
### 6.2. Command Line Compiling and Linking

Table 6.1: Useful g++ flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o</td>
<td>To specify the output filename.</td>
</tr>
<tr>
<td>-w</td>
<td>Disable all warning messages.</td>
</tr>
<tr>
<td></td>
<td>Note that the use of -w to suppress the compiler warnings is against</td>
</tr>
<tr>
<td></td>
<td>our coding standards, because for better reliability one should take</td>
</tr>
<tr>
<td></td>
<td>compiler warnings seriously, and not ignore them as if they do not</td>
</tr>
<tr>
<td></td>
<td>exist!</td>
</tr>
<tr>
<td>-Wall</td>
<td>Enable most compiler warnings.</td>
</tr>
<tr>
<td>-Werror</td>
<td>Treat compiler warnings as errors</td>
</tr>
<tr>
<td></td>
<td>Note that the use of this flag really shows that you are serious</td>
</tr>
<tr>
<td></td>
<td>about compiler warnings because you actually want to turn them into</td>
</tr>
<tr>
<td></td>
<td>errors.</td>
</tr>
<tr>
<td>-pedantic</td>
<td>Give the warnings demanded by ISO C++. The language extensions that</td>
</tr>
<tr>
<td></td>
<td>are supported by the GCC compiler but are not part of the ISO</td>
</tr>
<tr>
<td></td>
<td>specification of the language, are rejected.</td>
</tr>
<tr>
<td>-pedantic-errors</td>
<td>Like -pedantic, except that errors are produced instead of</td>
</tr>
<tr>
<td>-g</td>
<td>Turns on debugging. This makes your code ready to run under a</td>
</tr>
<tr>
<td></td>
<td>code inspector such as gdb.</td>
</tr>
<tr>
<td>-O</td>
<td>Turns on optimisation. You may also specify the level of</td>
</tr>
<tr>
<td></td>
<td>optimisation required, for example -O1 or -O2.</td>
</tr>
<tr>
<td>-E</td>
<td>Displays the pre-processor output on the screen (stdout).</td>
</tr>
<tr>
<td>-static</td>
<td>On systems that support dynamic linking, this prevents dynamic</td>
</tr>
<tr>
<td></td>
<td>linking with shared libraries.</td>
</tr>
<tr>
<td>-c</td>
<td>Compiles the given source down to an object file. This is used to</td>
</tr>
<tr>
<td></td>
<td>reduce the compile time of projects that consist of multiple files.</td>
</tr>
<tr>
<td>-MM</td>
<td>Displays the makefile dependencies for the given source file(s) on</td>
</tr>
<tr>
<td></td>
<td>the screen.</td>
</tr>
</tbody>
</table>
Any number of these flags can be inserted into the compile command. The following command will compile the HelloWorld.cpp program mentioned on page 64 to create an executable file called MyWorld.out. Furthermore, it will treat all warnings as errors and also include additional debugging information in this executable.

```
g++ -Werror -g HelloWorld.cpp -o MyWorld.out
```

### 6.3 Automating the build process

The `make` program was developed to automate some aspects of the compiling process. When used correctly, it only rebuilds the parts of the system whose source code has been changed since the last build.

The `make` program gets the inter-dependency of the source files in the system from a text file, usually called `Makefile` or `makefile`. The `make` program uses the data in the makefile to intelligently compile and link a system consisting of multiple C++ source files. It is the system designer’s responsibility to specify the details that the `make` program needs to compile and link the system efficiently and successfully. The `make` program checks the modification times of the files, and whenever the modification time of a file is more recent than something that depends on the file, the `make` command runs the build script accordingly.

#### Entries in a makefile

Each entry in the makefile uses the following format:

```
target: source file(s)
→ command
...
```

The word “target” in this template is the filename of the file which will be created or updated when any of its source files are modified. The list of source files, represented by the words “source file(s)” in the template, is also called the dependency list of the entry. The dependency list is a list of file names separated by spaces. The command(s) given in the subsequent line(s) is(are) executed in the given order. It is assumed that the execution of these commands in the given order creates the target file. The → in the above template represents a tab character. Each command must be preceded by →. The `make` program uses the → to distinguish between commands and dependency rules.
6.3. AUTOMATING THE BUILD PROCESS

Makefile entry to compile and link a single file

When the code of a system is located in a single .cpp, technically there is no reason to automate the process with make and use a makefile to compile it. It can be compiled by using a command line instruction, as described on page 64. You should, however, use make as early as possible in your C++ programming career. You will have to start using it sooner or later and will find it easier to use it when it really becomes necessary if it has been part of the process all along.

A single file can be compiled by using a single command line instruction. Consequently you need a makefile with a single entry to compile the file when using the make command. In this case, each of the three parts of the entry specified in the template on page 70 is trivial:

- The target is the name of the required executable that has to be created.
- The source file is the name of the file containing the source code.
- The command is the command line instruction needed to compile the program.

The makefile to compile the program in the example in on page 64 — while enabling most compiler warnings and preventing dynamic linking — consists of the following single entry:

```
MyWorld.out: HelloWorld.cpp
  g++ -Wall -static HelloWorld.cpp -o MyWorld.out
```

Makefile entries for selective compilation

The make program should be used if a system consists of multiple files. The input data to the make program (the makefile) serves a dual purpose. Firstly, it specifies all the commands that have to be executed to build the system and ultimately create its executable as well as the order in which they should be executed. Secondly, it provides the information needed to apply selective compilation as described on page 68.

When executed, the make program uses the entries in the makefile to determine which command(s) should be given to update the system. The commands that are executed are determined by the files that have changed since the last build. If Bike.cpp, in the example described on page 67, is changed it becomes newer than Bike.o which depends on it. The make program must then give a command to create a new Bike.o.
The information about when Bike.o should be updated, and what command should be executed to update Bike.o, is contained in the following makefile entry:

```
Bike.o: Bike.cpp Bike.h Wheel.h
      g++ -c Bike.cpp
```

The files on which GoRide.out depend, and the command to be given when any of these files are updated, are specified in the following entry:

```
GoRide.out: Bike.o Tricycle.o main.o
       g++ Bike.o Tricycle.o main.o -o GoRide.out
```

The first of the above entries states that whenever Bike.o is older than Bike.cpp, Bike.h or Wheel.h, the command g++ -c Bike.cpp should be given. As a result of executing this command, Bike.o will be more recent than all the other files in the system. This will trigger make to update all the targets that are dependent on Bike.o according to the entries in the makefile. Consequently the command of the second of the two above-mentioned makefile entries will be executed to create an updated version of GoRide.out. The following is a complete listing of an example makefile for the above-mentioned system:

```
GoRide.out: Bike.o Tricycle.o main.o
       g++ Bike.o Tricycle.o main.o -o GoRide.out

Bike.o: Bike.cpp Bike.h Wheel.h
       g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
       g++ -c Tricycle.cpp

main.o: main.cpp Bike.h Tricycle.h
       g++ -c main.cpp
```

Note that the order in which the commands are executed (if executed at all) is not determined by the order in which they are listed in the makefile. Usually the make program is invoked to create a single specified target, and the order of the dependency list of that target determines the order in which the required commands are executed. When the make program is invoked with the above data to create GoRide.out, it will start by inspecting the dependency list of GoRide.out. If one or more of the items in this dependency list of the current target are missing or outdated, it will find the instructions to build them in subsequent entries and execute them in the order that they are arranged in the dependency list.
Using the makefile with make

Once you have created your makefile and your corresponding source files, you are ready to use make. If you have called your makefile either Makefile or makefile, make will recognise it and use it if you give the following command at the command prompt:

```
make
```

It is actually necessary for you to specify an entry point in the makefile, which is the name of the target you want to create. The default entry point is the top entry in the makefile. Therefore you can avoid having to specify the target by making the top entry the instruction to create the final executable. You can use the make program to create any specific target by passing the required target as an argument to the make command as the entry point. Use the following command to create Tricycle.o using the above-mentioned makefile:

```
make Tricycle.o
```

You do not have to give your makefile a specific name – you may give it any name you desire. If you do not use the default name, you have to specify the name of the makefile by using the -f flag when giving the make command. If you called your makefile MyMakeData, use the following command to specify that the make command must use this file instead:

```
make -f MyMakeData
```

The following example is a command that specifies an entry point as well as a custom makefile:

```
make Tricycle.o -f MyMakeData
```

Dependency lists

To create the .o files of a system containing a large number of files, it is important to include in its dependency list all the files on which each .o depends. A particular .o file depends on its own .cpp file and .h file as well as all the .h files that are directly or indirectly included in the .cpp file.

If you are given the system that contains the classes shown in the UML class diagram of Figure 6.2, and the C++ definition of each of the classes in the diagram is stored in an .h file with the same name as the class, then the makefile entry to create Student.o should be:

```
Student.o: Student.cpp Student.h Borrower.h Loan.h Date.h Book.h
g++ -c Student.cpp
```
The dependency list of a given target is used to specify the files that are needed to create the target. The dependency list is also used to specify the order in which the files should be considered. For this reason, the order in which the files are listed in a dependency list is important. It does not make sense to use wild card characters (see Section 5.4) in dependency lists. The order is not determined if files are specified with wild card characters. Furthermore, the use of wild card characters defeats the purpose of the dependency list in terms of selective compilation because the wild cards might match files that need not be considered, resulting in some unnecessary compilation.

**Breaking long lines**

Sometimes entries in a makefile tend to be rather wide and consequently difficult to read and maintain. You should break long lines into more lines. A backslash (\) at the end of a line indicates that the next line should be interpreted as the continuation of the current line. The following entry is equivalent to the wide entry on page 73.

```
Student.o: Student.cpp\n    Student.h\n    Borrower.h\n    Loan.h\n    Book.h\n    Date.h
    g++ -c Student.cpp
```
It is important to have a new line character immediately after the backslash that indicates that the current line continues on the next line. The continuation line may start at the margin, or may be indented using any white space characters. It is more readable if you use indentation on continuation lines.

**Linking order**

When you have to link a number of .o files, it is important to list them in the correct order for the link command. The link instruction will create them in the order in which they are listed. As a general rule, if A depends on B, then B should be listed before A.

If fileA.cpp directly or indirectly includes fileB.h, then fileB.o should appear before fileA.o in the list of .o files in the link command. If we assume that a C++ file containing the main function, called LibSys.cpp, uses the classes shown in the UML class diagram of Figure 6.2, the following makefile entry will link the .o files to create an executable file called LibSys:

```
LibSys: Book.o Date.o Loan.o Borrower.o Student.o Lecturer.o
      g++ Book.o Date.o Loan.o Borrower.o
           Student.o Lecturer.o LibSys.cpp -o LibSys
```

Note that Book.o and Date.o may appear in any order because they are independent of any of the files of the system and also independent of each other. They both have to be listed before Loan.o because Loan.o depends on them. Likewise, Loan.o should come before Borrower.o, which should in turn come before Student.o and Lecturer.o. The last two objects are independent of each other and may therefore be listed in any order as long as they come after all the other files.

Using wild card characters (See Section 5.4) in a linking instruction may lead to unwanted side-effects because the order in which files are listed in a link command is significant. When the make process fails because the linking order is wrong, this is one of the most difficult errors to find.

**Comments**

It is good practice to include comments in your code to explain the code to readers. As explained in Section 2.3, comments are included in code to clarify code and give additional information that cannot be included in the code. You should avoid redundancy, however. It is important to use comments sparingly. Do not write comments which contain information that is already clear in the code. The rules for comments in code also applies to the entries in a makefile.
The text preceded by the # character in makefiles is treated as comments. The make program ignores this text. The following listing of our example makefile includes some comments:

```
# Linking the object code of the complete system:
GoRide.out: Bike.o Tricycle.o main.o
    g++ Bike.o Tricycle.o main.o -o GoRide.out

# Commands for partial compilation of c++ source files:
Bike.o: Bike.cpp Bike.h Wheel.h
    g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
    g++ -c Tricycle.cpp

main.o: main.cpp Bike.h Tricycle.h
    g++ -c main.cpp

# Custom commands:
run: GoRide.out
    ./GoRide.out

clean:
    rm -f GoRide.out *.o # deleting the executable, the .o files and backups.
```

The last two entries are custom commands. Custom commands are discussed in the next section. As shown in the last line of this example, comments may appear on the same line as a command provided that they are written after the command.

**Custom commands**

If you frequently use commands, you can define them as custom commands in your makefile. Do this by specifying the command as a target. It may be without any dependencies. These are called a pseudo targets. List the command(s) that you would like to execute when you invoke the custom command below the pseudo target. Precede each command with the usual tab character.

The following shows the format for such custom command:

```
target: optional dependencies
command
command
...
```
6.3. AUTOMATING THE BUILD PROCESS

The clean command shown in the above example is a custom command often used in makefiles. This command used wild card characters to delete all interim files and other redundant files from the current directory.

It is sometimes useful to force a full compilation the next time you give the make command. This may be needed when the system is recompiled on a different OS or a different distro or version of the OS.

The following is a typical definition of a custom command called clean:

```
clean:
    rm -f *.o *
```

This command uses the `rm` command to delete files (see Section 5.3). The `*` is a wildcard character. The `*.o` pattern specifies that all files with the `.o` extensions should be deleted. In the same way the `*~` pattern indicates that all files with names ending with `~` are also included in the list of files to be deleted. These are typical automatic backup files. The `-f` flag suppresses the error message that might be generated by the `rm` command, for example when this command is given although no such file exists.

Custom commands are executed by passing the required command as an argument to the `make` command. Give the following command to execute the above clean command:

```
make clean
```

When you recompile after a change and see a message such as the following, it is because the `make` utility did not observe the changes:

```
make: `GoRide.out' is up to date.
```

The message states that your target is up to date, though you know it is not. You can force it to compile by giving the above-mentioned clean command before the make command. You can, however, avoid this by ensuring that the file dependencies are correct in the makefile.

```
If you find that you often need to use this clean command, you have probably not included some of the dependencies in your makefile.
```

Custom commands enable you to add more steps to the compilation process, for example if you have an application called dingamaging that takes a `.dig` file to generates a `.cpp` file, the step to generate the `.cpp` file can also be included in the makefile.

Suppose you were using this fictitious application when developing the bicycle example started on page 67. The source code will then be in two files called Bike.dig and Tricycle.dig respectively. You will have to apply dingamaging to these files to create Bike.cpp and Tricycle.cpp.
The following is a complete listing of an example makefile for this system if it has been developed with the help of a third-party application called dingamaging:

```
GoRide.out: Bike.o Tricycle.o main.o
    g++ Bike.o Tricycle.o main.o -o GoRide.out

Bike.o: Bike.cpp Bike.h Wheel.h
    g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
    g++ -c Tricycle.cpp

main.o: main.cpp Bike.h Tricycle.h
    g++ -c main.cpp

Bike.cpp: Bike.dig
    dingamaging Bike.dig

Tricycle.cpp: Tricycle.dig
    dingamaging Tricycle.dig
```

6.4 Reducing the size of a makefile

Introduction

The make program has many features. Two of the powerful features it offers are the ability to define macros and the ability to specify generic rules by using some built-in macros. These features combined with the use of wild card characters (see page 54) enable the programmer to write concise makefiles.

Macros

Macros in makefiles are like string variables. They hold string constants which can be substituted anywhere in the makefile. The following is the definition of a macro for the list of object files in the running example:

```
OBJECTS = Bike.o Tricycle.o main.o
```

The programmer chooses the names that are given to macros. You should select macro names carefully. When selecting macro names you should apply the same rules as you would for variables in your programs. Most importantly, they should describe what they represent.
As is customary with our coding standards, we treat macros similarly to named constants in C++ programs by using ALL_CAPS when defining them.

The `make` program automatically expands a macro when it runs. Whenever the macro name appears inside round brackets and is preceded by a dollar sign, it will be replaced by its defined content. The following is a listing of our sample makefile. This time it uses the above-mentioned macro.

```plaintext
OBJECTS = Bike.o Tricycle.o main.o

# Linking the object code of the complete system:
GoRide.out: $(OBJECTS)
g++ $(OBJECTS) -o GoRide.out

# Commands for partial compilation of c++ source files:
Bike.o: Bike.cpp Bike.h Wheel.h
g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
g++ -c Tricycle.cpp

main.o: main.cpp Bike.h Tricycle.h
g++ -c main.cpp
```

### Special macros

In addition to the macros that you can create yourself, the `make` program uses a few built-in macros internally. Some of them are listed below:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>The current compiler (defaults to cc).</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>Special flags which are added to the built-in compile rule.</td>
</tr>
<tr>
<td>$@</td>
<td>Full name of the current target.</td>
</tr>
<tr>
<td>$?</td>
<td>A list of outdated files in the current dependency list.</td>
</tr>
<tr>
<td>$&lt;</td>
<td>The source file of the current (single) dependency.</td>
</tr>
</tbody>
</table>

You may use these macros when writing makefile entries.

### Rules

The real power of makefiles is seen when rules are used. In addition to the wild card characters explained in Section 5.4, the percentage sign (%) character is introduced. Its meaning is similar to the meaning of the * wild
card character, but it behaves differently when it is expanded. When * is used in a template, all the files matching the *-pattern are included in a single structure that adheres to the template. By contrast, when % is used in a template, a separate structure which adheres to the template is created for each instance matching the %-pattern. The following is an example of a rule which specifies that any .o file in the current directory depends on its corresponding .cpp file and can be created by compiling the .cpp file with the -c flag:

% .o: %.cpp  
g++ $< -Wall -c -o $@

The specification is that this compiling has to be done with warnings enabled. $< expands to the first dependency (a .cpp source file). $@ expands to the target (the corresponding .o file). This single rule can be used instead of a specific rule for each of the object files in the system. In our final version of our example makefile below, we use this rule:

```bash
CC = g++  
CFLAGS = -Wall -Werror  
LFLAGS = -static  
TARGET = GoRide.out  
OBJECTS = Bike.o Tricycle.o main.o

# Linking all the object code:  
all: $(OBJECTS)  
  $(CC) $(LFLAGS) $(OBJECTS) -o $(TARGET)

# Dependencies:  
Bike.o: Bike.h Wheel.h  
Tricycle.o: Tricycle.h Wheel.h  
main.o: Bike.h Tricycle.h

# Compilation rule:  
\%.o: \%.cpp  
  $(CC) \$< $(CFLAGS) -c -o $@

# Custom commands:  
clean:  
  rm -f $(TARGET) $(OBJ) *~ # deleting executable, .o's and backups
run: $(TARGET)  
  ./$(TARGET) # executing the program
```

We have expanded our use of macros. We have redefined some of the pre-defined macros, for example CC and CFLAGS, to contain details
specific to our needs. We defined a macro, called LFLAGS, containing a
different flag set, to be used when linking the object files. We included two
custom commands which are independent of any other files (they have no
dependency lists). One command removes redundant files and the other
executes the program.

The details needed in the dependency lists were reduced by using the
compilation rule. Because the rule specifies that each .o file is dependent
on its corresponding .cpp file, we no longer have to include the .cpp files
in the dependency lists. We only need to list all the .h files that are directly
or indirectly included in each of the .cpp files. The compilation rule also
specifies how the .o files can be created. Therefore, this detail is not needed
where the dependencies are listed. The make program will know when
to create a specific .o file through the dependency list without the command,
and will know how to create it through the specified compilation rule. This
generic makefile can now easily be modified to make it applicable to a
specific system. When applied to a new system, you only have to update
the dependencies and the macros.

6.5 A final note

Common errors

Syntax errors

- Not putting a tab character at the beginning of a command. As a
  result the command does not run.

- Putting a tab character at the beginning of a blank line. Then the make
  program complains that there is a “blank” command.

- Not pressing return just after the backslash. If the character directly
  before a newline is not a backslash, the text on the next line is not
  interpreted as being a continuation of the previous line. It is like not
  having a continuation character at all.

Errors in dependency lists

- Not including all dependencies. An .o file is dependent on its .cpp file
  as well as the header files that are directly or indirectly included in
  this .cpp file.
Mistakes in ordering

- Listing the files in a link command in a wrong order. If Aaa.cpp directly or indirectly includes Bbb.h, then Bbb.o should appear before Aaa.o in a compile or link command which includes these .o files in its source file list.

Survival kit

Example makefile for a system consisting of a single cpp file

Type the following makefile instruction in a text editor and save it as a file called makefile:

\[
<\text{executable-filename}> : <\text{cpp-filename}>
\]
\[
\rightarrow \text{g++ -static } <\text{cpp-filename}> \ -o \ <\text{executable-filename}>
\]

NOTE: The → is the tab character. Example:

\[
\text{Task3.out: Task3.cpp}
\]
\[
\rightarrow \text{g++ -static Task3.cpp } -o \ \text{Task3.out}
\]

Example makefile for a system consisting of multiple files

Create a makefile with \( n + 1 \) entries where \( n \) is the number of cpp files in your system. For each of the cpp files in your system, include a compile instruction in the makefile to compile it to an object file. This instruction should comply with the following template.

\[
<\text{object-file}> : <\text{cpp-file}> \ <\text{h-file1}> \ <\text{h-file2}> \ldots \ <\text{h-file-x}>
\]
\[
\rightarrow \ \text{g++ -c } <\text{cpp-file}> \ -o \ <\text{object-file}>
\]

Note that the h-files that should be listed are all the h-files that are directly or indirectly included in the specific cpp file. For example:

\[
\text{MyTools.o: MyTools.cpp MyData.h MyTools.h}
\]
\[
\rightarrow \ \text{g++ -c MyTools.cpp } -o \ \text{MyTools.o}
\]

Include a link instruction in the makefile to link all the object files to an executable. This link instruction should be the first instruction in the makefile, i.e. it should be inserted at the top. This instruction should comply with the following template:

\[
<\text{executable-file}> : <\text{o-file1}> \ <\text{o-file2}> \ldots \ <\text{o-file-n}>
\]
\[
\rightarrow \ \text{g++ } <\text{o-file1}> \ <\text{o-file2}> \ldots \ <\text{o-file-n}> \ -o \ <\text{executable-file}>
\]
6.5. A FINAL NOTE

It is important to list the o-files in the link instruction in an order which takes into account the dependencies of these files. If FileA.cpp is dependent on FileX.h, then FileX.o should be listed before (to the left of) FileA.o in the link instruction. For example:

```
zombieapocalypse: mydata.o mytools.o main.o
```

```bash
g++ mydata.o mytools.o main.o -o zombieapocalypse.out
```

**Put compiling and makefiles in your pocket**

The above survival summary will be enough to get you through most of your assignments. You are, however, strongly advised to work through this entire chapter and to read more about the GNU compiler and makefiles in manuals and other resources which you can buy in bookshops or find on the Internet. You can access the manuals of the make program and the GCC compiler by typing the following commands in the console:

```
man make
man gcc
```

When studying an example, you should verify that you understand the purpose and consequence of every command and every makefile entry. The best way to do this is to replicate the example in practice. Experiment with the running example by creating files called Bike.cpp, Bike.h, Tricycle.cpp and Tricycle.h as well as an h-file file called Wheel.h. The h-files should contain the prototypes of a few functions and the cpp-files should contain the implementations of these functions. Create a file called main.cpp containing a main function which instantiates objects which call the functions defined in the .h files. Now you can see for yourself how this system is compiled by means of the given commands.

Now that you have discovered how make can be applied to automate a wide variety of tasks, you should be able to use its capabilities successfully to manage the build process of the systems you may encounter in your future studies and future career.

---

2 Press q to quit the manual
Chapter 7

Software testing

7.1 Introduction

Any human activity, no matter how carefully it is done, is bound to involve mistakes. This applies particularly to software design, implementation and usage. Rettig [35] remarks:

Errors are more common, more pervasive, and more troublesome in software than with other technologies.

Testing is important throughout the software development life cycle, from its inception to its deployment and maintenance. The art of software testing is described in textbooks such as those by Black [2], Homès [15] and Kaner et al. [20], aimed at the professional tester. We discuss white box testing, black box testing and unit testing at a level applicable to novice programmers.

When testing a program, you should execute the program with different input values and observe the behaviour and output of the program. A set of input values where each variable in the program assumes one specified value is called a test case. If a program has only one variable, a test case will be one value. A test suite is a set of test cases for testing an entire program. The design of effective test suites is a discipline in its own right.

When designing the test suite for testing a program, the aim is to be as exhaustive as possible. Ideally a test suite should include every combination of inputs the program is supposed to handle and ensure that every possible execution path is executed. Such a test suite will allow the tester to verify that the program produces the expected output. It is not feasible in practice to design a comprehensive test suite. Testing is only one of the methods used for evaluating the correctness and reliability of programs. It should
be noted that, even with the best testing strategies, a system can never be guaranteed to be bug-free. Dijkstra [5], a Dutch computer scientist and winner of the 1972 Turing Award, asserts:

*Program testing can be used to show the presence of bugs, but never to show their absence.*

Programming errors are roughly classified in two categories, namely syntax errors and logic errors. Syntax errors refer to errors caused by not complying with the syntax of the programming language that is used, whereas logic errors refer to errors that make a program produce unintended or undesired output or cause other unexpected behaviour.

**Syntax errors**

When you compile a program, the compiler will produce error messages when it is unable to interpret the code. These errors are called *syntax errors*. One small error early in the code can often cause a large number of incomprehensible error messages. It is best to try to correct the errors one by one, starting at the first reported error. Recompile after each correction. You will notice that sometimes a large number of reported errors disappear after a single correction.

The error messages that the compiler reports often refer to concepts that novices have not yet mastered. At this stage, it would be best for you to scrutinise your code, line by line, starting at the line reported in the first error message, moving upward toward the beginning of your code. When looking for syntax errors, pay special attention to possible occurrences of the following:

- Missing semi-colons.
- Mismatched brackets, braces (curly brackets) and quotation marks, i.e. having an opening item but not its corresponding closing item, and *vice versa*.
- Misspelled variable names.
- Variables that are used without being declared.
- The incorrect use of operators, especially those that are defined with a combination of characters such as `+=`, `++`, `==`, `<<` and `>>`.

Compliance with coding standards as discussed in Chapter 2 will probably reduce the occurrence of syntax errors. Programmers who write reliable code as described in Section 2.5 and code in good style as described in Section 2.2, are less likely to introduce syntax errors when writing code.
Logic errors

Logic errors are typically not picked up by the compiler. The compiler is a program which takes a program written in a high-level language as input and produces the equivalent of the program in machine language. If the given program in the high-level language contains logic errors, those errors are replicated in the resulting executable. The coding standards guidelines for ensuring that the code is reliable, as described in Section 2.5, are aimed at reducing the likelihood of logic errors in code.

Logic errors may be stumbled upon while using a program or may be discovered during the formal testing of a program. Regardless of how and when such logic errors are found, a logic error can only be fixed after the cause of the erroneous result has been identified. Often the code causing the problem is located far from the point where the program fails. One way of finding a logic error is by investigating the program behaviour when the data that leads to the incorrect result is used. Such an investigation entails viewing the values of the variables that are involved in the calculation of the result. This can help to find the point where the variables start assuming the wrong values. Once you know which statement produces the erroneous results, you may be able to fix that statement. You need to know why it produces the wrong result so that you can fix it. Figuring this out often requires intelligence as well as insight and a deep knowledge of the programming language used.

7.2 Black box testing

Black box testing is also known as functional testing. Black box testing does not consider the internal structure of the code. It focuses solely on the outputs generated in response to the selected inputs and execution conditions [24]. Equivalence partitioning and boundary value analysis are popular black box testing methods.

Brute force

Say you have to design a test suite to test a portion of a program which is written to accept a value entered by a user. The value that is entered is the weight of a person in kilograms. It is specified that the value entered should be treated as an integer. If the weight entered is below 5 kg or above 300 kg, an error should be displayed instead. Since the values to be handled lie in a small range, it is likely that the programmer will use the short data type to store the value.
One approach would be to use \textit{brute force}, i.e. to test the program by entering every possible value that can be accepted by a short variable. That will be every single value from \(-32768 (-2^{15} + 1)\) to \(32767 (2^{15} - 1)\). This would take a long time and would be pointless. For example if the program fails with input 8000, there is no reason to believe that it will not fail with input 8003. Any reasonable algorithm for entering values between 4 and 300 is likely to treat the values 8000 and 8003 in the same way.

\textbf{Equivalence classes}

A test suite need not include multiple values that are likely to be treated similarly. Therefore, the test suite can be reduced significantly by selecting only one representative value for each group of values that are likely to be treated the same. Such a group of values is called an \textit{equivalence class}. There are four equivalence classes for the above-mentioned weight-input problem. These are shown in Table 7.1.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Equivalence class} & \textbf{Range of values} \\
\hline
Invalid – negative & -32768 to -1 \\
Invalid – smaller & 0 to 3 \\
Valid & 4 to 300 \\
Invalid – larger & 301 to 32767 \\
\hline
\end{tabular}
\caption{Equivalence classes for weight input where input value is short}
\end{table}

When designing a test suite for a program, the tester has to determine the equivalence classes for each input and select a representative value from each equivalence class. When a program has more than one input, each input has its own set of equivalence classes. The set of equivalence classes for a system as a whole is the set of all possible combinations of inputs. Therefore, the total number of equivalence classes is the product of the number of classes of the individual inputs. For example if a program has two inputs where one input has four equivalence classes and the other input has three equivalence classes that are all different from the equivalence classes of the first input, the total number of equivalence classes for the system is \(4 \times 3 = 12\).

\textbf{Testing at the boundaries}

More errors in software occur at the boundaries of equivalence classes than in the middle. For example it is more likely that the program can erroneously accept 301 as a valid weight in the above example, than accepting a weight of 801 as valid. For this reason, test suites should not
only include one representative from each equivalence class, but also explicitly contain the extremes of each of the equivalence classes. A proper test suite derived from the above equivalence classes would therefore be the set \{-32768, -1, 0, 1, 2, 3, 4, 299, 300, 301, 32767\}.

When taking into account that a comprehensive test suite should contain the extremes of each of the equivalence classes, this means that this program will have 8 values from the four equivalence classes of its first input and 6 values for its second input, amounting to at least \(8 \times 6 = 48\) different test cases. In systems with many distinct inputs, the total number of system equivalence classes can become very large.

**Testing files**

When the program uses file input and output, you have to check specific things related to the files themselves to ensure that the program functions as expected.

- Are the correct variable types used for the file variables? Make sure that an input file variable is of type `ifstream` and an output file variable is of type `ofstream`.

- Is the correct variable type used for the file name? The file name should be given as a literal (name in double quotes) or as a `c-string` variable.

- When reading from a file, the data on the file has to be an acceptable type for the data type of the input variable that has to receive the value. For example if the input variable is an `int` the data on the file has to be an integer.

- Does the program properly handle data type mismatches when reading from a file?

- Is sufficient memory available to hold the file your program will read?

- Have all files been opened before use and closed after use?

- Are end-of-file conditions detected and handled correctly?

- Are I/O error conditions handled correctly?

- Does the program properly handle `File not Found` errors?
7.3 White box testing

White box testing is also known as structural testing or glass box testing. When applying white box testing, test cases are designed on the basis of the structural design of the source code. It focuses on the control flow or data flow of a program and attempts to design test suites that execute each possible execution path in the program [38]. Branch testing and condition coverage testing are popular white box testing methods.

Testing conditional statements

When testing conditional statements, one has to consider possible comparison errors. Tests that are designed to test Boolean expressions in a program are called predicate coverage tests. The following summarises some examples of the comparison errors discussed by Myers et al. [29]:

- Using incorrect Boolean expressions. Programmers frequently confuse relations such as at most, at least, greater than, not less than, and less than or equal to and are prone to mistakes when writing logical expressions involving Boolean operands such as and, or and not. Examples of a few typical mistakes are illustrated here:

  - If you want to determine whether \( i \) is between 2 and 10, the expression \( 2 < i < 10 \) is incorrect. Instead, it should be \( (2 < i) \land (i < 10) \).

  - If you want to determine whether \( i \) is greater than \( x \) or \( y \), \( i > x \lor y \) is incorrect. Instead, it should be \( (i > x) \lor (i > y) \).

  - If you want to compare three numbers for equality, \( \text{if}(a == b == c) \) does something quite different from what one would expect.

  - For expressions containing more than one Boolean operator, the assumptions about the order of evaluation and the precedence of operators are often confused. That is, if you see an expression such as \( \text{if}(a == 2) \land (b == 2) \lor (c == 3) \), do you understand whether the and or the or is performed first?

- Comparing variables that have different data types, such as comparing a character string to an address, date or number. This includes comparisons between variables of different sizes such as float with double. If so, ensure that the conversion rules are well understood.
• Comparing fractional or floating-point numbers that are represented in base-2 by the underlying machine. This is an occasional source of errors because of truncation and the base-2 approximations of base-10 numbers.

• Using invalid integer arithmetic. For instance, if \( i \) is an integer variable, the value of the expression \( 2 \cdot i / 2 == i \) depends on whether \( i \) is odd or even and whether the multiplication or division is performed first.

Testing loops

Loops apply expressions in the same way as conditional statements. Therefore the design of test suites to identify comparison errors applies similarly to the testing of loops. Furthermore, loops can have control flow errors. Here we discuss some questions which may be asked to guide the discovery of control flow errors in a loop, which Myers et al. [29] suggest.

**Will the loop terminate?** It is not easy to identify values which may make a loop iterate forever. You should argue about the loop condition. Is it guaranteed that this condition changes when the loop body executes? Is it guaranteed that this change will eventually lead to the condition becoming untrue?

**Is non-execution a problem?** Consider the conditions upon entry of the loop to verify if it is possible that the loop will not execute. If so, does this represent an oversight? For instance, what happens if \( x \) is greater than \( z \) in the following for loop?

```plaintext
for (i = x; i <= z; i++)
{
    ...
}
```

Similarly, what happens if NOTFOUND is initially false in the following loop?

```plaintext
while (NOTFOUND)
{
    ...
}
```
Are there any off-by-one errors? Investigate if the loop iterates one too many or one too few times. This is a common error in zero-based loops. You may often forget to count 0 as a number. For example, if you want to create C++ code for a loop that iterates 10 times, the following would be wrong, as it performs 11 iterations:

```cpp
for (int i = 0; i <= 10; i++)
{
    cout << i;
}
```

You should ask the above questions when designing test suites for testing loops. The test suite should contain values to cover all possibilities, particularly for counter-controlled loops. The following is a list of the possibilities suggested by Beizer [1] to assist with specifying suitable values to evaluate a loop. The selection of values for a test suite is based on the structure of the loop that has to be tested. For each of the items in the list of possibilities, sample test values to test the following loop are given to illustrate the item: `for(int x = 0, x <= 10, ++x)`

1. Non-boundary cases:
   - $p$ passes though the loop, i.e. $\text{MIN} < p < \text{MAX}$, where $\text{MIN}$ is the minimum passes through the loop and $\text{MAX}$ is the maximum passes through the loop (e.g $x = 6$ or $x = 7$).
   - Skip the loop entirely (e.g. $x = -3$ or $x = 17$).

2. Boundary cases at the beginning:
   - Skip the loop entirely (e.g $x = -1$).
   - Only one pass through the loop (e.g. $x = 0$).
   - Two passes through the loop (e.g. $x = 1$).

3. Boundary cases at the end:
   - $\text{MAX} - 1$ passes through the loop (e.g. $x = 9$).
   - $\text{MAX}$ passes through the loop (e.g. $x = 10$).
   - $\text{MAX} + 1$ passes through the loop (e.g. $x = 11$).
Testing units

Modular design can drastically reduce the number of test cases needed to test a system. With modular design, the problem is divided into subproblems which are solved separately by writing functions to solve the subproblems. This technique contributes significantly to the testability of a program because the functions can be tested in isolation.

Modular design is the cornerstone of modern software design thought [44]. It improves the overall quality, including the testability, of software [7]. The importance of modularity increases as the size of software systems grows. Many techniques and strategies have been proposed to modularise software systems in various contexts. Unit testing can be applied when a program is modular. This allows the developers and testers to test portions of a system before implementing all its functions fully.

The process of testing the individual sub-programs separately is called unit testing. Each independent unit will require far fewer test cases. Furthermore, the total number of test cases needed to test the entire system would require fewer cases where combinations across units are used. Therefore, the number will be be closer to the sum of the number of test cases for each unit instead of being their product. But you should note that the impact of different units on each other cannot be removed entirely. Therefore, test suites consisting of combinations across units will always be needed.

Testing functions

The most rudimentary technique for the modularisation of software is to divide a program into functions. When doing this, a simplified form of unit testing can be applied to test a program which consists of a main program and some functions. The functions can be tested separately. You have to write stubs and drivers to test your functions. A function stub is a function with the correct function signature but with an empty body or only a return statement returning a plausible default value. A program which is written for the sole purpose of testing a function is called a driver. A driver can be used to invoke each of the functions that have to be tested with a well-designed test suite for each function. Similarly, the main program can be tested on its own by using stubs for the functions that return the appropriate values designed to form a test suite for testing the main program.

The same techniques to test an entire small program have to be applied when testing one function. One would design a test suite to deal with the logical expressions, conditional statements and loops in the function. The test suite should be supplemented by using the function parameters as the basis for the design of equivalence classes.
Below is a list of the questions that you should ask when thinking about the correctness of functions:

- Do the number and order of parameters received by the function correspond to those in the prototype, and are the correct number of arguments sent by each call in the calling program?

- Do the attributes (e.g. data type and size) of each parameter match the attributes of each corresponding argument?

- Does the unit system of each parameter match the unit system of each corresponding argument? For example, it would be wrong if the parameter is expressed in degrees but the argument is expressed in radians.

- Does a function alter a parameter which is intended to be only an input value? Pass variables by reference only if they have to be modified in the function.

Integrating units

When a system is divided into smaller units which can be tested in isolation, the manner in which the modules are combined should be considered when testing the system. Ultimately, the order in which the modules are coded and tested is determined by the manner in which they are combined.

One approach when combining previously tested units into a whole is simply to put them all together at once and test the whole system. This is often called big bang integration.

An alternative approach is called incremental integration. When applying incremental integration, the previously tested units are added to the system framework, one at a time in an appropriate order, until the complete system has been formed. The benefit of incremental integration is that it may be more apparent where to find an error if the integration fails, because it has to be related to the module that was added last. Note that it would not necessarily mean that the error is in the last module. The error may still be in any of the modules that have already been integrated. It will, however, only be in code which is dependent on the module that was added last.

In systems which have many distinct inputs, the total number of system equivalence classes can become very large. This implies that it is often unrealistic to try to test every system-wide equivalence class. Testers can resolve this dilemma by carefully selecting a representative set of combinations, covering the equivalence classes of each input among them, but not necessarily all the combinations.
7.4 Debugging tools

After finding a bug in a program, you still have to track down the source of the error, i.e. the statement producing the unexpected result. Some integrated programming environments (IDEs) include tools that enable you to observe the values of selected variables and how they change while stepping through a program. Such a tool typically allows you to execute the program one statement at a time or to stop execution at a specified break point.

When you do not have access to debugging tools, you can inspect the values of the variables at any point in the program by adding output statements to the program at the points in the program where you want to inspect these values. Ideally, such a statement should include information about where they are in the program as well as what variables are shown, together with the current values of these variables. This method is effective to pinpoint the position in the code where variables start assuming wrong values. When applying this method, it is important to remove the extra output statements once the error has been fixed.

Most contemporary high-level programming languages, including C++ and Java, support the notion of an assert function. This function serves the purpose of the output statements described in the previous paragraph. However, there is no need to remove these statements after the bug has been found because the compiler can be instructed in a single compiler directive to ignore all assert statements.

Using assert in C++

The C++ assert function is useful for stopping program execution by means of an appropriate error message as well as finding the location in the program where the error occurred.

The function can be used in a situation where certain conditions must be met. This function is included in thecassert library. Therefore this library should be included in the program preamble so that the function can be called. The syntax to use assert is:

```c++
assert(<expression>);
```

The expression should be a logical expression. If the expression evaluates to false, the program will terminate, otherwise it will continue normally. When the assert function makes the program terminate, a message is written to the standard error device.

In the following example, the assert function is used to abort the program if the quotient function is called with the denominator being 0. If the
program is compiled and executed, the program terminates on the second call to the function, which triggers an assertion failure to signal the bug.

```cpp
/* assert example */
#include <cassert>
#include <iostream>

using namespace std;

double quotient(double numerator, double denominator) {
    assert(denominator != 0);
    return (numerator / denominator);
}

int main ()
{
    cout << "5 / 2 = " << quotient(5, 2) << endl;
    cout << "5 / 0 = " << quotient(5, 0) << endl;
    return 0;
}
```

The output of the program is as follows:

5 / 2 = 2.5
Assertion failed: (denominator != 0), file assertExample.cpp, line 8.
5 / 0 = Abort trap: 6

Although the assert function is useful during program development, it is not advisable to have active assert statements in a program which is put into use. The compilation of the assert statements in a program can be disabled by including the following pre-processor directive before the directive to include the library:

```cpp
#define NDEBUG
```

### 7.5 A final note

**Common errors**

Syntax errors are mistakes in code that prevent the compiler from producing the executable code. Usually the messages given by the compiler help to identify the error so that it can be fixed.
The following are common syntax errors:

- Missing semi-colons.
- Mismatched brackets, braces (curly brackets) and quotation marks, i.e. having an opening item but not its corresponding closing item, and *vice versa*.
- Misspelled variable names.
- Variables that are used without being declared.
- Incorrect use of operators, especially those that are defined with a combination of characters such as `+=`, `++`, `==`, `<<`, and `>>`.

The following mistakes often manifest as logical errors in programs written by novices:

- When if-statements and their corresponding else statements are not clearly specified.
- When the `=` and `==` operators are not used with their correct meaning.
- When breaks are not used correctly to alter the program flow in a switch statement.
- Off-by-one execution of loops – i.e. executing the loop one too many or one too few times. This happens if the loop counter starts or ends one value too early or too late. In an extreme case, it can be *off-by-two* if this error appears in one direction at the start and in the other direction at the end.
- Off-by-one reference to an element — i.e. the code refers to an element one index to the left or to the right of the correct element in a sequence.
- Processing beyond the limits of an array.
- Erroneous control expressions in loops, resulting in infinite loops.
- Misconceptions about the scope and persistence of variables.
- Misconceptions about the different ways in which parameters can be passed to functions.
- When the input stream is not handled correctly, especially after it has entered the fail state.
Survival kit

When dealing with compiler errors, correct the errors one by one, starting at the first reported error. Recompile after every correction. If a compiler error is not obvious, it may be better if you do not assume that the message reports the error correctly. Rather scrutinise your code line by line, starting on the line reported in the first error message, moving upwards towards the beginning of your code. Look for missing semi-colons, mismatched brackets, mismatched quotation marks, misspelled variable names, etc.

Design clever test suites to deal with logical errors. Table 7.2 proposes appropriate test strategies to test some of the most common defects.

Table 7.2: Common defects and testing strategies for discovering them

<table>
<thead>
<tr>
<th>Defect</th>
<th>Testing strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical conditions not formulated</td>
<td>Compute equivalence classes of the variables used in logical conditions</td>
</tr>
<tr>
<td>correctly</td>
<td></td>
</tr>
<tr>
<td>Loop structures not behaving correctly</td>
<td>Design test cases which execute a loop zero, exactly once, and more than once. Also test boundaries and extremes on the first and last iterations of the loop.</td>
</tr>
<tr>
<td>Off-by-one errors</td>
<td>Develop boundary tests in which you verify that the program handles data on all boundaries appropriately.</td>
</tr>
<tr>
<td>Not handling null conditions</td>
<td>Determine possible null conditions and test for them. For example, when data is read from a file, test the program with an empty file. Ideally the program should report an appropriate error message.</td>
</tr>
<tr>
<td>Overflow</td>
<td>Test the program with values that may exceed the capacity of inappropriately chosen data types.</td>
</tr>
</tbody>
</table>

Put software testing in your pocket

An essential skill in software development is the ability to design test suites which are likely to uncover logic errors. It remains a challenge to design adequate test suites. We can only rely on cleverly designed test suites that test the behaviour of a reasonable portion of the program in a reasonable variety of situations.

Becoming a good programmer and software tester comes with experience. A programmer who keeps a log of his/her errors is likely to learn from them and will gradually make fewer mistakes.
Chapter 8

Conquering technology

8.1 Introduction

The opportunities created in the introductory programming course are an essential part of conquering new technologies. Being able to program enhances your ability to use technology in your future workplace effectively. You are encouraged to seize the opportunities provided in the introductory programming course to increase your programming proficiency. With patience, dedication and perseverance, you will be able to achieve the desired level of programming proficiency and technological expertise for your future career.

This chapter does not claim to be a panacea to enable everyone to become a proficient programmer. It merely contains suggestions about the activities you can do to learn the needed concepts and gain deeper understanding. The chapter begins by providing advice on attitudes and then gives programming-specific advice. We include general advice which may be applied in different learning contexts. The article titled CS1 Students Speak: Advice for Students by Students by Hanks et al. [13] gives advice and discusses attitudes, mind-sets, self-theories and success factors relevant to different learning contexts. You might find it insightful and useful to augment the arguments offered in this chapter.

One thing we know is that how we teach has a limited bearing on the success of our students — i.e. no matter how well we teach, some students will fail. The converse is also true. Students learn in spite of their teachers — i.e. no matter how badly we teach, some students will succeed. The onus of achieving success is more on the students themselves than on the lecturers. Students have to take control of their own learning.
The following diagram shows four possible ways a student might feel after completing the course. The statements on the left may lack commitment and those at the bottom may lack competence. You can achieve the ideal state at the top right when you have reached your optimal level of competence through intrinsically motivated effort.

<table>
<thead>
<tr>
<th>Ability</th>
<th>Commitment</th>
</tr>
</thead>
<tbody>
<tr>
<td>I can do the work but fail to see its relevance for me</td>
<td>I enjoyed the work and learned a lot</td>
</tr>
<tr>
<td>I am glad this is over</td>
<td>I worked very hard but could have learned more</td>
</tr>
</tbody>
</table>

The diagram illustrates our premise that it takes both commitment and ability to be truly successful. Contrary to a commonly accepted view that one is born with a fixed ability to learn and achieve, we support a more adaptive view of ability, namely that learning makes you smarter [14]. If you consider yourself inherently smart, you may be tempted to think that you can pass the module without having to learn. This is a deplorably bad attitude. Even if you have already mastered most of the curriculum topics, you are advised to make the most of the opportunity. Learn beyond what is expected, add enjoyment to it and eventually you will reach greater heights.

If programming is not your passion, you may find it difficult to learn. This problem has been observed since the beginning of programming [6] and has remained a problem ever since [18, 48]. You should rise to the challenge. It is clear that those who work hard and have the right attitude are more likely to succeed. Adopt a positive attitude and learn for the sake of gaining knowledge and of developing skills, instead of finding ways to get the maximum marks for the least possible effort, because this attitude is more beneficial in the long term.

Students have unique learning styles. Everyone does not benefit equally from the different activities suggested in this chapter. Students are encouraged to use the activities that suit their personal learning styles in a way that maximises their own learning. Note that students' work tempo and style may vary. Some students could probably manage to spend less than the recommended time on studying, but many other students may find that they have to spend more time on studying to keep up to date. Whatever the case, you should work hard from the beginning and keep on working at the same pace until the end.
8.2 Active participation

Attend lectures

Attending the lectures has long-term rewards. You will benefit more from attending class when you prepare for lectures, participate during the lecture and review the work soon afterwards. If you start missing classes to attend to other pressing issues, you should reconsider your time management.

An additional benefit of attending class is that you learn to know the other students in the class. Besides building your knowledge and skills, being at university is also about making friends you could keep for the rest of your life. A good place to find those friends is right here in class.

Attend lab sessions

Lab sessions are an extension of the lectures. Here you are required to work on the assignments that are designed to reinforce the concepts taught in the lectures. If you do not use these facilities and services, you get less value from the investment your sponsor/parents made when paying your tuition fees. By committing to attending the lab sessions, you make it easier for yourself to work on a regular basis, as discussed in Section 8.5.

The benefits of working in the lab outweigh by far the benefits of doing your assignments at home. The computers in the lab are already configured for doing the assignments whereas your computer at home may not be. In the lab you also have the opportunity to ask for assistance from your peers and the teaching assistants so that you can overcome certain obstacles.

Participate in online discussions

Participation in online discussions has a significant positive effect on student learning [3, 13, 33]. When participating, you should adhere to Netiquette. Chapter 1 provides more information about the benefits of using discussion forums and how you can get the best out of them.

Form a study group

Small-group learning is effective in promoting greater academic achievement [41]. The effectiveness of a study group depends heavily on good leadership and management. Tips and guidelines on forming successful study groups can be found in many helpful web resources:\n
\footnote{1}{http://www.collegeatlas.org/successful-study-groups.html}
\footnote{2}{http://www.wikihow.com/Form-a-Study-Group}
Interact with your lecturers

Avoid the undesirable situation of having to meet the lecturer in person only when you want to discuss the possibility that you think you were treated unfairly. Meet your lecturer informally before class or catch the lecturer on the way back to his/her office after a lecture.

Consider making an appointment with the lecturer or a tutor to discuss tests and practical assignments after these have been assessed. Use this option sparingly because your lecturers may feel offended if you have not first used other options, such as asking questions during lectures, asking a teaching assistant during a practical lab session or participating in the discussion forum to solve your problem, instead of having a one-on-one session with the lecturer or tutor.

8.3 Read more

Those who read widely have higher intelligence and greater general knowledge than those who do not. Stanovich and Cunningham [42] state that reading generally makes you smarter and it keeps you sharp as you age. Reading for leisure is extremely beneficial, but you should also do focused reading so that you can broaden your knowledge about the subject matter of the course. Adequate and relevant reading is the foundation of all other learning activities. The other activities are easier and more effective when supported by appropriate reading. In this section, we discuss some of the items you should read.

Study guide and announcements

Knowing what is going on, where to find resources and what you are expected to do may contribute greatly to your overall success.

Textbook

The prescribed textbook is your most important resource and should be consulted regularly. Read ahead and formulate questions about anything you feel uncertain about. It is likely that these questions will be answered in the lecture before you have to ask them.

Manuals

There is a common saying: when all else fails, read the manual. This is not a good idea because a manual should never be read as a last resort. The
user manuals and technical manuals for software products often contain valuable information which might not be widely known. You may discover features and tricks that could improve your competence and efficiency as a programmer if you read the language specifications of the programming language you use, the documentation about your compiler as well as the man pages of the operating system you use. Technicians often use the acronym RTFM to express their irritation about other people’s questions or lack of knowledge about trivialities they would have known if they had read the instructions. Read more about this acronym in Wikipedia\(^3\).

**Code**

People read novels before they start writing them. Musicians read and play music before they start composing. Why should writing code be any different? You should read a lot of code before writing your own and should continue to read and evaluate code throughout your programming career. Reading code will develop your code-writing skills and may inspire you to write better code. Because there is such a huge amount of open source code, it should be easy to find code to read.

The following excerpt from an interview with Bill Gates [23] is good advice:

**Interviewer:** Is studying computer science the best way to prepare to be a programmer?

**Bill Gates:** No. the best way to prepare is to write programs, and to study great programs that other people have written. In my case, I went to the garbage cans at the Computer Science Center and I fished out listings of their operating system. You got to be willing to read other people’s code, then write your own, then have other people review your code. You’ve got to want to be in this incredible feedback loop where you get the world-class people to tell you what you’re doing wrong.

**Magazines**

Regular reading of topics not necessarily covered in the curriculum can give you an extra edge. To broaden your interest and keep up to date with

\(^3\)https://en.wikipedia.org/wiki/RTFM
the latest trends, consider subscribing to a computer magazine such as DrDobbs\textsuperscript{4}, eWeek\textsuperscript{5} or PC Magazine\textsuperscript{6}.

**Other resources**

You should consult multiple resources so that you can broaden your knowledge about the matters on which different resources agree and can form your own opinions about the matters where different resources do not agree. There are many textbooks and tutorials about C++. The prescribed textbook has a companion site which contains useful summaries and additional activities. The cplusplus\textsuperscript{7} website is worth bookmarking when learning C++.

8.4 **Write code**

The foundation activity in programming is *writing code*. You have to write a lot of code to become good at it. You should avoid repeatedly practising a limited set of skills. Rather do exercises which become increasingly more challenging and try to find neat new solutions, based on your own understanding of the problem instead of re-applying old solutions.

**Complete the assignments**

The given practical assignments are designed to provide practical exercise in the key skills needed. Ideally, you should be able to do these without assistance. Although there is some lenience about allowing students to leave out a few assignments, you should not give in to this temptation. Try to complete all the assignments and attempt to complete as many as possible of the optional assignments.

**Do the exercises at the end of each chapter**

The programming exercises at the end of each chapter in the textbook are well designed and focus on the core concepts of the chapter. These problems can usually be solved by using only the language constructs discussed in the given chapter and previous chapters. They are ideal for providing focused programming experience at the correct level.

\textsuperscript{4}http://www.drdobbs.com/
\textsuperscript{5}http://www.eweek.com/
\textsuperscript{6}http://www.pcmag.com/
\textsuperscript{7}http://www.cplusplus.com/
8.5 Productive learning

Students who stick to a steady and continuous work schedule and keep up to date are more likely to succeed than students who work sporadically or work in quick frantic bursts before crucial deadlines and tests/examinations. Zimmerman [50] notes that the actions needed for effective self-regulated learning are to plan, set goals, organise, self-monitor and self-evaluate.

Have a plan

It is crucial to manage your time. If you do not have a plan, you are likely to fall behind. If you list all your activities (including sleep, eating, family, leisure) in your schedule, you will find that you are able to enjoy leisure time without feeling guilty. Your plan will assure you that you have provided enough time for all your responsibilities without compromising your relaxation and leisure. The key to a good schedule is knowing how much you have to study and scheduling it in the time that you have available, and then sticking to the plan. This is aptly explained in WikiHow\(^8\).

Work in an environment conducive to learning

Avoid distractions. Work in a clean, organised space because it will benefit your productivity. You may find that you are more productive when you work in the library than in a coffee shop. If you study in a room where you also do other chores, first finish and clean up your chores before you sit down to study. A cluttered learning environment may clutter your mind. Read more about this on the Western Governors University’s blog\(^9\).

Develop test-writing skills

Writing tests and exams is a skill in its own right. There are many blogs and websites with useful guidelines. You may find the following hints useful:

- Plan the time you can spend on each question and allow time to re-read your answers at the end of the exam.

- Read and understand a question before trying to answer it. If a question is ambiguous, interpret it in the best way you can. In your answer, point out why you think the question is not clear and state the assumptions you made to remove the ambiguity.

\(^8\)http://www.wikihow.com/Create-a-Study-Schedule
\(^9\)http://www.wgu.edu/blogpost/improve-online-study-environment
• When you are stuck while answering a question, move on to another question. If time permits, you can come back to finish the problamatic question.

Apply self-evaluation

Use the questions at the end of each chapter in the textbook, the online quizzes as well as questions in past exam papers and tests, to assess your own progress. You can use these resources to give you an idea of the kind of questions that are likely to be asked in tests and exams. Remember, however, that it is essential to use these resources in a way that will enrich your knowledge and deepen your understanding of the work.

*M* **Memorising questions and answers constitutes shallow learning. It gives you a fleeting knowledge whereas deep learning ensures lasting knowledge.**

Use these resources for deeper learning by verifying how well you understand the concepts you have learned to date. Check your knowledge at regular intervals; for example after studying each chapter in the textbook. When attempting each of these, write down the answers from memory. After completing them, verify the correctness of your answers using memoranda, the textbook or other trusted sources. Take note of your mistakes. Knowing what you got wrong and why it was wrong can help you clear up misconceptions.

Create your own resources

It is widely acknowledged that explaining something to someone else is an effective method of learning. If you create summaries and write explanations in a style which fellow students may find useful, you will discover that your own knowledge and skills have improved. The following are some ideas about things you can create:

• Make your own notes or lecture slides.
• Make a list of programming errors and explain how each can be avoided.
• Design your own assignments to implement.
• Make a YouTube video to explain a concept.
• Create an interactive tutorial.
8.6 A final note

Common errors

Some mistakes that students are inclined to make include the following:

**Under-estimating the time and effort they need.**
You may have a false idea that you will not need to make as much of an effort as other students to master this module, especially if you already have some programming experience. Students who repeat the module are particularly likely to fall into this trap.

**Being over-optimistic about what they can achieve.**
Students who have fallen behind often take on more work than they can realistically cover in an effort to catch up with their peers. The reality is that if you could not cope with the given workload in the past, it is highly unlikely that you can cope in future with an even heavier workload.

**Having the delusion that they will have more time later.**
If you constantly find that you are telling yourself: *things will be better when this or that is over*, the bad news is that this state of affairs will probably never change because the things you did not do now while you were dealing with more pressing issues, will create the same problem later. The only way out is to reconsider and replan your time management strategies.

Survival kit

This section summarises some key strategies to avoid the above-mentioned pitfalls. These strategies should not, however, be treated as a magic recipe to enable everyone to become a proficient programmer. You have to decide for yourself how you can use these to shape your own future.

- Assess where you are, i.e. know what you do not know, set realistic goals and work towards achieving your goals.
- Work hard from the beginning and keep working at the same pace until the end.
- Go out of your way to gain maximal experience in understanding programming problems and writing your own code to solve them.
- Read a lot: read the textbook and additional resources. Also, read the code that experts have written.
Put learning how to program in your pocket

Being able to program is an important skill needed in any career, not just for programmers. In future it is likely that smart technology will take care of many things for us. A fair understanding of the workings of technology and the ability to code it in the way we want are essential so that we can use these technologies to our advantage.

It is clear that those who work hard and do a lot of practical programming are more likely to succeed than those who do not. A positive attitude and learning for the sake of gaining knowledge and developing skills, instead of finding ways to get the maximum marks for the least possible effort, will be more beneficial in the long term.

The level of success is increased when you achieve a state of mind where you feel cognitively efficient, deeply involved and highly motivated, and find enjoyment in the work. This is possible when the perceived challenges that an activity poses are in balance with your perceived abilities or skills [26]. You should therefore constantly challenge yourself so that you can achieve more advanced abilities and skills and ultimately enjoy rising to the challenges.
Bibliography


BIBLIOGRAPHY


Index

~, see home directory
*, see wild card character
?, see wild card character
\, see continuation character
#, see pre-processor directive, see comments in a makefile
$@, see macros in makefiles
$(), see macros in makefiles
$<, see macros in makefiles
$?, see macros in makefiles
%, see wild card character
&, 59, see asynchronous execution

abbreviations, see forum
ability, 100
academic integrity, 25
acknowledge resource, 31
anxious, vii
Arch Linux, 38
assert, 95
assignments, 104
asynchronous, 59
attributes of a file, 52
avoid multiple declarations, 65
bash, 44
big bang integration, 94
black box testing, 87
boot
from CD, 43
from DVD, 43
from flash drive, 43
boundary testing, 88
break point, 95
breaking long lines, 74
brute force, 87
bugs, 85, see error
camelCase, 15
cassert, 95
CD - boot from, 43
cd command, 48
           not found, 49
change directory, 48
change log, 32
change the name of a file, 54, see
          mv command
citation, 31
clarity of code, 16
clean command, 76
code
          clarity, 16
          efficiency, 22
          flexibility, 19
          reliability, 20
          reuse, 28
          robustness, 21
coding standard, 13
coding style, 14
collaboration, 34
combining units, 94
command
          cd, 48
INDEX

not found, 49
clean, 76
cp, 52
gunzip, 58
gzip, 57
ls, 50
make, 70, 72, 74
man, 58
mkdir, 48
more, 50
mv, 54
rm, 53, 77
tar, 55
untar, 57
command documentation, 58
command interpreter, 48
command line, 48
command line navigation, 48
command prompt, 48
comments, 17, 75
commitment, 100
compile, 63–66
compiler
directive, see pre-processor
directive
tar file, 55
current directory, see working directory
custom commands, 76
Debian, 38
debug, 95
debugging tools, 95
delete
directory, 54
file, 53
dependency list, see makefile
directory, 48
delete, 54
move, 54
naming rule, 48
remove, 54
tree, 48, 52
discussion board, see forum
display
content of a file, 50
directory tree, 52
distro, see Linux distro
documentation generator, 30
drama triangle, ix, x
driver, 93
drop out, xiii
dual boot, 40
DVD
boot from, 43
e-mail, 3–6
attachments, 4
body, 3
netiquette, 3
request tracking, 5
response, 5, 6
subject line, 3
efficiency of code, 22
emoticon, see forum
enthusiasm, xii
entry in a makefile, 70
INDEX

equivalence class, 88
error, 21, 86, 87
  compiler, 86
  logic, 21, 87, 97
  runtime, 21
  syntax, 86, 96
exam-writing skills, 105
execute, 64
extract
tar file, 55, 57, 58
zip file, 58
Fedora, 39
file
  attributes, 52
  compression, 57
  copy, 52
  delete, 53
  directory, 48
  execute, 64
  list, 50
  move, 54
  naming convention, 55
  naming rule, 48
  path, 48, 49, 53, 54, 56
  remove, 53
  rename, 54, see mv command
  show content, 50
  size, 52
  testing, 89
  view content, 50
firefox, 44, 63
flags, 55
flash drive, 41
  boot from, 43
flexibility of code, 19
folder, see directory
forum, 6–10, 101
  abbreviations, 10, 11
  emoticons, 10, 11
  noise, 8
  off-topic, 8, 12
reply, 8
sms language, 10, 11
thread, 7
writing style, 9, 10
function testing, 93
functional testing, 87
g++, see gcc
gcc, 44, 64, 66
glass box, see white box
go to a directory, 48
good choice, xii
guard, 65
gunzip command, 58
gzip command, 57
hacking, 26
header file, 65
home directory, 49
hot keys, 59
I’m OK – you’re OK, v, xiii
identifier names, 15, 17
include a file, 65
incremental integration, 94
indentation rules, 15
install
  Linux, 39–43
  linux packages, 44
  integrating units, 94
  integrity, xii, 25
Karpman drama triangle, x
lab session, 101
layout rules, 15
LDXE, 38, 44
lecture, 101
lecturer, 102
life position, v
link, 63, 67, 75
linking order, 75
linux, 37
distribution, 38  
dual boot, 40  
fake on MacOS, 41  
fake on Windows, 42  
install, 39–43  
install options, 39  
LDXE, 38  
native installation, 39  
on a memory stick, 41  
packages, 43  
virtual machine, 40  

**linux distro**  
Arxh Linux, 38  
Debian, 38  
Fedora, 39  
Lubuntu, 38  
OpenSUSE, 39  
Red Hat, 39  
Ubuntu, 38  

list file attributes, 52  
list files, 50  
logic error, 21, 87, 97  
loop testing, 91  
ls command, 50  
Lubuntu, 38  

MacOS, 41  
macros in makefiles, 78, 79  
make, see makefile  
make directory, 48  
makefile, 70, 73  
  clean, 76  
  dependency list, 70, 73, 81  
  entry, 70, 72, 74  
  macros, 78, 79  
man command, 58  
man pages, 58, 102  
memory stick, see flash drive  
message board, see forum  
mkdir command, 48  
modes of the tar command, 55  
modular design, 93  

more command, 50  
mov e a directory, 54  
mov e a file, 54  
mv command, 54  

naming conventions, 15, 17, 55  
navigation, 48  
NDEBUG, 96  
netiquette rules, 2  
noise, see forum  

off-topic, see forum  
OK Corral, v, vi  
online forum, see forum  
OpenSUSE, 39  
operating system, 37  
order of files in link command, 75  

packages, 44  
paranoid, vi  
parent directory, 49  
persecutor role, x  
phishing, 4  
placement of brackets, 15  
plagiarism, 29  
planning studies, see schedule  
portable linux, 41  
practical assignments, 26  
pre-processor directive, 65, 66, 95, 96  
preamble, 30  
professional, xii  
programming error, see error  
programming process, 63  
psychopath, vi  

read  
  code, 103  
  cplusplus website, 104  
  magazines, 103  
  man pages, 102  
  manuals, 102
study guide, 102
study, 102
textbook, 102
read more, 102
recursive (R flag), 52, 54
Red Hat, 39
refactoring, 27
reliability of code, 20
remove
directory, 54
file, 53
rename a file, 54, see mv command
request-tracking systems, 5
rescuer role, x
resources, 106
responding to e-mail, 5, 6
responsibility, vii, xii
reuse of code, 28
rm command, 53, 77
robust code, 21
root directory, see home directory
runtime error, 21
schedule, 105
SciTE, 44, 63
scope, 20
selective compilation, 68, 71, 74
self-evaluation, 106
serialise a directory, 55
show
ccontent of a file, 50
directory tree, 52
size of a file, 52
sms language, see forum
spam, 4
special makefile macros, 79
structural testing, see white box
stub, 93
study
exercises, 104
group, 101
resources, 102–104, 106
schedule, 105
study guide, 102
style of code, 14
suicide, vi
syntax error, 81, 86, 96
tape archive, 55
tar command, 55
tar file, 55, 58
append, 55, 56
create, 55
extract, 55, 58
update, 55, 56
tarball, see tar file
teamwork, 34
terminal, 48
test-writing skills, 105
testing, 85, 89, 93
black box, 87
boundaries, 88
brute force, 87
conditional statements, 90
equivalence class, 88
files, 89
functional, 87
functions, 93
glass box, 90
loops, 91
structural, 90
test case, 85
test suite, 85, 88
units, 93
white box, 90
thread in a discussion, see forum
Ubuntu, 38
unit, 93
defining, 93
integration, 94
testing, 93
untar, 57
unzip, 58
update
tar file, 55
USB stick, see flash drive
user friendly, 21

value system, xii
variable names, 15, 17
variable scope, 20
verbose, 56
victim role, x, xii, xiii
view
    content of a file, 50
    directory tree, 52
virtual machine, 40

white box testing, 90
wild card characters, 54, 56, 74–79
win win, vi
Windows, 42
working directory, 49
working together, 34
writing exam, 105

zip command, 57
It is axiomatic that programming (along with reading, writing and arithmetic) is a key skill one must have to be considered literate in the 21st century. It is the ability to make digital technology do whatever one wants it to do — within the possible; to bend digital technology to one’s needs. For this reason, many degree programmes recommend that an introductory programming module should be part of the curriculum.

Over the past decade, Vreda has identified the topics that students regard as hurdles preventing them from successfully mastering the core content of their introductory programming module. The assumption was previously that students know how to study, how to behave in online communication, will be able to teach themselves how to use a number of utility programs and can automatically acquire certain skills which they have to use when doing their programming assignments. The notes and guidelines in Tricks of the Trade for Novice Programmers Volume 1 can bridge the gap and help students overcome these obstacles to become more successful in their first programming course and in their studies beyond it.

**Vreda Pieterse** has been the co-ordinating lecturer of the introductory programming module at the University of Pretoria since 2009. She has over thirty years of teaching experience, covering a variety of areas in Computer Science. She holds an MSc in Computer Science, a postgraduate Diploma in Higher Education as well as a BSc Hons in Mathematics.

**Marie Pieterse** designed the book cover.