Contents

Preface vii

1 Computers and Programs **1**
- 1.1 Computers and computation . . . 2
- 1.2 Programs and programming . . . 3
- 1.3 A first C program . . . 6
- 1.4 The task of programming . . . 9
- 1.5 Be careful . . . 10
- Exercises . . . 12

2 Numbers In, Numbers Out **13**
- 2.1 Identifiers . . . 13
- 2.2 Constants and variables . . . 14
- 2.3 Operators and expressions . . . 20
- 2.4 Numbers in . . . 22
- 2.5 Numbers out . . . 23
- 2.6 Assignment statements . . . 25
- 2.7 Case study: Volume of a sphere . . . 26
- Exercises . . . 27

3 Making Choices **29**
- 3.1 Logical expressions . . . 29
- 3.2 Selection . . . 32
- 3.3 Pitfalls to watch for . . . 34
- 3.4 Case study: Calculating taxes . . . 36
- 3.5 The switch statement . . . 37
- Exercises . . . 40

4 Loops **45**
- 4.1 Controlled iteration . . . 45
- 4.2 Case study: Calculating compound interest . . . 49
- 4.3 Program layout and style . . . 50
- 4.4 Uncontrolled iteration . . . 53
- 4.5 Iterating over the input data . . . 57
- Exercises . . . 59

5 Getting Started with Functions 63
5.1 Abstraction . . . 63
5.2 Compilation with functions . . . 67
5.3 Library functions . . . 70
5.4 Generalizing the abstraction . . . 72
5.5 Recursion . . . 74
5.6 Case study: Calculating cube roots . . . 76
5.7 Testing functions and programs . . . 78
Exercises . . . 80

6 Functions and Pointers 83
6.1 The main function . . . 83
6.2 Use of void . . . 84
6.3 Scope . . . 86
6.4 Global variables . . . 87
6.5 Static variables . . . 90
6.6 Pointers and pointer operations . . . 91
6.7 Pointers as arguments . . . 94
6.8 Case study: Reading numbers . . . 96
Exercises . . . 97

7 Arrays 101
7.1 Linear collections of like objects . . . 101
7.2 Reading into an array . . . 103
7.3 Sorting an array . . . 105
7.4 Arrays and functions . . . 107
7.5 Two-dimensional arrays . . . 111
7.6 Array initializers . . . 114
7.7 Arrays and pointers . . . 115
7.8 Strings . . . 118
7.9 Case study: Distinct words . . . 123
7.10 Arrays of strings . . . 126
7.11 Program arguments . . . 127
Exercises . . . 129

8 Structures 133
8.1 Declaring structures . . . 133
8.2 Operations on structures . . . 135
8.3 Structures, pointers, and functions . . . 139
8.4 Structures and arrays . . . 142
Exercises . . . 143

9 Problem Solving Strategies **145**
9.1 Generate and test . . . 145
9.2 Divide and conquer . . . 146
9.3 Simulation . . . 151
9.4 Approximation techniques . . . 156
9.5 Solution by evolution . . . 160
Exercises . . . 161

10 Dynamic Structures **163**
10.1 Run-time arrays . . . 163
10.2 Linked structures . . . 170
10.3 Binary search trees . . . 177
10.4 Function pointers . . . 179
10.5 Case study: A polymorphic tree library . . . 183
Exercises . . . 190

11 File Operations **193**
11.1 Text files . . . 193
11.2 Binary files . . . 195
11.3 Case study: Merging multiple files . . . 199
Exercises . . . 202

12 Algorithms **203**
12.1 Measuring performance . . . 203
12.2 Dictionaries and searching . . . 206
12.3 Hashing . . . 207
12.4 Quick sort . . . 212
12.5 Heap sort . . . 218
12.6 Merge sort . . . 222
12.7 Other problems and algorithms . . . 225
Exercises . . . 226

13 Everything Else **229**
13.1 Some more C operations . . . 229
13.2 Integer representations and bit operations . . . 230
13.3 The C preprocessor . . . 235
13.4 What next? . . . 238
Exercises . . . 239

Index **241**

Preface

Since commencing this project I have been asked many times what I can possibly contribute to a market already awash with books about C and programming. My motivation has formed over more than twenty years of teaching programming to first-year university students, watching their reactions to and behavior with a range of texts, and a range of programming languages. My observations led to the following specification:

- *Length*: To be palatable to undergraduate students, and accessible when referred to, a programming text must be succinct. Books of $500+$ pages are daunting because of their sheer size, and the underlying message tends to get lost among the trees. I set myself a length target of 250 pages, and have achieved what I wanted within that limit. For the most part I have avoided terseness; but of necessity some C features have been glossed over. I don't think that matters in a first programming subject.

- *Value for money*: Students are (quite rightly) sceptical of \$$100$ books, and will often commence the semester without owning it. Then, if they do buy one, they sell it again at the end of the semester, in order to recoup their money. I sought to write a book that students would not question as a purchase, nor consider for later sale. With the cooperation of the publishers, and use of the SprintPrint format, this aim has also been met.

- *Readability*: More than anything else, I wanted to write a book that students would willingly read, and with which they would engage as active learners. The prose is intended to be informative rather than turgid, and the key points in each section have been highlighted, to allow students to quickly remind themselves of important concepts.

- *Practicality*: I didn't want to write a reference manual, containing page upon page of function descriptions and formatting options. Students learning programming for the first time instead need to be introduced to a compact core of widely-applicable techniques, and be shown a pool of examples exploring those techniques in action. The book I have ended up with contains over 100 examples, each a working C program. A number of the examples are purely illustrative, and do no useful computation, but they are outnumbered by the "real" programs that tackle real problems.

- *Context*: I wanted to do more than describe the syntax of a particular language. I also wanted to establish a context, by discussing the programming process itself, instead of presenting programs as static objects. I have also not shirked from expressing my personal opinions where appropriate – students should be encouraged to actively question the facts they are being told, to better cement their own understanding. And because many new students are not quite sure what a career in computing actually entails, I have described some of the various ways in which computing professionals interact.

- *Excitement:* Last, but certainly not least, I wanted a book that would enthuse students, and let them see some of the excitement in computing. Few things can match the elegance of quick sort, for example. Yet many of texts I have worked with fail to recognize the exhilaration that is possible – almost as if those authors have let too many years slip by since they last sat down and wrote a program.

My work over the last year has been driven by these six goals.

How to use this book

In terms of possible courses, this book is intended to be used in two slightly different ways. Students who are majoring in non-computing disciplines require C programming skills as an adjunct rather than a primary focus. Chapters 1 to 8 present the core facilities available in almost all programming languages. Chapter 9 then rounds out that treatment with a discussion of problem solving techniques, including some larger programs, to serve as models. There are also six case studies in the first nine chapters, intended to provide a basis on which the exercises at the ends of the chapters can be tackled. For a service course, use Chapters 1 to 9, and leave the more able students to read the remainder of the book on their own.

Chapters 10 to 13 delve deeper into the facilities that make C the useful tool that it is, and consider dynamic structures, files, and searching and sorting algorithms. They also include two more case studies. These four chapters should be included in a course for computer science majors, either in the initial programming subject, or, as we do at the University of Melbourne, at the beginning of their second subject.

In terms of presentation, I teach programming as a dynamic activity, and hope that you will consider doing the same. More than anything else, programmers learn by programming, in the same way that artists learn by drawing and painting. Art students also learn by observing an expert in action, and then mimicking the same techniques in their own work. They benefit by seeing the first lines drawn on a canvas, the way the paint is layered, and the manner in which the parts are balanced against each other to make a cohesive whole.

The wide availability of computers in lecture theaters has allowed the introduction of similarly dynamic lessons in computing classes. By always having the computer available, I have enormous flexibility to show the practical impact of whatever topic is being taught in that lecture. So my lectures consist of a mosaic of prepared slides on an overhead projector; pre-scripted programming examples using the computer; and a healthy dose of unscripted exploratory programming. With the live demonstrations I am able to let the students see me work with the computer exactly as I am asking them to, including making mistakes, recognizing and fixing syntax errors, puzzling over logic flaws, and halting infinite loops.

The idea is to show the students not just the end result of a programming exercise as an abstract object, but to also show them, with a running commentary, how that result is achieved. That the presentation includes the occasional dead end, judicious backtracking and redesign, and sometimes quite puzzling behavior, is all grist for the mill. The students become highly involved, and for example have at times chorused out loud at my (sometimes deliberate, sometimes inadvertent) introduction of errors. The unpredictability of this process is a wonderful way of retaining their attention. The web also helps with this style of teaching – the programs that are generated in class can be made accessible in their final form, so students are free to participate, rather than frantically copy.

Running a lecture in this way requires a non-trivial amount of confidence, both to be able to get it right, and to deal with the consequences of sometimes getting it wrong. More than once I have admitted that I need to go and read a manual before coming back to them in the next class with an explanation of some obscure behavior that we have uncovered. But the benefits of taking these risks are considerable: "what will happen if..." is a recurring theme in my lectures, and whether it is a rhetorical question from me, or an actual interjection from a student, we always go to the computer and find out.

Supervised laboratory classes should accompany the lecture presentations. Students learn the most when trying it for themselves, but need to be able to ask questions while they do. Having students work on programming projects is also helpful. But the need for assessment should be separated from the need for support and feedback. So my programming assignments are marked and returned, but count for zero marks in the final grade. That way students are free to cooperate and collaborate if they wish. I then ask at least one exam question relating directly to the project. In a sense, the students receive their project feedback shortly after completing it, and receive their project assessment a month later, when they sit their final examination. Once they understand that carrot, there is no need for a stick, and they willingly work on the assigned task.

The exercises at the end of each chapter also include broader non-programming questions, for use in discussion-based tutorial classes.

Software and teaching support

All of the program fragments in this book exist and are available for your use, as are sample answers to many of the exercises. If you are planning to make use of this book in an educational environment, please contact me (`alistair@cs.mu.oz.au`) identifying your institution, and the subject you are teaching. I will gladly reply with a complete set of programs, and a guide as to which page of the book each is from. A set of PDF lecture slides to match the book is also available on request. For obvious reasons, I do not plan to make these resources publicly available via a web page, so you do have to ask. An errata page listing known defects in the book appears at `http://www.cs.mu.oz.au/~alistair/ppsaa/errata.pdf`.

Acknowledgements

This book has been just ten months in the writing, but nearly three decades in the making. I learned programming in the first half of the 1970s as a pimply secondary school student in Wellington, New Zealand, and will always be grateful to my two maths teachers, Bob Garden and Ron Ritz, for their vision and enthusiasm. Our programming involved a dialect of Fortran, and was carried out with a bent paper clip; special preprinted cards that you popped chads out of; and a bike ride to the local bank branch to drop the completed programs into a courier bag for transmission to their "Electronic Data Processing Center". Each compile/execute cycle took about three days, so we quickly learned to be accurate.

My interest in computing was deepened during my University study, and I thank all of the Computer Science staff that worked at the University of Canterbury in New Zealand during the period 1977–1979. Worth special mention is Tadao Takaoka: in his own inimitable way, it was he who interested me in algorithms, and who served as a role model for the academic life that I have pursued for twenty years.

Since then, it has primarily been academic colleagues at the University of Canterbury and at the University of Melbourne that have influenced me, by sharing their knowledge and skills. I first taught introductory programming in 1982, and have done so every year since then. The people that I have worked with on those subjects, or on other academic projects, have all left a mark on this book. In roughly chronological order, they include: Rod Harries, Robert Biddle, Tim C. Bell, Ian Witten, Ed Morris, Rodney Topor, Liz Sonenberg, Lee Naish, Harald Søndergaard, Roy Johnston, Peter Stuckey, Tim A.H. Bell, Martin Sulzmann, and Owen de Kretser. Justin Zobel provided assistance with Section 12.3, and Bernie Pope made numerous insightful suggestions. I have also benefitted from supportive academic leadership at the Universities of Canterbury and Melbourne: John Penny, Dick Cooper, Wolfgang Kreutzer, Peter Poole, Peter Thorne, Leon Sterling, and now Rao Kotagiri, have managed the two academic departments that I have worked in.

Several first-year students at the University of Melbourne found errors for me during the second half of 2002, including Johan Chan, Tim Davis, Chin Yee Lee, Daniel Salerno, and Matt van der Peet. Peter Hawkins also pointed out a number of typos.

Finally, there is family: my parents, Duncan and Hilda Moffat, always explored knowledge and valued learning, and taught me to do the same. My children, Anne and Kate, are now moving down a similar path. Thau Mee, my wife, thinks I am just a little bit nutty for beating away at a computer late in the evening and (as it is now) at 6am in the morning, but she knows that it brings me pleasure, and she tolerates my weaknesses. She didn't write a word, but there can be no doubt that she is the second author of what you are holding in your hands.

Alistair Moffat,
Melbourne, Australia
`http://www.cs.mu.oz.au/˜alistair`

December 2, 2002