Preface

Learning to program involves progressing through a series of basic concepts; each is fundamentally new and cannot be trivially constructed from previous ones. Programming concepts can be divided into two broad categories: *data* and *control* flow. While many books concentrate on data flow, i.e., data structures and objects, this book concentrates on control flow. The basic control-flow concepts are selection and looping, subroutine and object member call/return, recursion, routine pointer, exception, coroutine, and concurrency; each of these concepts is fundamental because it cannot be trivially constructed from the others. The difficulty in understanding and using these concepts grows exponentially from loops to concurrency, and a programmer must understand all of these concepts to be able to solve the complete gamut of modern programming problems.

Control-flow issues are extremely relevant in modern computer languages and programming styles. In addition to the basic control-flow mechanisms, virtually all new computer languages provide some form of exceptional control-flow to support robust programming. As well, concurrency capabilities are appearing with increasing frequency in both new and old programming languages. The complexity issues and reliability requirements of current and future software demand a broader knowledge of control flow; this book provides that knowledge.

The book starts with looping, and works through each of the different kinds of control-flow concepts, examining why each is fundamental and where it is useful. Time is spent on each concept according to its level of difficulty, with coroutines and concurrency receiving the most time as they should be new to the reader and because of their additional complexity. The goal is to do for control flow what is done for data structures in a data-structure book, i.e., provide a foundation in fundamental control-flow, show which problems require certain kinds of control flow, and which language constructs provide the different forms of control flow. In essence, a good data-structure book should provide its reader with the skills to select the proper data structure from a palette of structures to efficiently solve a problem; this book attempts to provide the same skill for control flow.

The book is intended for students learning advanced control-flow and concurrent programming *at the undergraduate level*. Often the only advanced control-flow

students see is concurrency, and this material is usually presented as a secondary issue in an upper-year operating-systems and/or database course. Often the amount of undergraduate time spent on concurrency is only 2 to 3 weeks of classes. Only if a student progresses to graduate school is there an opportunity to obtain additional material on concurrency. However, the need for advanced control-flow and especially concurrency is growing at a tremendous rate. New programming methodologies are requiring new forms of control flow, and new programming languages are supporting these methodologies with new control structures, such as concurrency constructs. Also, all computers now contain multi-threading and multi-cores, while multiple processors and distributed systems are ubiquitous, which all require advanced programming methodologies to take full advantage of the available parallelism. Therefore, all of these advance forms of control flow are becoming basic programming skills needed by all programmers, not just graduate students working in the operating system or database disciplines.

Prerequisites

The material in the book is presented *bottom up*: basic concepts are presented first, with more advanced concepts built slowly and methodically on previous ones. Because many of the concepts are difficult and new to most readers, extra time is spent and additional examples are presented to ensure each concept is understood before progressing to the next. The reader is expected to have the following knowledge:

- intermediate programming experience in some object-oriented language;
- an introduction to the programming language C++.

Furthermore, the kinds of languages discussed in this book are imperative programming languages, such as, Pascal, C, Ada, Modula-3, Java, C#, etc. The reader is expected to have a good understanding of these kinds of languages.

Why C++

This book uses C++ as the base programming-language. The reason is that C++ is becoming one of the dominant programming languages because it:

- is based on C, which has a large programmer and legacy code base,
- is as efficient as C in most situations,
- contains low-level features, e.g., direct memory access, needed for: systems programming, memory management, embedded/real-time,

- has high-level features, e.g., exception handling, objects, polymorphism, and an extensive standard library, which programmers now demand,
- allows direct interaction with UNIX/Windows.

Concurrency in C++

Prior to C++11, C++ did not provide all the advanced control-flow mechanisms needed for modern programming; most importantly, it lacked concurrency features. Early in the development of C++, Bjarne Stroustrup (primary creator of C++), said the following on this point:

> My conclusion at the time when I designed C++ was that no single model of concurrency would serve more than a small fraction of the user community well. I could build a single model of concurrency into C++ by providing language features that directly supported its fundamental concepts and ease its use through notational conveniences. However, if I did that I would favor a small fraction of my users over the majority. This I declined to do, and by refraining from doing so I left every form of concurrency equally badly supported by the basic C++ mechanisms. [8]

True to Stroustrup's opinion, C++11 has adopted a simple, low-level, approach to concurrency. Prior to C++11, many different concurrency approaches for C++ were implemented with only varying degrees of adoption, unlike the C programming language, which has two dominant but incompatible low-level concurrency libraries: pthreads and Win32. Interestingly, concurrency *cannot* be safely added to a language via library code [1, 2, 4, 6], so what is not there, often cannot be added later.

High-Level Concurrency

C++'s simple concurrency limits its future in the parallel domain. Therefore, it is imperative C++ be augmented with high-level concurrency facilities to extend its concurrent-programming base. The ideal scenario is for a single consistent high-level powerful concurrency mechanism; but what should it look like? In theory, any high-level concurrency paradigm/model can be adapted into C++. However, C++ does not support all concurrency approaches equally well, e.g., models such as tuple space, message passing, and channels have no preexisting connection in the language. C++ is fundamentally based on a class model using routine call, and its other features leverage this model. Any concurrency approach matching this C++ model is better served because its concepts interact consistently with the language. In other words, programmers using C++ benefit best from a design that applies the "Principle of Least Astonishment" whenever possible. Therefore, let C++'s design principles dictate which concurrency approaches fit best. For an object-oriented language, thread/stack is best associated with class, and mutual-exclusion/

synchronization with member routines. This simple observation fits many object-oriented languages, and leverages both the existing object-oriented mechanisms and programmer knowledge about them. The resulting concurrency mechanism becomes an extension of the language model, and programmers work from a single model rather than an add-on feature requiring a different way of thinking to use.

μC++

The purpose of this book is to cover advanced control-flow beyond those in C++11. While much of the material in this book is language independent, at some point it is necessary to express the material in a concrete form. Having a concrete mechanisms for use in both examples and assignments provides a powerful bridge between what is learned and how that knowledge is subsequently used in practice. To provide a concrete form, this book uses a dialect of C++, called μC++ , as a vehicle to explain ideas and present real examples.[1] (Download or Github μC++ and install it using command `sudo sh u++-6.1.0.sh`.) μC++ was developed in response to the need for expressive high-level language facilities to teach advanced control-flow and concurrency [5]. Basically, μC++ extends the C++ programming language [9] in the same way that C++ extends the C programming language. The extensions introduce new objects that augment the existing control flow facilities and provide for light-weight concurrency on uniprocessor and parallel execution on multiprocessor computers. Nevertheless, μC++ has its own design bias, short comings, and idiosyncrasies, which make it less than perfect in all situations. These problems are not specific to μC++, but occur in programming-language design, which involves many compromises to accommodate the practical aspects of the hardware and software environment in which languages execute.

This book does not cover all of the details of μC++, such as how to compile a μC++ program or how to use some of the more complex μC++ options. These details are covered in the μC++ Annotated Reference Manual [3]. When there is a discrepancy between this book and the reference manual, the manual always takes precedence, as it reflects the most recent version of the software. Unfortunately, the current dynamic nature of software development virtually precludes any computer-science textbook from being perfectly up to date with the software it discusses.

[1] A similar approach was taken to teach concurrency using a dialect of Pascal, called Pascal-FC (Functionally Concurrent) [7].

http://www.springer.com/978-3-319-25701-3

Understanding Control Flow
Concurrent Programming Using μC++
Buhr, P.A.
2016, XXI, 741 p. 100 illus. in color., Hardcover
ISBN: 978-3-319-25701-3