Preface

This book presents an introduction to programming using the language C++. No prior knowledge of programming is expected. However, the book is oriented towards a reader who has finished about 12 years of schooling in the Science stream. Our target readers are likely pursuing a degree in Science or Engineering, and only some will major in Computer Science.

We believe that our target reader will be able to learn programming by reading the book and solving suggested problems entirely through self-study. But the book can also be used as a text, and is detailed enough to serve as a reference.

Any programming textbook must accomplish three goals.

1. It must describe the syntax and semantics of the programming language.
2. It must explain the challenges in using the language to design programs.
3. Finally, the book must motivate the student to study for the first two goals!

The last goal requires explaining the position of programming in the world of science and engineering, and in the general world of ideas, and of course designing examples and exercises which excite the student. In the rest of this preface, we explain how we work towards these three goals.

Our approach, developed over several years of teaching programming to our target audience, has two unusual features.

1. First, the book is accompanied by a graphics package, Simplecpp. This package is useful for drawing and animating simple two-dimensional shapes. We believe this assists us in giving more vivid explanations of concepts as well as in assigning more interesting (and yet challenging) exercises to students.
2. Second, we have made a conscious effort to draw programming examples from a variety of areas, from math and science and even art. This has many advantages as will be seen shortly.

In Section 0.1, we begin by considering questions of motivation and philosophy. In Section 0.2, we outline our approach to teaching the C++ language. In Section 0.3, we discuss program design. In Section 0.4, we discuss the motivation behind our graphics package, Simplecpp.

We have also included some non-graphics features which we have found practically useful for getting students off to a quick start in writing interesting programs. This is considered in Section 0.5. Finally, in Section 0.6, we discuss how the book can be fitted into a curriculum involving one or two semesters.

0.1 The Philosophical Appeal of Computing

Science is what we understand well enough to explain to a computer.

Donald E Knuth

Why should anyone learn to write programs? A very persuasive answer is economic; programming is a skill which essentially guarantees you a job. Even if you do not plan to enter the software industry, programming is a useful auxiliary skill for many professions.

These considerations help in persuading students to study programming. But they will not necessarily make for excitement which is essential for learning. For this, it is necessary to explain why Computer Science and programming is intrinsically interesting.

It might be said that the goal of Science, or indeed of all human endeavour, is to understand the universe better. This understanding manifests itself as mental models of the various phenomena, and most such models are computational! Using a computer, you can make these mental models more explicit, and can more easily experiment with them. So computing is intimately connected to what it means to understand something. But this cannot be left as just another "nice" philosophical observation. It must be put into action.

The models you have of the world around you can be made to come alive on a computer. You have been told in Physics courses that the planets must move around the sun; but if you know programming, you can write a program to explore this claim. You can write programs to explore almost anything around you, transportation systems, chemical reactions, biological processes, social interactions. You can analyze pictures, or even literary texts, besides performing tax and salary calculations. Many of these exciting possibilities are within the grasp of our target audience. It is our experience that twelve years of school education is enough to enable students to raise questions about the world and also be able to answer some of those questions using programming skills learned in even a single course!

Thus, in this book we expect to not only teach programming, but also show how programming appears in a very diverse range of applications. We find that this appeals to our audience which doesn't intend to major in computer science, not at the beginning of the course anyway. We develop substantial programs for applications drawn from math, science, engineering, operations research, and even topics which are more like Art. Interestingly, many of these programs aptly illustrate important computer science concepts, e.g. recursion.

We thus believe that our treatment better integrates programming with the math and science skills (not to mention general worldly skills!) that the students already have. We feel this synergistically benefits the learning of computer programming and the other sciences.

Note, that by and large, we do not teach anything besides programming in this book. For building the applications we draw on what the student already knows. However, to make the book self-contained, we have included detailed descriptions of the concerned applications, highlighting the computational aspects.

The last few chapters contain somewhat advanced material, e.g. representations of graphs and their uses in representing circuits, the web graph, city maps. Also discussed are discrete event simulation, issues such as deadlocks in simulation, Dijkstra's shortest path algorithm as a simulation, a somewhat elaborate simulation of an airport, and finally the Newton-Raphson method in multiple dimensions with an application from mechanics. These are meant as help to students in doing projects and for exciting students to further study.

0.2 C++ Syntax and Semantics

This book presents the various constructs of C++ in a fairly comprehensive manner. We also present some of the more recent additions to the C++ language, such as lambda expressions. These are more convenient in many situations where previously function pointers or function objects were used. We dwell on different constructs in proportion to their importance; for example, arrays can be used in a variety of different ways: we provide examples of many such uses. In general, we attempt to provide a large number of examples for each programming construct. We also discuss some topics which might be considered "advanced", e.g. reference counting. This is discussed in Appendix G.

We have tried to present ideas in order of intellectual simplicity as well as simplicity of programming syntax. The general presentation style is: "Here is a problem which we would like to solve but we cannot using the programming primitives we have learned so far; so let us learn this new primitive". Object oriented programming is clearly important, but an attempt is made to let it evolve as a programming need. We discuss this below.

We also present a somewhat detailed overview of computer hardware. We feel that this is essential to satisfy the curiosity of our target audience, and also to make it easier for them to understand concepts such as program state, addresses and pointers, and also compilation.

C, C++ and Object Oriented Programming

The dominant paradigm in modern programming practice is clearly the object oriented paradigm. Likewise, C++ is clearly more convenient for the (experienced) programmer than C. So it could be asked: should we teach object oriented programming from "day 1"? Should we teach C++ directly or as an evolution of C?

Several educators have attempted to introduce classes and objects very early. But this is not considered easy, even by the proponents of the approach. The reasons are several. For example, for very simple programs, organizing programs into classes might be very artificial and verbose. Expecting a student to actually develop classes very early requires understanding function abstraction (for developing member functions/methods) even before control structures are understood. This can appear unmotivated and overwhelming.

Our discussion of object oriented programming can be considered to begin in Chapter 5: creating a graphical shape on the screen requires creating an object of a graphics class. In the initial chapters, it is only necessary to use classes, not build new classes. Thus, shapes can be created, and member functions invoked on them to move them around, etc.

The major discussion of classes including the modern motivations happens in Chapter 18. However, member functions are introduced in Section 17.5. Inheritance is presented in Chapter 25. Chapter 26 presents inheritance based design. It contains a detailed example in which a program developed earlier, without inheritance, is redeveloped, but this time using inheritance. This vividly shows how inheritance can help in writing reusable, extensible code.

A brief description of the use of inheritance in the design of Simplecpp graphics system is also given, along with an extension to handle composite objects.

As to the question of whether to teach C++ directly or C first, our view is pragmatic. As discussed earlier, most chapters begin with a ''crisis statement" which is followed by the resolution of the crisis in the rest of the chapter. We have attempted to order the crises in increasing order of intellectual complexity. Sometimes this causes C++ statements tobe introduced first, e.g. use of operators >> and

<< for input output rather than `scanf` and `printf`.[1] Sometimes C features get introduced first, which we then evolve to the C++ features. Thus, we present `structs` first (Chapter 17) and then generalize them to classes (Chapter 18). This enables us to gradually motivate each important idea. Likewise we discuss arrays before C++ vectors. This is because arrays are concrete and hence easier to understand. The abstraction of vectors must be studied, but it can be unconvincing at the beginning.[2]

We have a substantial discussion of the C++ standard library and template functions and template classes.

0.3 The Design of Programs

There is of course a distinction between learning the syntax and semantics of a programming language, and acquiring the ability to design programs. The former could be considered to be akin to program comprehension: someone who understands the syntax and semantics of a language should be able to work out how a program executes and predict what answers it will produce.

Design is very different. The phrase program design could mean one of two things: (a) writing a program when you reasonably well know what calculations are required to be performed, (b) first inventing the algorithm (usually requires some cleverness) and then coding it into a program. Interpretation (b) could perhaps be referred to as algorithm design. In this book, we consider algorithm design only to a small extent: specifically, we explore the use of recursion in designing algorithms. Some of the harder problems we consider here are backtrack search and structural recursion as applied to an expression drawing problem. We also discuss some analysis of algorithms, but we recognize that these are topics for later courses.

The major focus is on designing programs to solve problems, where the learner is generally conversant with how the problem is to be solved, i.e. how the problem could be solved manually using pencil and paper. There are many challenges in turning such informal knowledge into a program.

- Devising computer representations for real life/mathematical entities in the problem. Asserting invariant properties of the representations.
- Identifying the patterns in the computations that are required to be performed and expressing those computations using a programming language. This is difficult because the pattern in the computation may not directly match the primitives available in the language.
- Writing structured, extensible code. We discuss alternative ways of expressing the same logic, as well as issues relating to naming of variables, ideas such as avoiding use of global variables.
- Writing object oriented code. We have discussed this earlier.
- Reasoning about programs. We discuss ideas such as assertions, pre and post conditions, invariants, and use them in proving program correctness as appropriate.
- We discuss some elementary strategies and practices in testing and debugging.

Many of the challenges mentioned above arise naturally as we try to develop programs for problems drawn from science, math and other areas. Simple two-dimensional geometric graphics also provides fascinating programming problems, as we will discuss shortly.

[1] We do not discuss C language features such as `printf` and `scanf` which do not have any pedagogical merit. Likewise, we have omitted discussion of C language features such as unions because these are no longer relevant and are subsumed in inheritance.

[2] The later chapters of the book do use vectors and other standard classes as needed, rather than using arrays.

An important aspect of problems from science and other areas known to the student is that the student is generally familiar with the calculations that need to be performed. What is needed is deciding how to organize these calculations in general. Such problems are perfect for teaching programming without involving algorithm design.

0.4 Graphics and Simplecpp

I hear and I forget. I see and I remember. I do and I understand.

Confucius

A large part of the human experience deals with pictures and motion. Humans have evolved to have a good sense of geometry and geography, and are experts at seeing patterns in pictures and also planning motion. If this expertise can be brought into action while learning programming, it can make for a more mature interaction with the computer. It is for this reason that Simplecpp was primarily developed.

Our package Simplecpp contains a collection of graphics classes which allow simple geometrical shapes to be drawn and manipulated on the screen. Each shape on the screen can be commanded to move or rotate or scale as desired. Taking inspiration from the children's programming language Logo, each shape also has a pen, which may be used to trace a curve as the shape moves. The graphics classes enable several computational activities such as drawing interesting curves and patterns and performing animations together with computations such as collision detection. These activities are challenging and intuitive at the same time.

The graphics classes are used right from the first chapter. The introductory chapter begins with a program to draw polygons. The program statements command a turtle[3] holding a pen to trace the lines of the polygon. An immediate benefit is that this simple exercise makes the imperative aspect of programming and notions such as control flow very obvious. A more important realization, even from this very elementary session is the need to recognize patterns. A pattern in the picture often translates to an appropriate programming pattern, say iteration or recursion. Identifying and expressing patterns is a fundamental activity in programming in general. This principle is easily brought to the fore in picture drawing.

As you read along, you will see that graphics is useful for explaining many concepts, from variable scoping and parameter passing to inheritance based design. Graphical facilities make several traditional examples (e.g. fitting lines to points, or simulations) very vivid. Finally, graphics is a lot of fun, a factor not to be overlooked. After all, educators worldwide are concerned about dwindling student attention and how to attract students to academics.

0.5 First Day/First Month Blues

C++, like many professional programming languages, is not easy to introduce to novices. Many introductory programming books begin with a simple program that prints the message "hello world". On the face of it, this is a very natural beginning. However, even a simple program such as this appears

[3] Represented by a triangle on the screen, as in the language Logo.

complicated in C++ because it must be encased in a function, `main`, having a return type `int`. The notion of a return type is clearly inappropriate to explain on the very first day. Other concepts such as namespaces are even more daunting. The only available course of action is to tell the students, "don't worry about these, you must write these mantras whose meaning you will understand later". This doesn't seem pedagogically satisfactory.

After the student has somehow negotiated through `int`, `main` and namespaces, there is typically a long preparatory period in which substantial basic material such as data types and control structures has to be learnt, until any interesting program can be written. Psychologically and logistically, this "slow" period is a problem. Psychologically, a preparatory period without too much intellectual challenge can be viewed by the student as boring, which is a bad initial impression for the subject. Second, in most course offerings, students tend to have weekly lecture hours and weekly programming practice hours. In the initial weeks, students are fresh and raring to go. It is disappointing to them if there is nothing exciting to be done, not to mention the waste of time.

To counter these problems, the following features have been included in Simplecpp. Instead of the main program being specified inside a function main returning int, a preprocessor macro `main_program` is defined. It expands to "`int main()`". Thus, the main program can be written as

```
main_program{
body
}
```

Further, once the student loads in the Simplecpp package using `#include <simplecpp>`, nothing additional needs to be loaded, nor using directives given. The Simplecpp package itself loads other header files such as `iostream` and issues the using directives. These "training wheels" are taken off when functions, etc., are explained (Section 11.1).

A second "language extension" is the inclusion of a "repeat" statement. This statement has the form

```
repeat(count){
body
}
```

and it causes the body to execute as many times as the value of the expression count. This is also implemented using preprocessor macros and it expands into a for statement.

We believe that the repeat statement is very easy to learn, given a good context. Indeed, it is introduced in Chapter 1, where instead of using a separate statement to draw each edge of a polygon, a single line-drawing statement inside a repeat statement suffices. In fact, the turtle-based graphical drawing examples are compelling enough that students do not have any difficulty in understanding nested repeat statements either.

In the second and third chapters, there is a discussion of computer hardware, data representation and data types. These topics are important, but are not amenable to good programming exercises. For this period, the repeat statement together with the notions introduced in the first chapter can be used very fruitfully to generate relevant and interesting programming exercises.

0.6 Fitting the Book into a Curriculum

The book can be used for either a one-semester course or a two-semester sequence. For a one-semester course, the recommended syllabus is Chapter 1 through Chapter 7, Chapter 9 through Chapter 11, Chapter 13 through Chapter 15, Chapter 17, Chapter 18, Section 21.1, Chapter 22, and Chapter 25. Many of these chapters contain multiple examples of the same concept, all of these need not be "covered" in class. Some sections of these chapters could be considered "advanced", e.g. Section 10.4 which talks about game tree search for the game of Nim.

A two-semester sequence can cover Chapters 1 through Chapter 18 in the first semester, going over them carefully and considering at length aspects such as proving correctness of programs. The second semester could cover the remaining chapters. In the second semester, it would be appropriate to introduce (and use) some of the modern ideas such as reference counting pointers (Appendix G). A substantial programming project would also be appropriate. The book discusses many ideas for this.

Mumbai, May 2014 **Abhiram G Ranade**