# Why C++?

CHAPTER 2

*A language is a dialect with an army.*
*Old proverb*

We, in the functional verification trade, write code for a living. Well, we do that, and also puzzle over code that has been written and that has yet to be written. Because functional verification is a task that only gets more complex as designs become more complex, the language we work in determines how well we can cope with this increasing complexity.

The authors believe that C++ is the most appropriate choice for functional verification, but as with any choice, there are trade-offs. This chapter discusses the advantages and disadvantages of using C++ for functional verification. We'll look at the following topics:

- An abbreviated comparison of the languages and libraries available for functional verification
- Why C++ is the best choice for verification
- The side benefits of using C++, such as the ability to share code among the software, diagnostics, and modeling teams
- The disadvantages of using C++

## Overview

Coding for functional verification can be separated into two parts. One is the generic programming part, and the other is the chip testing part. The generic part includes writing structures, functions, and interactions, using techniques such as OOP to manage complexity. The chip testing part includes connecting to the chip, running many threads, and managing random variables.

The generic programming part becomes more and more crucial as the complexity of the hardware to be tested grows. While the problem of connecting to a more complex chip tends to grow only linearly, the overall problem of dealing with this increased complexity grows exponentially.

The authors believe the generic part of programming is served well by C++. That language's features and expressive capabilities far exceed those of any other language currently in widespread use for functional verification.[1] There is already more software written today in C++ than will ever be written for just verification alone. (If one assumes that coders for both domains are equally productive, there are over three million programmers currently, which is and will always will be greater than the number of C++ verification engineers.)

Still, the most important factor is getting our job done. What about the specific issues of connecting to and exercising a chip? Aren't specific languages (such as Vera or "e") better? The answer depends on you, your team, and your project. The verification-specific languages can make it easier to wiggle the chip's "wires," exercise the chip's functionality simultaneously, and vary the configuration of and traffic through the chip. While these tasks are not trivial, the percentage of lines of code required tend to be small.

The tasks that are made simpler by a verification-specific language generally increase linearly with the complexity of the chip. In other words, there are more wires to connect, more independent threads to run, more variables to constrain, and so on.

---

1. The authors collectively have written in 14 languages and been paid to code in seven of them.