O'REILLY®

C++ Today

The Beast is Back

Jon Kalb & Gašper Ažman

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

C++ Today

The Beast Is Back

Jon Kalb & Gašper Ažman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

C++ Today
by Jon Kalb and Gašper Ažman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Rachel Roumeliotis and Katie Schooling
Production Editor: Shiny Kalapurakkel
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery

May 2015: First Edition

Revision History for the First Edition
2015-05-04: First Release
2015-06-08: Second Release

978-1-491-92758-8

[LSI]

Table of Contents

Preface

This book is a view of the C++ world from two working software engineers with decades of combined experience programming in this industry. Of course this view is not omniscient, but is filled with our observations and opinions. The C++ world is vast and our space is limited, so many areas, some rather large, and others rather interesting, have been omitted. Our hope is not to be exhaustive, but to reveal a glimpse of a beast that is ever-growing and moving fast.

CHAPTER 1
The Nature of the Beast

In this book we are referring to C++ as a "beast." This isn't from any lack of love or understanding; it comes from a deep respect for the power, scope, and complexity of the language,[1] the monstrous size of its installed base, number of users, existing lines of code, developed libraries, available tools, and shipping projects.

For us, C++ is the language of choice for expressing our solutions in code. Still, we would be the first to admit that users need to mind the teeth and claws of this magnificent beast. Programming in C++ requires a discipline and attention to detail that may not be required of kinder, gentler languages that are not as focused on performance or giving the programmer ultimate control over execution details. For example, many other languages allow programmers the opportunity to ignore issues surrounding acquiring and releasing memory. C++ provides powerful and convenient tools for handling resources generally, but the responsibility for resource management ultimately rests with the programmer. An undisciplined approach can have disastrous consequences.

Is it necessary that the claws be so sharp and the teeth so bitey? In other popular modern languages like Java, C#, JavaScript, and Python, ease of programming and safety from some forms of

1 When we refer to the C++ language, we mean to include the accompanying standard library. When we mean to refer to just the language (without the library), we refer to it as the core language.

programmer error are a high priority. But in C++, these concerns take a back seat to expressive power and performance.

Programming makes for a great hobby, but C++ is not a hobbyist language.[2] Software engineers don't lose sight of programming ease of use and maintenance, but when designing C++, nothing has or will stand in the way of the goal of creating a truly general-purpose programming language that can be used in the most demanding software engineering projects.

Whether the demanding requirements are high performance, low memory footprint, low-level hardware control, concurrency, high-level abstractions, robustness, or reliable response times, C++ must be able to do the job with reasonable build times using industry-standard tool chains, without sacrificing portability across hardware and OS platforms, compatibility with existing libraries, or readability and maintainability.

Exposure to the teeth and claws is not just the price we pay for this power and performance—sometimes, sharp teeth are exactly what you need.

C++: What's It Good For?

C++ is in use by millions[3] of professional programmers working on millions of projects. We'll explore some of the features and factors that have made C++ the language of choice in so many situations. The most important feature of C++ is that it is both low- and high-level. Due to that, it is able to support projects of all sizes, ensuring a small prototype can continue scaling to meet ever-increasing needs.

High-Level Abstractions at Low Cost

Well-chosen abstractions (algorithms, types, mechanisms, data structures, interfaces, etc.) greatly simplify reasoning about programs, making programmers more productive by not getting lost in the details and being able to treat user-defined types and libraries as well-understood and well-behaved building blocks. Using them,

2 Though some C++ hobbyists go beyond most professional programmers' day-to-day usage.

3 *http://www.stroustrup.com/bs_faq.html#number-of-C++-users*

developers are able to conceive of and design projects of much greater scope and vision.

The difference in performance between code written using high-level abstractions and code that does the same thing but is written at a much lower level[4] (at a greater burden for the programmer) is referred to as the "abstraction penalty."

As an example: C++ introduced an I/O model based on streams. The streams model offers an interface that is, in the common case, slightly slower than using native operating system calls. However, in most cases, it is fast enough that programmers choose the superior portability, flexibility, and type-safety of streams to faster but less-friendly native calls.

C++ has features (user-defined types, type templates, algorithm templates, type aliases, type inference, compile-time introspection, runtime polymorphism, exceptions, deterministic destruction, etc.) that support high-level abstractions and a number of different high-level programming paradigms. It doesn't force a specific programming paradigm on the user, but it does support procedural, object-based, object-oriented, generic, functional, and value-semantic programming paradigms and allows them to easily mix in the same project, facilitating a tailored approach for each part.

While C++ is not the only language that offers this variety of approaches, the number of languages that were also designed to keep the abstraction penalty as low as possible is far smaller.[5] Bjarne Stroustrup, the creator of C++, refers to his goal as "the zero-overhead principle," which is to say, no abstraction penalty.

A key feature of C++ is the ability of programmers to create their own types, called *user-defined types* (UDTs), which can have the power and expressiveness of built-in types or fundamentals. Almost anything that can be done with a fundamental type can also be done with a user-defined type. A programmer can define a type that functions as if it is a fundamental data type, an object pointer, or even as a function pointer.

---

4 For instance, one can (and people do) use virtual functions in C, but few will contest that `p->vtable->foo(p)` is clearer than `p->foo()`.

5 Notable peers are the D programming language, Rust, and, to a lesser extent, Google Go, albeit with a much smaller installed base.

C++ has so many features for making high-quality, easy to use libraries that it can be thought of as a language for building libraries. Libraries can be created that allow users to express themselves in a natural syntax and still be powerful, efficient, and safe. Libraries can be designed that have type-specific optimizations and to automatically clean up resources without explicit user calls.

It is possible to create libraries of generic algorithms and user-defined types that are just as efficient or almost as efficient as code that is not written generically.

The combination of powerful UDTs, generic programming facilities, and high-quality libraries with low abstraction penalties make programming at a much higher level of abstraction possible even in programs that require every last bit of performance. This is a key strength of C++.

Low-Level Access When You Need It

C++ is, among other things, a systems-programming language. It is capable of and designed for low-level hardware control, including responding to hardware interrupts. It can manipulate memory in arbitrary ways down to the bit level with efficiency on par with hand-written assembly code (and, if you really need it, allows inline assembly code). C++, from its initial design, is a superset of C,[6] which was designed to be a "portable assembler," so it has the dexterity and memory efficiency to be used in OS kernels or device drivers.

One example of the kind of control offered by C++ is the flexibility available for where user-defined types can be created. Most high-level languages create objects by running a construction function to initialize the object in memory allocated from the heap. C++ offers that option, but also allows for objects to be created on the stack. Programmers have little control over the lifetime of objects created on the stack, but because their creation doesn't require a call to the heap allocator, stack allocation is typically orders of magnitude faster. Due to its limitations, stack-based object allocation can't be a

6 Being a superset of C also enhances the ability of C++ to interoperate with other languages. Because C's string and array data structures have no memory overhead, C has become the "connecting" interface for all languages. Essentially all languages support interacting with a C interface and C++ supports this as a native subset.

general replacement for heap allocation, but in those cases where stack allocation is acceptable, C++ programmers win by avoiding the allocator calls.

In addition to supporting both heap allocation and stack allocation, C++ allows programmers to construct objects at arbitrary locations in memory. This allows the programmer to allocate buffers in which many objects can be very efficiently created and destroyed with great flexibility over object lifetimes.

Another example of having low-level control is in cache-aware coding. Modern processors have sophisticated caching characteristics, and subtle changes in the way the data is laid out in memory can have significant impact on performance due to such factors as look-ahead cache buffering and false sharing.[7] C++ offers the kind of control over data memory layout that programmers can use to avoid cache line problems and best exploit the power of hardware. Managed languages do not offer the same kind of memory layout flexibility. Managed language containers do not hold objects in contiguous memory, and so do not exploit look-ahead cache buffers as C++ arrays and vectors do.

Wide Range of Applicability

Software engineers are constantly seeking solutions that scale. This is no less true for languages than for algorithms. Engineers don't want to find that the success of their project has caused it to outgrow its implementation language.

Very large applications and large development teams require languages that scale. C++ has been used as the primary development language for projects with hundreds of engineers and scores of modules.[8] Its support for separate compilation of modules makes it possible to create projects where analyzing and/or compiling all the project code at once would be impractical.

A large application can absorb the overhead of a language with a large runtime cost, either in startup time or memory usage. But to be useful in applications as diverse as device drivers, plug-ins, CGI

7 *http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206*

8 For a small sample of applications and operating systems written in C++: *http://www.stroustrup.com/applications.html*

modules, and mobile apps, it is necessary to have as little overhead as possible. C++ has a guiding philosophy of "you only pay for what you use." What that means is that if you are writing a device driver that doesn't use many language features and must fit into a very small memory footprint, C++ is a viable option, where a language with a large runtime requirement would be inappropriate.

Highly Portable

C++ is designed with a specific hardware model in mind, and this model has minimalistic requirements. This has made it possible to port C++ tools and code very broadly, as machines built today, from nanocomputers to number-crunching behemoths, are all designed to implement this hardware model.

There are one or more C++ tool chains available on almost all computing platforms.[9] C++ is the only high-level language alternative available on all of the top mobile platforms.[10]

Not only are the tools available, but it is possible to write portable code that can be used on all these platforms without rewriting.

With the consideration of tool chains, we have moved from language features to factors outside of the language itself. But these factors have important engineering considerations. Even a language with perfect syntax and semantics wouldn't have any practical value if we couldn't build it for our target platform.

In order for an engineering organization to seriously consider significant adoption of a language, it needs to consider availability of tools (including analyzers and other non-build tools), experienced engineers, software libraries, books and instructional material, troubleshooting support, and training opportunities.

Extra-language factors, such as the installed user base and industry support, always favor C++ when a systems language is required and tend to favor C++ when choosing a language for building large-scale applications.

9 "An incomplete list of C++ compilers": *http://www.stroustrup.com/compilers.html*

10 C++ is supported on iOS, Android, Windows Mobile, and BlackBerry: *http://visualstudiomagazine.com/articles/2013/02/12/future-c-plus-plus.aspx*

Better Resource Management

In the introduction to this chapter, we discussed that other popular languages prioritize ease of programming and safety over performance and control. Nothing is a better example of the differences between these languages and C++ than their approaches to memory management.

Most popular modern languages implement a feature called *garbage collection*, or GC. With this approach to memory management, the programmer is not required to explicitly release allocated memory that is no longer needed. The language runtime determines when memory is "garbage" and recycles it for reuse. The advantages to this approach may be obvious. Programmers don't need to track memory, and "leaks" and "double dispose" problems[11] are a thing of the past.

But every design decision has trade-offs, and GC is no exception. One issue with it is that collectors don't recognize that memory has become garbage immediately. The recognition that memory needs to be released will happen at some unspecified future time (and for some, implementations may not happen at all—if, for example, the application terminates before it needs to recycle memory).

Typically, the collector will run in the background and decide when to recycle memory outside of the programmer's control. This can result in the foreground task "freezing" while the collector recycles. Since memory is not recycled as soon as it is no longer needed, it is necessary to have an extra cushion of memory so that new memory can be allocated while some unneeded memory has not yet been recycled. Sometimes the cushion size required for efficient operation is not trivial.

An additional objection to GC from a C++ point of view is that memory is not the only resource that needs to be managed. Programmers need to manage file handles, network sockets, database connections, locks, and many other resources. Although we may not be in a big hurry to release memory (if no new memory is being requested), many of these other resources may be shared with other

11 It would be hard to over-emphasize how costly these problems have been in non-garbage collected languages.

processes and need to be released as soon as they are no longer needed.

To deal with the need to manage all types of resources and to release them as soon as they can be released, best-practice C++ code relies on a language feature called *deterministic destruction*.

In C++, one way that objects are instantiated by users is to declare them in the scope of a function, causing the object to be allocated in the function's stack frame. When the execution path leaves the function, either by a function return or by a thrown exception, the local objects are said to have gone out of scope.

When an object goes out of scope, the runtime "cleans up" the object. The definition of the language specifies that objects are cleaned up in exactly the reverse order of their creation (reverse order ensures that if one object depends on another, the dependent is removed first). Cleanup happens immediately, not at some unspecified future time.

As we pointed out earlier, one of the key building blocks in C++ is the user-defined type. One of the options programmers have when defining their own type is to specify exactly what should be done to "clean up" an object of the defined type when it is no longer needed. This can be (and in best practice is) used to release any resources held by the object. So if, for example, the object represents a file being read from or written to, the object's cleanup code can automatically close the file when the object goes out of scope.

This ability to manage resources and avoid resource leaks leads to a programming idiom called RAII, or Resource Acquisition Is Initialization.[12] The name is a mouthful, but what it means is that for any resource that our program needs to manage, from file handles to mutexes, we define a user type that acquires the resource when it is initialized and releases the resource when it is cleaned up.

To safely manage a particular resource, we just declare the appropriate RAII object in the local scope, initialized with the resource we need to manage. The resource is guaranteed to be cleaned up exactly once, exactly when the managing object goes out of scope, thus solv-

12 It may also stand for Responsibility Acquisition Is Initialization when the concept is extended beyond just resource management.

ing the problems of resource leaks, dangling pointers, double releases, and delays in recycling resources.

Some languages address the problem of managing resources (other than memory) by allowing programmers to add a `finally` block to a scope. This block is executed whenever the path of execution leaves the function, whether by function return or by thrown exception. This is similar in intent to deterministic destruction, but with this approach, every function that uses an object of a particular resource managing type would need to have a `finally` block added to the function. Overlooking a single instance of this would result in a bug.

The C++ approach, using RAII, has all the convenience and clarity of a garbage-collected system, but makes better use of resources, has greater performance and flexibility, and can be used to manage resources other than memory. Generalizing resource management instead of just handling memory is a strong advantage of this approach over garbage collection and is the reason that most C++ programmers are not asking that GC be added to the language.

Industry Dominance

C++ has emerged as the dominant language in a number of diverse product categories and industries.[13] What these domains have in common is either a need for a powerful, portable systems-programming language or an application-programming language with uncompromising performance. Some domains where C++ is dominant or near dominant include search engines, web browsers, game development, system software and embedded computing, automotive, aviation, aerospace and defense contracting, financial engineering, GPS systems, telecommunications, video/audio/image processing, networking, big science projects, and ISVs.[14]

13 *http://www.lextrait.com/vincent/implementations.html*

14 Independent software vendors, the people that sell commercial applications for money. Like the creators of Office, Quicken, and Photoshop.

CHAPTER 2

The Origin Story

This may be old news to some readers, and is admittedly a C++-centric telling, but we want to provide a sketch of the history of C++ in order to put its recent resurgence in perspective.

The first programming languages, such as Fortran and Cobol, were developed to allow a domain specialist to write portable programs without needing to know the arcane details of specific machines.

But systems programmers were expected to master such details of computer hardware, so they wrote in assembly language. This gave programmers ultimate power and performance at the cost of portability and tedious detail. But these were accepted as the price one paid for doing systems programming.

The thinking was that you either were a domain specialist, and therefore wanted or needed to have low-level details abstracted from you, or you were a systems programmer and wanted and needed to be exposed to all those details. The systems-programming world was ripe for a language that allowed to you ignore those details except when access to them was important.

C: Portable Assembler

In the early 1970s, Dennis Ritchie introduced "C,"[1] a programming language that did for systems programmers what earlier high-level

1 *http://cm.bell-labs.co/who/dmr/chist.html*

languages had done for domain specialists. It turns out that systems programmers also want to be free of the mind-numbing detail and lack of portability inherent in assembly-language programming, but they still required a language that gave them complete control of the hardware when necessary.

C achieved this by shifting the burden of knowing the arcane details of specific machines to the compiler writer. It allowed the C programmer to ignore these low-level details, except when they mattered for the specific problem at hand, and in those cases gave the programmer the control needed to specify details like memory layouts and hardware details.

C was created at AT&T's Bell Labs as the implementation language for Unix, but its success was not limited to Unix. As the portable assembler, C became the go-to language for systems programmers on all platforms.

C with High-Level Abstractions

As a Bell Labs employee, Bjarne Stroustrup was exposed to and appreciated the strengths of C, but also appreciated the power and convenience of higher-level languages like Simula, which had language support for *object-oriented programming* (OOP).

Stroustrup realized that there was nothing in the nature of C that prevented it from directly supporting higher-level abstractions such as OOP or type programming. He wanted a language that provided programmers with both elegance when expressing high-level ideas and efficiency of execution size and speed.

He worked on developing his own language, originally called C With Classes, which, as a superset of C, would have the control and power of portable assembler, but which also had extensions that supported the higher-level abstractions that he wanted from Simula. [DEC]

The extensions that he created for what would ultimately become known as C++ allowed users to define their own types. These types could behave (almost) like the built-in types provided by the language, but could also have the inheritance relationships that supported OOP.

He also introduced templates as a way of creating code that could work without dependence on specific types. This turned out to be very important to the language, but was ahead of its time.

The '90s: The OOP Boom, and a Beast Is Born

Adding support for OOP turned out to be the right feature at the right time for the '90s. At a time when GUI programming was all the rage, OOP was the right paradigm, and C++ was the right implementation.

Although C++ was not the only language supporting OOP, the timing of its creation and its leveraging of C made it the mainstream language for software engineering on PCs during a period when PCs were booming.

The industry interest in C++ became strong enough that it made sense to turn the definition of the language over from a single individual (Stroustrup) to an ISO (International Standards Organization) Committee.[2] Stroustup continued to work on the design of the language and is an influential member of the ISO C++ Standards Committee to this day.[3]

In retrospect, it is easy to see that OOP, while very useful, was overhyped. It was going to solve all our software engineering problems because it would increase modularity and reusability. In practice, reusability goes up within specific frameworks, but these frameworks introduce dependencies, which reduce reusability between frameworks.

Although C++ supported OOP, it wasn't limited to any single paradigm. While most of the industry saw C++ as an OOP language and was building its popularity and installed base using object frameworks, others where exploiting other C++ features in a very different way.

2 *http://www.open-std.org/jtc1/sc22/wg21/*

3 Most language creators retain control of their creation or give them to standards bodies and walk away. Stroustrup's continuing to work on C++ as part of the ISO is a unique situation.

Alex Stepanov was using C++ templates to create what would eventually become known as the Standard Template Library (STL). Stepanov was exploring a paradigm he called *generic programming*.

Generic programming is "an approach to programming that focuses on designing algorithms and data structures so that they work in the most general setting without loss of efficiency." [FM2G]

Although the STL was a departure from every other library at the time, Andrew Koenig, then the chair of the Library Working Group for the ISO C++ Standards Committee, saw the value in it and invited Stepanov to make a submission to the committee. Stepanov was skeptical that the committee would accept such a large proposal when it was so close to releasing the first version of the standard. Koenig asserted that Stepanov was correct. The committee would not accept it…if Stepanov didn't submit it.

Stepanov and his team created a formal specification for his library and submitted it to the committee. As expected, the committee felt that it was an overwhelming submission that came too late to be accepted.

Except that it was brilliant!

The committee recognized that generic programming was an important new direction and that the STL added much-needed functionality to C++. Members voted to accept the STL into the standard. In its haste, it did trim the submission of a number of features, such as hash tables, that it would end up standardizing later, but it accepted most of the library.

By accepting the library, the committee introduced generic programming to a significantly larger user base.

In 1998, the committee released the first ISO standard for C++. It standardized "classic" C++ with a number of nice improvements and included the STL, a library and programming paradigm clearly ahead of its time.

One challenge that the Library Working Group faced was that it was tasked not to create libraries, but to standardize common usage. The problem it faced was that most libraries were either like the STL (not in common use) or they were proprietary (and therefore not good candidates for standardization).

Also in 1998, Beman Dawes, who succeeded Koenig as Library Working Group chair, worked with Dave Abrahams and a few other members of the Library Working Group to set up the Boost Libraries.[4] Boost is an open source, peer-reviewed collection of C++ libraries,[5] which may or may not be candidates for inclusion in the standard.

Boost was created so that libraries that might be candidates for standardization would be vetted (hence the peer reviews) and popularized (hence the open source).

Although it was set up by members of the Standards Committee with the express purpose of developing candidates for standardization, Boost is an independent project of the nonprofit Software Freedom Conservancy.[6]

With the release of the standard and the creation of Boost.org, it seemed that C++ was ready to take off at the end of the '90s. But it didn't work out that way.

The 2000s: Java, the Web, and the Beast Nods Off

At over 700 pages, the C++ standard demonstrated something about C++ that some critics had said about it for a while: C++ is a complicated beast.

The upside to basing C++ on C was that it instantly had access to all libraries written in C and could leverage the knowledge and familiarity of thousands of C programmers.

But the downside was that C++ also inherited all of C's baggage. A lot of C's syntax and defaults would probably be done very differently if it were being designed from scratch today.

Making the more powerful user-defined types of C++ integrate with C so that a data structure defined in C would behave exactly the same way in both C and C++ added even more complexity to the language.

---

4 *http://www.boost.org/users/proposal.pdf*

5 *http://boost.org/*

6 *https://sfconservancy.org/*

The addition of a streams-based input/output library made I/O much more OOP-like, but meant that the language now had two complete and completely different I/O libraries.

Adding operator overloading to C++ meant that user-defined types could be made to behave (almost) exactly like built-in types, but it also added complexity.

The addition of templates greatly expanded the power of the language, but at no small increase in complexity. The STL was an example of the power of templates, but was a complicated library based on generic programming, a programming paradigm that was not appreciated or understood by most programmers.

Was all this complexity worth it for a language that combined the control and performance of portable assembler with the power and convenience of high-level abstractions? For some, the answer was certainly yes, but the environment was changing enough that many were questioning this.

The first decade of the 21st century saw desktop PCs that were powerful enough that it didn't seem worthwhile to deal with all this complexity when there were alternatives that offered OOP with less complexity.

One such alternative was Java.

As a bytecode interpreted, rather than compiled, language, Java couldn't squeeze out all the performance that C++ could, but it did offer OOP, and the interpreted implementation was a powerful feature in some contexts.[7]

Because Java was compiled to bytecode that could be run on a Java virtual machine, it was possible for Java applets to be downloaded and run in a web page. This was a feature that C++ could only match using platform-specific plug-ins, which were not nearly as seamless.

So Java was less complex, offered OOP, was the language of the Web (which was clearly the future of computing), and the only downside

7 "Build once, run anywhere," while still often not the case with Java, is sometimes much more useful for deployment than the "write once, build anywhere" type of portability of C++.

was that it ran a little more slowly on desktop PCs that had cycles to spare. What's not to like?

Java's success led to an explosion of what are commonly called managed languages. These compile into bytecode for a virtual machine with a just-in-time compiler, just like Java. Two large virtual machines emerged from this explosion. The elder, Java Virtual Machine, supports Java, Scala, Jython, Jruby, Clojure, Groovy, and others. It has an implementation for just about every desktop and server platform in existence, and several implementations for some of them. The other, the Common Language Interface, a Microsoft virtual machine, with implementations for Windows, Linux, and OS X, also supports a plethora of languages, with C#, F#, IronPython, IronRuby, and even C++/CLI leading the pack.

Colleges soon discovered that managed languages were both easier to teach and easier to learn. Because they don't expose the full power of pointers[8] directly to programmers, it is less elegant, and sometimes impossible, to do some things that a systems programmer might want to do, but it also avoids a number of nasty programming errors that have been the bane of many systems programmers' existence.

While things were going well for Java and other managed languages, they were not going so well for C++.

C++ is a complicated language to implement (much more than C, for example), so there are many fewer C++ compilers than there are C compilers. When the Standards Committee published the first C++ standard in 1998, everyone knew that it would take years for the compiler vendors to deliver a complete implementation.

The impact on the committee itself was predictable. Attendance at Standards Committee meetings fell off. There wasn't much point in defining an even newer version of the standard when it would be a few years before people would begin to have experience using the current one.

About the time that compilers were catching up, the committee released the 2003 standard. This was essentially a "bug fix" release

8 Java's "references" can be `null`, and can be re-bound, so they are pointers; you just can't increment them.

with no new features in either the core language or the standard library.

After this, the committee released the first and only C++ Technical Report, called TR1. A technical report is a way for the committee to tell the community that it considers the content as standard-candidate material.

The TR1 didn't contain any change to the core language, but defined about a dozen new libraries. Almost all of these were libraries from Boost, so most programmers already had access to them.

After the release of the TR1, the committee devoted itself to releasing a new update. The new release was referred to as "0x" because it was obviously going to be released sometime in 200x.

Only it wasn't. The committee wasn't slacking off—they were adding a lot of new features. Some were small nice-to-haves, and some were groundbreaking. But the new standard didn't ship until 2011. Long, long overdue.

The result was that although the committee had been working hard, it had released little of interest in the 13 years from 1998 to 2011.

We'll use the history of one group of programmers, the ACCU, to illustrate the rise and fall of interest in C++. In 1987, The C Users Group (UK) was formed as an informal group for those who had an interest in the C language and systems programming. In 1993, the group merged with the European C++ User Group (ECUG) and continued as the Association of C and C++ Users.

By the 2000s, members were interested in languages other than C and C++, and to reflect that, the group changed its name to just the initials ACCU. Although the group is still involved in and supporting C++ standardization, its name no longer stands for C++, and members are also exploring other languages, especially C#, Java, Perl, and Python.[9]

9 *http://accu.org/index.php/aboutus*

By 2010, C++ was still in use by millions of engineers, but the excitement of the '90s had faded. There had been over a decade with few enhancements released by the Standards Committee. Colleges and the cool kids were defecting to Java and managed languages. It looked like C++ might just turn into another legacy-only beast like Cobol.

But instead, the beast was just about to roar back.

CHAPTER 3

The Beast Wakes

In this chapter and the next, we are going to be exploring the factors that drove interest back to C++ and the community's response to this growing interest. However, we'd first like to point out that, particularly for the community responses, this isn't entirely a one-way street. When a language becomes more popular, people begin to write and talk about it more. When people write and talk about a language more, it generates more interest.

Debating the factors that caused the C++ resurgence versus the factors caused by it isn't the point of this book. We've identified what we think are the big drivers and the responses, but let's not forget that these responses are also factors that drive interest in C++.

Technology Evolution: Performance Still Matters

Performance has always been a primary driver in software development. The powerful desktop machines of the 2000s didn't signal a permanent change in our desire for performance; they were just a temporary blip.

Although powerful desktop machines continue to exist and will remain very important for software development, the prime targets for software development are no longer on the desk (or in your lap). They are in your pocket and in the cloud.

Modern mobile devices are very powerful computers in their own right, but they have a new concern for performance: *performance per watt*. For a battery-powered mobile device, there is no such thing as spare cycles.

Earlier we pointed out that C++ is the only high-level language available[1] for all mobile devices running iOS, Android, or Windows. Is this because Apple, which adopted Objective-C and invented Swift, is a big fan of C++? Is it because Google, which invented Go and Dart, is a big fan of C++? Is it because Microsoft, which invented C#, is a big fan of C++? The answer is that these companies want their devices to feature apps that are developed quickly, but are responsive and have long battery life. That means they need to offer developers a language with high-level abstraction features (for fast development) and high performance. So they offer C++.

Cloud-based computers, that is, computers in racks of servers in some remote data center, are also powerful computers, but even there we are concerned about performance per watt. In this case, the concern isn't dead batteries, but power cost. Power to run the machines, and power to cool them.

The cloud has made it possible to build enormous systems spanning hundreds, thousands, or tens of thousands of machines bound to a single purpose. A modest improvement in speed at those scales can represent substantial savings in infrastructure costs.

James Hamilton, a vice president and distinguished engineer on the Amazon Web Services team, reported on a study he did of modern high-scale data centers.[2] He broke the costs down into (in decreasing order of significance) servers, power distribution & cooling, power, networking equipment, and other infrastructure. Notice that the top three categories are all directly related to software performance, either performance per hardware investment or performance per watt. Hamilton determined that 88% of the costs are dependent on performance. A 1% performance improvement in code will almost produce a 1% cost savings, which for a data center at scale will be a significant amount of money.

---

1 C++ is not necessarily the *recommended* language on mobile platforms but is supported in one way or another.

2 *http://perspectives.mvdirona.com/2010/09/overall-data-center-costs/*

For companies with server farms the size of Amazon, Facebook, Google, or Microsoft, not using C++ is an expensive alternative.

But how is this different from how computing in large enterprise companies has always been done? Look again at the list of expense categories. Programmers and IT professionals are not listed. Did Hamilton forget them? No. Their cost is in the noise. Managed languages that have focused on programmer productivity at the expense of performance are optimizing for a cost not found in the modern scaled data center.[3]

Performance is back to center stage, and with it is an interest in C++ for both cloud and mobile computing. For mobile computing, the "you only pay for what you use" philosophy and the ability to run in a constrained memory environment are additional wins. For cloud computing, the fact that C++ is highly portable and can run efficiently and reliably on a wide variety of low-cost hardware are additional wins, especially because one can tune directly for the hardware one owns.

Language Evolution: Modernizing C++

In 2011, the first major revision to Standard C++ was released, and it was very clear that the ISO Committee had not been sitting on its hands for the previous 13 years. The new standard was a major update to both the core language and the standard library.[4]

The update, which Bjarne Stroustrup, the creator of C++, reported "feels like a new language,"[5] seemed to offer something for everyone. It had dozens of changes, some small and some fundamental, but the most important achievement was that C++ now had the features programmers expected of a modern language.

The changes were extensive. The page count of the ISO Standard went from 776 for the 2003 release to 1,353 for the 2011 release. It

3 To the extent that such languages are being used for prototyping, to bring features to market quickly, or for software that doesn't need to run at scale, there is still a role for these languages. But it isn't in data centers at scale.

4 And much appreciated. In a 2015 survey, Stack Overflow found that C++11 was the second "most loved" language of its users (after newcomer Swift). *http://stackoverflow.com/research/developer-survey-2015*

5 *https://isocpp.org/tour*

isn't our purpose here to catalogue them all. Other references are available for that.[6] Instead, we'll just give some idea about the kinds of changes.

One of the most important themes of the release was simplifying the language. No one would like to "tame the beast" of its complexity more than the Standards Committee. The challenge that the committee faces is that it can't remove anything already in the standard because that would break existing code. Breaking existing code is a nonstarter for the committee.

It may not seem possible to simplify by adding to an already complicated specification, but the committee found ways to do exactly that. It addressed some minor annoyances and inconsistencies, and added the ability to have the compiler deduce types in situations where the programmer used to have to spell them out explicitly. It added a new version of the "for" statement that would automatically iterate over containers and other user-defined types.

It made enumeration and initialization syntax more consistent, and added the ability to create functions that take an arbitrary number of parameters of a specified type.

It has always been possible in C++ to define user-defined types that can hold state and be called like functions. However, this ability has been underutilized because the syntax for creating user-defined types for this purpose was verbose, was hardly obvious, and as such added some inconvenient overhead. The new language update introduced a new syntax for defining and instantiating function objects (lambdas) to make them convenient to use. Lambdas can also be used as closures, but they do not automatically capture the local scope—the programmer has to specify what to capture explicitly.

The 2011 update added better support for character sets, in particular, better support for Unicode. It standardized a regular expression library (from Boost via the TR1) and added support for "raw" literals that makes working with regular expressions easier.

The standard library was significantly revised and extended. Almost all of the libraries defined in the TR1 were incorporated into the standard. Types that were already defined in the standard library,

6 *http://en.wikipedia.org/wiki/C%2B%2B11*

such as STL containers, were updated to reflect new core language features; and new containers, such as a singly-linked list and hash-based associative containers, were added.

All of these features were additions to the language specification, but had the effect of making the language simpler to learn and use for everyday programming.

Reflecting that C++ is a language for library building, a number of new features made life easier for library authors. The update introduced language support for "perfect forwarding." *Perfect forwarding* refers to the ability of a library author to capture a set of parameters to a function and "forward" these to another function without changing anything about the parameters. Boost library authors had demonstrated that this was achievable in classic C++, but only with great effort and language mastery.

Now, mere mortals can implement libraries using perfect forwarding by taking advantage of a couple of features new in the 2011 update: variadic templates and rvalue references.

A richer type system allows better modeling of requirements that can be checked at compile time, catching wide classes of bugs automatically. The tighter the type system models the problem, the harder it is for bugs to slip through the cracks. It also often makes it easier for compilers to prove additional invariants, enabling better automatic code optimization. New features aimed at library builders included better support for type functions.[7]

Better support for compile-time reflection of types[8] enables library writers to adapt their libraries to wide varieties of user types, using the optimal algorithms for the capabilities the user's objects expose without additional burden on the users of the library.

The update also broke ground in some new areas. Writing multi-threaded code in C++ has been possible, but only with the use of platform-specific libraries. With the concurrency support intro-

7 Implemented as templated `using` aliases.

8 Through a plethora of new type-traits and subtle corrections to the SFINAE rules. Substitution Failure is not an Error is an important rule for finding the correct template to instantiate, when more than one appears to match initially. It allows for probing for capabilities of types, since using a capability that isn't offered will just try a different template.

duced in the 2011 update, it is now possible to write multithreaded code and libraries in a portable way.

This update also introduced move semantics, which Scott Meyers referred to as the update's "marquee feature." Avoiding unnecessary copies is a constant challenge for programmers who are concerned about performance, which C++ programmers almost always are. Because of the power and flexibility of "classic" C++, it has always been possible to avoid unnecessary copies, but sometimes this was at the cost of code that took longer to write, was less readable, and was harder to reuse.

Move semantics allow programmers to avoid unnecessary copies with code that is straightforward in both writing and reading. Move semantics are a solution to an issue (unnecessary copies) that C++ programmers care about, but is almost unnoticed in other language environments.

This isn't a book on how to program. Our goal is to talk *about* C++, not teach it. But we can't help ourselves, we want to show what modern C++ really means, so if you are interested in code examples of how C++ is evolving, don't skip Chapter 5, Digging Deep on Modern C++.

As important as it was to have a new standard, it wouldn't have had any meaningful impact if there were no tools that implemented it.

Tools Evolution: The Clang Toolkit

Due to its age and the size of its user base, there are many tools for C++ on many different platforms. Some are proprietary, some are free, some are open source, some are cross-platform. There are too many to list, and that would be out of scope for us here. We'll discuss a few interesting examples.

Clang is the name of a compiler frontend for the C family of languages.[9] Although it was first released in 2007, and its code generation reached production quality for C and Objective-C later that decade, it wasn't really interesting for C++ until this decade.

9 C, C++, Objective-C, and Objective-C++

Clang is interesting to the C++ community for two reasons. The first is that it is a new C++ compiler. Due to its wide feature-set and a few syntactic peculiarities that make it very hard to parse, new C++ frontends don't come along everyday. But more than just being an alternative, its value lay in its much more helpful error messages and significantly faster compile times.

As a newer compiler, Clang is still catching up with older compilers on the performance of generated code[10] (which is usually of primary consideration for C++ programmers). But its better build time and error messages increase programmer productivity. Some developers have found a best-of-both-worlds solution by using Clang for the edit-build-test-debug cycle, but build production releases with an older compiler. For developers using GCC, this is facilitated by Clang's desire to be "drop in" compatible with GCC. Clang brought some helpful competition to the compiler space, making GCC also improve significantly. This competition is benefiting the community immensely.

One result of the complexity of C++ is that compile-time error messages can sometimes be frustratingly inscrutable, particularly where templates are involved. Clang established its reputation as a C++ compiler by generating error messages that were more understandable and more useful to programmers. The impact that Clang's error messages have had on the industry can be seen in how much other compilers have improved their own.[11]

The second reason that Clang is interesting to the C++ community is because it is more than just a compiler; it is an open source toolkit that is itself implemented in high-quality C++. Clang is factored to support the building of development tools that "understand" C++.

Clang contains a static analysis framework, which the `clang-tidy` tool uses. Writing additional checkers for the framework is quite simple. Using the Clang toolkit, programmers can build dynamic analyzers, source-to-source translators, refactoring tools, or make any number of other kinds of tools.

10 For some CPUs and/or code cases, it has caught up or passed its competitors.

11 Some examples comparing error messages from Clang with old and newer versions of GCC: *https://gcc.gnu.org/wiki/ClangDiagnosticsComparison*

There are a number of dynamic analyzers that come built into Clang: AddressSanitizer,[12] MemorySanitizer,[13] LeakSanitizer,[14] and ThreadSanitizer.[15] The compile time flag `-fdocumentation` will look for Doxygen-style comments and warn you if the code described doesn't match the comments.

Metashell[16] is an interactive environment for template metaprogramming. American fuzzy lop[17] is a security-oriented fuzzer that uses code-coverage information from the binary under test to guide its generation of test cases. Mozilla has built a source code indexer for large code bases called DXR.[18]

Over time, the performance of Clang's generated code will improve, but the importance of that will pale compared to the impact on the community of the tools that will be built from the Clang toolkit. We'll see more and more tools for understanding, improving, and verifying code as well as have a platform for trying out new core language features.[19]

Library Evolution: The Open Source Advantage

The transition to a largely open source world has benefited C++ relative to managed languages, but especially Java. This came from two sources. First, shipping source code further improved runtime-performance of C++; and second, the availability of source reduced the advantage of Java's "build once, run anywhere" deployment story, since "write once, build for every platform" became viable.

The model used by most proprietary libraries was for the library vendor to ship library headers and compiled object files to application developers. Among the implications of this are the fact that this limits the portability options available to application developers.

12 *http://clang.llvm.org/docs/AddressSanitizer.html*

13 *http://clang.llvm.org/docs/MemorySanitizer.html*

14 *http://clang.llvm.org/docs/LeakSanitizer.html*

15 *http://clang.llvm.org/docs/ThreadSanitizer.html*

16 *https://metashell.readthedocs.org/en/latest/*

17 *http://lcamtuf.coredump.cx/afl/*

18 *https://dxr.readthedocs.org/en/latest/*

19 Clang and its standard library implementation, libc++, are usually the first compiler and library to implement new C++ features.

Library vendors can't provide object files for every possible hardware/OS platform combination, so inevitably practical limits prevented applications from being offered on some platforms because required libraries were not readily available.

Another implication is that library vendors, again for obvious practical reasons, couldn't provide library object files compiled with every combination of compiler settings. This would mean the final application was almost always suboptimal in the way that their libraries were compiled.

One particular issue here is processor-specific compilation. Processor families have a highly compatible instruction set that all new processors support for backward compatibility. But new processors often add new instructions to enable their new features. Processors also vary greatly in their pipeline architectures, which can make code that performs well on one processor less desirable on another. Compiling for a specific processor is therefore highly desirable.

This fact had worked in Java's favor. Earlier we referred to Java as an interpreted language, which is true to a first approximation, but managed languages are implemented with a just-in-time compiler that can enhance performance over what would be possible by strictly interpreting bytecode.[20] One way that the JIT can enhance performance is to compile for the actual processor on which it is running.

A C++ library provider would tend to provide a library object compiled to the "safe," highly-compatible instruction set, rather than have to supply a number of different object files, one for each possible processor. Again, this would often result in suboptimal performance.

But we no longer live in a world dominated by proprietary libraries. We live in an open source world. The success and influence of the Boost libraries contributed to this, but the open source movement has been growing across all languages and platforms. The fact that

20 The JIT has the ability to see the entire application. This allows for optimizations that would not be possible to a compiler linking to compiled library object files. Today's C++ compilers use link-time (or whole-program) optimization features to achieve these optimizations. This requires that object files be compiled to support this feature. On the other hand, the JIT compiler was hampered by the very dynamic nature of Java, which forbade most of the optimizations the C++ compiler can do.

libraries are now available as source code means that developers can target any platform with any compiler and compiler options that they choose, and support optimizations that require the source.

Cloud computing only reinforces this advantage. In a cloud computing scenario, developers can target their own hardware with custom builds that free the compiler to optimize for the particular target processor.

Closed-source libraries also forced library vendors to eschew the use of templates, instead relying on runtime dispatch and object-oriented programming, which is slower and harder to make type-safe. This effectively barred them from using some of the most powerful features of C++. These days, vending template libraries with barely any compiled objects is the norm, which tends to make C++ a much more attractive proposition.

CHAPTER 4
The Beast Roars Back

In this chapter, we'll discuss a number of C++ resources, most of which are either new or have been revitalized in the last few years. Of course this isn't an exhaustive list. Google and Amazon are your friends.

WG21

Our first topic is the ISO Committee for C++ standardization, which at 25 years old, is hardly a new resource, but it certainly glows with new life. The committee is formally called ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21).[1] Now you know why most people just call it the C++ Standards Committee.

As big an accomplishment as it is to release a new or updated major standard like C++98 or C++11, it doesn't have much practical impact if there are no tools that implement it. As mentioned earlier, this was a significant issue with the release of the standard in 1998. Committee attendance fell off because implementation was understood to be years away.

But this was not the case for the release in 2011. Tool vendors had been tracking the standard as it was being developed. Although it called for significant changes to both the core language and the stan-

1 *https://isocpp.org/std/the-committee*

dard library, the new update was substantially implemented by a couple of different open source implementations, GCC and Clang, soon after its release.[2] Other tool vendors had also demonstrated their commitment to the update. Unlike some language updates,[3] this was clearly adopted by the entire community as *the* path forward.

The psychological impact of this should not be underestimated. Thirteen years is a very long time in the world of programming languages. Some people had begun to think of C++ as an unchanging definition, like chess or Latin. The update changed the perception of C++ from a dying monster of yesteryear into a modern, living creature.

The combination of seeing C++ as a living creature, and one where implementations closely followed[4] standardization, meant that Standards Committee meeting attendance began to increase.[5]

The committee reformulated itself[6] to put the new members to the best use. It had long been formed of a set of subcommittees called working groups. There were Core, Library, and Evolution working groups; but with many new members and so many areas in which the industry is asking for standardization, new working groups were the answer. The Committee birthed over a dozen new "Domain Specific Investigation & Development" groups.

The first new product of the committee was a new standard in 2014. C++14 was not as big a change as C++11, but it did significantly improve some of the new features in C++11 by filling in the gaps discovered with real-world experience using these new features.

The target for the next standard release is 2017. The working groups are busy reviewing and developing suggestions that may or may not become part of the next update.

---

2 Note the weasel words "substantially" and "soon." Complete and bug-free implementations of every feature aren't the point here. The crux is that the community recognized that C++11 was a real thing and the need to get on board right away.

3 We are looking at you, Python 3.0.

4 Or, in the case of some features, preceded.

5 The meeting where a new standard is officially voted to be a release is always highly attended. The first few meetings after that did see a bit of a drop, but soon the upward trend was clear.

6 *https://isocpp.org/std/the-committee*

The existence of a vital Standards Committee that is engaged with the language users, tool vendors, and educators is a valuable resource. Actively discussing and debating possible features is a healthy process for the entire community.

In Chapter 6, The Future of C++, we'll discuss more about the working groups and what they are working on.

Tools

Clang is clearly the most significant new development in the C++ toolchain, but the resurgence of interest in C++ has brought more than just Clang to the community. We'll discuss a few interesting newcomers.

biicode[7] is a cross-platform dependency manager that you can use to download, configure, install, and manage publicly available libraries, like the Boost libraries. This is old-hat for other languages, but this is new technology for C++. The site only hit 1.0 in the middle of 2014 and is still in beta, but it has thousands of users and has been growing aggressively.

Undo Software[8] has a product called UndoDB, which is a reversible debugger. The idea of a reversible debugger, one that supports stepping backward in time, is so powerful that it has been implemented many times. The problem with previous implementations is that they run so slowly and require so much data collection that they aren't practical for regular use. Undo has found a way to increase the speed and reduce the data requirements so that the approach is practical for the first time. This product isn't C++ only, but its marketing is focused on the C++ community.

JetBrains[9] has built its reputation on IDEs with strong refactoring features and has over a dozen products for developers. But until launching CLion[10] in 2015, it's not had a C++ product. CLion is a cross-platform IDE for C and C++ with support for biicode. CLion can be used on Windows, but for developers that use Microsoft's

7 *https://www.biicode.com/*

8 *http://undo-software.com/*

9 *https://www.jetbrains.com/*

10 *https://www.jetbrains.com/clion/*

Visual Studio for C++ development, JetBrains is updating ReSharper,[11] its VS extension, which supports C#, .NET, and web-development languages to also support C++.

The last tool that we'll mention isn't a development tool, but a deployment tool. OSv[12] is an operating system written in C++ that is optimized for use in the cloud. By rethinking the requirements of a virtual machine existing in the cloud, Cloudius Systems has created an OS with reduced memory and CPU overhead and lightweight scheduling. Why was this implemented in C++ instead of C? It turns out that gets asked a lot:

> While C++ has a deserved reputation for being incredibly complicated, it is also incredibly rich and flexible, and in particular has a very flexible standard library as well as add-on libraries such as boost. This allows OSV code to be much more concise than it would be if it were written in C, while retaining the close-to-the-metal efficiency of C.[13]
>
> —OSv FAQ

Standard C++ Foundation

Many languages[14] are either created by or adopted by a large company that considers the adoption of the language by others a strategic goal and so markets and promotes the language. Although AT&T was supportive of C++,[15] it never "marketed" the language to encourage adoption by external developers.

Various tool vendors and publishers have promoted C++ tools or books, but until this decade, no organization[16] has marketed C++ itself. This didn't seem to be an impediment to the growth and acceptance of the language. But in 2010, as the committee was about to release the largest update to the standard since it was created,

11 *https://www.jetbrains.com/resharper/*

12 *http://osv.io/*

13 *http://osv.io/frequently-asked-questions/*

14 Consider Java, JavaScript, C#, Objective-C.

15 It gave the rights to the C++ manual to the ISO. *http://www.stroustrup.com/bs_faq.html#revenues*

16 The only central organization for C++ was the Standards Committee, but promoting the language was outside of its charter (and resources).

interest in C++ had noticeably increased. The time seemed ripe for a central place for C++-related information.

At least it seemed like a good idea to Herb Sutter, the Standards Committee's Convener.[17] Sutter wanted to build an organization that would promote C++ and be independent of (but supported by) the players in the C++ community. With their support, he was able to launch the Standard C++ Foundation[18] and *http://isocpp.org* in late 2012.

In addition to serving as a single feed for all C++-related news, the website also became the home for the "C++ Super-FAQ."[19] The Super-FAQ acquired its name because it is the merger of two of the largest FAQs in C++.

Bjarne Stroustrup, as the language's creator, was the target of countless, often repetitive questions, so he had created a large FAQ on his personal website.[20]

The moderators of the Usenet group comp.lang.c++ were also maintaining a FAQ[21] for C++. In 1994, Addison-Wesley published this as "C++ Faqs" by moderators Marshall Cline and Greg Lomow. In 1998, Cline and Lomow were joined by Mike Girou with the second edition,[22] which covered the then recently released standard.

Both of these FAQs have been maintained online separately for years, but with the launch of *http://isocpp.org*, it was clearly time to merge them. The merged FAQ is in the form of a wiki so that the community can comment and make improvements.

Today isocpp.org is the home not only to the best source of news about the C++ community and the Super-FAQ, but also has a list of upcoming events, some "getting started" help for people new to C++, a list of free compilers,[23] a list of local C++ user groups,[24] a list

17 "Convener" is ISO speak for committee chair.

18 *http://isocpp.org/about/*

19 *https://isocpp.org/faq*

20 *http://www.stroustrup.com/*

21 The concept and term "FAQ" was invented by Usenet group moderators.

22 *http://www.amazon.com/FAQs-2nd-Marshall-P-Cline/dp/0201309831/*

23 *https://isocpp.org/get-started*

24 *https://isocpp.org/wiki/faq/user-groups-worldwide*

of tweets,[25] recent C++ questions from Stack Overflow, information about the Standards Committee and the standards process, including statuses, upcoming meetings,[26] and links for discussion forums by working group.[27]

Boost: A Library and Organization

As described earlier, Boost was created to host free, open source, peer-reviewed libraries that may or may not be candidates for standardization. Boost has grown to include over 125 libraries,[28] is the most used C++ library outside of the standard library, and it has been the single best source of libraries accepted into the standard since its inception in 1998.

The Boost libraries and boost.org have become the center of the Boost community, which is made up of the volunteers who have developed, documented, reviewed, maintained, and distributed the libraries.

Since 2005, Boost as an organization has regularly participated in the Google Summer of Code program, giving students an opportunity to learn cutting-edge C++ library development skills.[29]

Since 2006, Boost has gathered for an intimate, week-long annual conference, originally called BoostCon. More about this later.

Recently, Robert Ramey, a Boost library author, has built the Boost Library Incubator[30] to help C++ programmers produce Boost-quality libraries. The incubator offers advice and support for authors and provides interested parties with the opportunity to examine code and documentation of candidate libraries and leave comments and reviews. There are currently over 20 libraries[31] in the incubator, all open to reviews and/or comments.

---

25 Follow @isocpp *https://twitter.com/isocpp*

26 *https://isocpp.org/std*

27 *https://isocpp.org/forums*

28 *http://www.boost.org/doc/libs/*

29 *http://www.boost.org/community/gsoc.html*

30 *http://rrsd.com/blincubator.com/*

31 *http://rrsd.com/blincubator.com/alphabetically/*

C++ is an amazing tool for building high-quality libraries and frameworks, so while the 125+ libraries in Boost are the most distributed (other than the standard library), they only scratch the surface of the libraries and frameworks available for C++. There are publicly available lists of libraries on Wikipedia[32] and cppreference.com.[33]

Q&A

The Internet revolution has changed the practical experience of writing code in any language. The combination of powerful public search engines and websites with reference material and thousands of questions and answers dramatically reduces the time lag between needing an answer about a particular language feature, library call, or programming technique, and finding that answer.

The impact is disproportionately large for languages that are very complicated, have a large user base (and therefore lots of online resources) or, like C++, both. Here are some online resources that C++ programmers have found useful.

Nate Kohl noticed that there were some sites with useful references for other languages,[34] so in 2000, he launched cppreference.com.[35] Initially he posted documentation as static content that he maintained himself. From the beginning, there were some contributions from across the Internet, but in 2008, the contribution interest was too much to be manageable, so he converted the site to a wiki.

Kohl's approach is to start with high-level descriptions and present increasing detail that people can get into if they happen to be interested. His theory is that examples are more useful to people trying to solve a problem quickly than rigorous formal descriptions.

The wiki has the delightful feature that all the examples are compilable right there on the website. You can modify the example and then run it to see the result. *Right there on the wiki!*

32 *http://en.wikipedia.org/wiki/Category:C%2B%2B_libraries*

33 *http://en.cppreference.com/w/cpp/links/libs*

34 Such as Sun's early Java API reference site.

35 *http://cppreference.com*

As useful as documentation and examples are, some people learn better in a question-and-answer format. In 2008, Stack Overflow[36] launched as a resource for programmers to get their questions answered. Stack Overflow allows users to submit questions about all kinds of programming topics and currently contains over 300,000 answered questions on C++.

In order for a Q&A site to be useful, it needs to provide a good way to find the question you are looking for, and high-quality answers. Your favorite Internet search engine does pretty well with Stack Overflow, and the answers tend to be of high quality. Post a question, and it might be answered by Jonathan Wakely, Howard Hinnant, James McNellis, Dietmar Kühl, Sebastian Redl, Anthony Williams, Eric Niebler, or Marshall Clow. These are the people that have built the libraries you are asking about.

As useful as a Q&A site like Stack Overflow is, some people feel more comfortable in a more traditional forum environment. Alex Allain, who wrote the book *Jumping into C++* [JIC] has built cprogramming.com[37] into a community site of its own with references, tutorials, book reviews, tips, problems, quizes, and several forums.

Of course, for those that like their Q&A retro style, some usernet groups still exists for C++: comp.lang.c++,[38] comp.lang.c++.moderated,[39] and comp.std.c++.[40]

As you'd expect of a language with a large user base, there are a lot of Internet hangouts for learning about and discussing C++. There are dozens of blogs[41] and an active subreddit.[42] Jens Weller maintains a blog at Meeting C++ called Blogroll that is a weekly list of the latest C++ blogs.[43]

36 *http://stackoverflow.com/questions/tagged/c%2b%2b*

37 *http://cprogramming.com*

38 *https://groups.google.com/forum/!forum/comp.lang.c++.moderated*

39 *https://groups.google.com/forum/!forum/comp.lang.c++.moderated*

40 *https://groups.google.com/forum/#!forum/comp.std.c++*

41 *http://www.quora.com/What-are-the-best-blogs-on-C++*

42 *http://www.reddit.com/r/cpp/*

43 *http://meetingcpp.com/index.php/blogroll.html*

A special mention goes to the freenode.net ##C++ IRC channel, members of which are known for mercilessly tearing apart every snippet of code you might care to show them. Funny how harsh critique makes good programmers. They also take care of the channel's pet, `geordi`, the friendliest C++ evaluation bot the world has ever known.

Conferences and Groups

Throughout the 2000s, the market for conference-going C++ programmers was largely served by SD West in the US and ACCU in Europe. Neither conference was explicitly for C++, but both attracted a lot of C++ developers and content.

Beginning in 2010 and for every year since, Andrei Alexandrescu, Scott Meyers, and Herb Sutter have worked together to produce C++ and Beyond.[44] They've described it as a "conference-like event" rather than a conference. We won't quibble. These are three-speaker, three-day events with advanced presentations by the authors of some the most successful books on C++.[45] Registration is limited[46] to provide for more speaker-audience interaction. Most attendees have over a decade of C++ experience, are well informed about programming in C++, and value the opportunity for informal discussions with the speakers.

In 2006, Dave Abrahams and Beman Dawes started BoostCon, which was designed to allow the Boost community to meet face to face and discuss ideas with each other and users. The intention was for content to be Boost Library-related, but serve a wider audience than just the Boost Library developers. Over time, the content became a little more mainstream, but it always focused on cutting-edge library development, and attendance was never greater than 100.

While at BoostCon in May, 2011, Jon Kalb[47] approached the conference planning committee with the idea of making BoostCon more

44 *http://cppandbeyond.com/*

45 And D.

46 The registration limit has varied from 60 to 120, depending on the venue. The 2010 event was so popular an "encore" event was held a couple of months later.

47 The same.

mainstream. He argued that BoostCon, while small, was very successful and had the potential to be *the* mainstream C++ conference of North America. Kalb proposed that BoostCon change its name to something with C++ in it, add a third track, and grow the number of attendees. He pointed out that by the next conference (May 2012), the new standard update would be released, and there would be a lot of demand for sessions on C++11. The new track could be entirely made up of C++11 tutorials. The planning committee accepted the ideas, and C++Now was born. Something must have been in the air, because C++Now was only one of three new C++ conferences in 2012.

Late in 2011, Microsoft announced the first GoingNative conference for February 2012, about three months before the first C++Now. This conference was different from C++Now in a number of ways, but was the same in one important way.

Despite the fact that it was produced (and subsidized) by Microsoft, the content was entirely about portable, standard C++. It was larger, with probably about four times as many attendees as BoostCon. It was shorter, lasting two days as opposed to a week. GoingNative sessions were professionally live-streamed to the world, instead of the "in-house" video recording done at BoostCon/C++Now. Instead of multiple tracks with sessions by speakers from across the community, GoingNative had a single track filled entirely with "headliners." Almost all of the GoingNative 2012 speakers either had been BoostCon keynote speakers or would be later be C++Now keynoters.

C++Now 2012 had three tracks,[48] including one that was a C++11 tutorial track. Conference attendance jumped to 135 from 85 the previous year.

Jens Weller, inspired by attending C++Now, decided to create a similar conference for Europe in his home country of Germany. The first Meeting C++ conference[49] was held in late 2012, and at 150 attendees, it was larger in its first year than C++Now.

48 BoostCon always had two tracks. Initially, the plan was that one track was for Boost Library developers and the other for users. Over time this distinction was lost, but the conference continued to have two tracks.

49 *http://meetingcpp.com/*

Weller has been an active C++ evangelist,[50] and Meeting C++ has continued to grow. It is now at four tracks and is expecting 400 attendees in 2015. Weller's influence has extended beyond the Meeting C++ conference. He has launched and is supporting several local Meeting C++ user groups across Europe.[51]

The list of C++ user groups[52] includes groups in South America as well as North America and Europe.

C++Now 2013 reached the registration limit that planners had set at 150. The Boost Steering Committee decided that instead of continuing to grow the conference, it would cap attendance at 150 indefinitely. After this decision was announced, Chandler Carruth, treasurer of the Standard C++ Foundation, spoke with Kalb about launching a new conference under the auspices of the foundation.

Later, Carruth and Kalb would pitch this to Herb Sutter, the foundation's chair and president. He was instantly on board, and CppCon was born. The first CppCon attracted almost 600 attendees to Bellevue, Washington in September of 2014. It had six tracks featuring 80 speakers, 100 sessions, and a house band. One of the ambitious goals of CppCon is to be *the* platform for discussion about C++ across the entire community. To further that goal, the conference had its sessions professionally recorded and edited so that over 100 hours of high-quality C++ lectures are freely available.[53]

Videos

The CppCon session videos supplement an amazing amount of high-quality video content on C++. Most of the C++ conferences mentioned earlier have posted some or all of their sessions online free. The BoostCon YouTube channel[54] has sessions from both

50 Weller also blogs regularly about the proposal papers for each standards meeting: *http://meetingcpp.com/index.php/blog.html*

51 *http://meetingcpp.com/index.php/user-groups.html*

52 *https://isocpp.org/wiki/faq/user-groups-worldwide*

53 After publishing the videos on YouTube, the conference received requests for an alternative from developers in countries where YouTube is blocked. So the conference asked Microsoft's Channel 9 to host them as well.

54 *https://www.youtube.com/user/Boostcon*

BoostCon and C++Now. Both Meeting C++[55] and CppCon[56] have YouTube channels as well.

Channel 9, Microsoft's developer information channel, makes its videos available in a wide variety of formats, including audio-only for listening on the go. In addition to hosting the CppCon videos,[57] some sessions from C++ and Beyond in 2011[58] and 2012,[59] all of the sessions of the two GoingNative conferences in 2012[60] and 2013,[61] and a number of other videos on C++,[62] Channel 9 also has a series on C++ that is called C9::GoingNative.[63]

If your tastes or requirements run more toward formal training, there are some good C++ courses on both pluralsight[64] and udemy.[65]

CppCast

Although there are lots of C++ videos, there is relatively little audio.[66] This follows from the fact that when discussing C++, we almost always want to look at code, which doesn't work well with just audio. But audio does work for interviews, and Rob Irving has launched CppCast,[67] the only podcast dedicated to C++.

Books

There are hundreds of books on C++, but in an era of instant Internet access to thousands of technical sites and videos of almost every

55 *https://www.youtube.com/user/MeetingCPP*

56 *https://www.youtube.com/user/CppCon*

57 *http://channel9.msdn.com/Events/CPP/C-PP-Con-2014*

58 *http://channel9.msdn.com/Tags/cppbeyond+2011*

59 *http://channel9.msdn.com/Tags/cppbeyond+2012*

60 *http://channel9.msdn.com/Events/GoingNative/GoingNative-2012*

61 *http://channel9.msdn.com/Events/GoingNative/2013*

62 *http://channel9.msdn.com/Tags/c++*

63 *http://channel9.msdn.com/Shows/C9-GoingNative*

64 *http://www.pluralsight.com/tag/c++*

65 *https://www.udemy.com/courses/search/?q=c%2B%2B*

66 Although Channel 9 content is available as audio-only, this tends to work well only for panels and interviews.

67 *http://cppcast.com/*

subjects, publishing tech books is not the business that it once was, so the number of books with coverage of the 2011 and/or 2014 releases is not large.

We should make the point that because of the backward compatibility of standard updates, most of the information in classic C++ books is still largely valid. For example, consider what Scott Meyers has said about his classic book *Effective C++* [68]

> Whether you're programming in "traditional" C++, "new" C++, or some combination of the two, then, the information and advice in this book should serve you well, both now and in the future.[69]

Still, using a quality book that is written with the current standard in mind gives you confidence that there isn't a better way to do something. Here are a few classic C++ books that have new editions updated to C++11 or C++14:

- Bjarne Stroustrup has new editions of two of his books. *Programming: Principles and Practice Using C++, 2nd Edition* [PPPUC] is a college-level textbook for teaching programming that just happens to use C++. He has also updated his classic *The C++ Programming Language* [TCPL] with a fourth edition. The overview portion of this book is available separately as *A Tour of C++* [ATOC], a draft of which can be read online free at isocpp.org.[70]
- Nicolai Josuttis has released a second edition of his classic, *The C++ Standard Library: A Tutorial and Reference* [TCSL], which covers C++11.
- Barbara Moo has updated the *C++ Primer* [CP] to a fifth edition that covers C++11. The primer is a gentler introduction to C++ than *The C++ Programming Language*.

Here are a couple of books that are new, not updates of classic C++ versions:

Scott Meyers' *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* [EMC] was one of the most eagerly awaited books in the community. An awful lot of today's C++ pro-

68 *http://www.amazon.com/Effective-Specific-Improve-Programs-Designs/dp/0321334876/*

69 *http://scottmeyers.blogspot.de/2011/03/effective-c-in-c0x-c11-age.html*

70 *https://isocpp.org/tour*

grammers feel like the *Effective C++* series was a formative part of our C++ education, and we've wanted to know for a long time what Scott's take on the new standard would be. Now we can find out.

Unlike the previously mentioned books, Anthony Williams' *C++ Concurrency in Action: Practical Multithreading* [CCIA] is not about C++ generally, but just focuses on the new concurrency features introduced in C++11. The author is truly a concurrency expert. Williams has been the maintainer of the Boost Thread library since 2006, and is the developer of the just::thread implementation of the C++11 thread library. He also authored many of the papers that were the basis of the thread library in the standard.

CHAPTER 5

Digging Deep on Modern C++

The power of the additional features that were introduced with the 2011 and 2014 updates comes not just from the changes, but from the way these changes integrate with classic features. This is the primary reason the update feels like a whole new language, rather than a version of classic C++ with a collection of new features bolted on.

In this chapter, we will demonstrate what that means, which requires looking at some code. Feel free to skip this chapter if your interest in C++ is not as a coder.

Type Inference: Auto and Decltype

When a language supports type inference, it is often presented as just a convenient way to not have to explicitly write out types. "The compiler already knows the type—why should the programmer have to write it out?" Indeed, this point of view is important. Oftentimes, the type is just visual clutter, as demonstrated by the definition and usage of c_v_s_iter in Example 5-1, which is written in a pre-C++11 style.

Example 5-1. Type inference: visual type clutter

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

// trivial implementation of the unix uniq utility
```

```
// read lines from stdin
// write sorted unique lines to stdout
int main(int argc, char** argv) {
  using std::vector; using std::string; using std::sort;
  using std::unique; using std::cin; using std::cout;

  vector<string> lines;
  while (cin) {
    lines.emplace_back();
    getline(cin, lines.back());
  };

  sort(lines.begin(), lines.end());

  typedef typename vector<string>::const_iterator c_v_s_iter;
  c_v_s_iter const last = unique(lines.begin(), lines.end());
  lines.resize(last - lines.begin()); // keep only the result of
                                      // unique

  for (c_v_s_iter i = lines.begin(), e = lines.end(); i < e; ++i) {
    cout << (*i) << '\n';
  }
}
```

If this doesn't seem like a big deal, it's because it isn't—in this case. Still, it breaks the flow of reading code because of completely unnecessary complications. One's mind wonders if the equivalent Python would be easier to read.

We can do better with a modern style.

Example 5-2. Type inference: less clutter

```
  sort(begin(lines), end(lines));
  auto const last = unique(begin(lines), end(lines));
  lines.resize(last - begin(lines)); // keep only the result of
                                     // unique

  for (auto const& line : lines) {
    cout << line << '\n';
  }
```

Once code becomes more complex, overflowing the programmer's working memory, one is grateful for the absence of unnecessary symbols.

However, type inference does not merely make things easier. In some cases, it takes code from unthinkable[1] to obvious. For example, let's implement it with `boost::range`.

Example 5-3. Type inference: complex iterator types

```
sort(begin(lines), end(lines));
for (auto const& line : lines | uniqued) {
  cout << line << '\n';
}
```

The type `uniqued` returns is intractable at the least (especially over multiple pipes of `filtered`, `sliced`, etc.), and moreover, the programmer really does not care about what the iterator type is. The only relevant fact is that dereferencing it yields a string.[2] The code therefore expresses intent without extraneous detail and without losing a sliver of performance.

The addition of type inference does more than give the programmer another tool for writing programs faster. It also fundamentally destroyed the last remnants of the notion that type annotations are there for the compiler. Instead, the types one actually writes out are a layer of assertions that the programmer instructs the compiler to *check*, and clues to the reader. Writing `auto` means that the type is unimportant; only its behavior, as defined by its usage, is important. Writing the type out explicitly means the type *has* to be exactly what it says. Put another way: *types are henceforth always intentional, never circumstantial*.

Put this way, it is trivial to recognize the missing element in this scenario: `auto` means *any type*, and writing the type out explicitly means *exactly this type, or a type convertible to it*. There is no way, however, to constrain the type only partially. The family of solutions to this problem are called *concepts* in C++, but so far, the committee has not reached consensus on important details of this feature. Get-

1 While Boost has continualy proven that nothing is *impossible* (`boost::bind` comes to mind), for the vast majority of programmers, unthinkable is much the same thing as impossible.

2 Before C++11, one had to use std::copy to transfer the results into an output iterator, which in this case would be a `std::ostream_iterator<string>(cout, "\n")`, but this is inelegant and rather inflexible compared to just using a `for` loop, since writing output iterators for every purpose is rather involved, not to mention verbose.

ting it right the first time is important, as is not breaking existing code, and so concepts have been left out of the standard so far. The search for the ideal solution continues.

How Move Semantics Support Value-Semantic and Functional Programming

In the previous chapter, we indirectly explored the beautiful world of procedural programming: `std::sort` and `std::unique` are algorithms that mutate state passed to them. While this makes them wonderful building blocks for procedural programs, they do not compose well with a more functional style of programming.[3]

In C++, writing in a functional style often came with a price of a mandatory copy of a parameter.[4] Consider the implementation of `sorted` and `uniqued` as functions.

Example 5-4. Implementation of sorted

```
template <typename Container>
Container sorted(Container x) {
  std::sort(begin(x), end(x));
  return x;
}
```

Example 5-5. Implementation of uniqued

```
template <typename Container>
Container uniqued(Container x) {
  x.resize(std::unique(begin(x), end(x)) - begin(x));
  return x;
}
```

In classic C++, the natural way of getting a vector of unique lines `uniqued(sorted(lines))` results in two (or three) expensive copies: first, `lines` is copied to become `sorted`'s parameter, and then the return value of `sorted` is copied to become `uniqued`'s parameter.

3 In functional programming, functions do not modify global state or the state of their parameters. Computational results are exclusively captured by return values. *https://wiki.haskell.org/Functional_programming*

4 For performance-obsessed C++ programmers, issues like this could make functional programming a nonstarter.

Finally, the return value of `uniqued` is copied into a variable in the local scope, should one choose to store it.

In C++11, the two return values are moved instead[5]—in the case of `vector`, only pointers to the internal structure are copied. The rationale for moving return values instead of copying is that we know that the very next thing that happens to the returned temporary object is destruction. This well-chosen mechanism enabled us to get rid of two copies. But what about the first copy of `lines`? We get rid of it using `std::move`, which instructs the compiler to treat its parameter as if it were an unnamed temporary.

Example 5-6. Getting rid of all copies

```
lines = uniqued(sorted(std::move(lines)));
```

In the snippet, there are no copies at all. As a final step, the result is moved back into `lines`.[6] This is just as efficient as the procedural version, while the functions, as implemented, supply all of the advantages of reasoning about code that come with a pure-functional style.

The ranged for loop is as easy to write as it would be in Python, but without performance loss.

Example 5-7. Ranged for loops can iterate over function return values

```
for (auto const& line : uniqued(sorted(std::move(lines)))) {
  cout << line << '\n';
}
```

No More Output Parameters

One of the problems classic C++ inherited from C is that functions only return a single value. While one could have, in fact, returned a `std::pair`, `boost::tuple`, or other type defined simply to hold multiple values, it was not commonly done in practice, in part because returning large objects tended to be expensive if not done

5 If the copy is not elided entirely. "Elided" is standard-speak for a copy omitted as unnecessary.

6 Should we have needed `lines` to be untouched, we would have assigned to a different variable and not used the move().

correctly,[7] and in part because returning objects just to unpack them in several following lines is rather verbose.

The alternative was to use *output parameters*. One would create the objects that would become a function's return value before calling the function, and then pass references to them to the function, which would assign values to them.

This is very efficient in execution, but not in programmer productivity. The output parameters are indistinguishable from the input parameters of the function when one is reading code, necessitating a thorough knowledge of exactly what the function does to its parameters in order to reason about code. Output parameters, consequently, were recognized as detracting from clarity.

The new tuple library, aided by move semantics to stay fast and lean, allows for much improvement. Consider the case where one wants to calculate some statistics of a sequence of numbers. We chose length, minimum and maximum, with the average and variance left as an exercise for the reader.

Example 5-8. Compute the length, min, and max of the values

```
template <typename ConstInputIterator,
          typename MinMaxType =
              iterator_value_type<ConstInputIterator>>
auto lenminmax(ConstInputIterator first, ConstInputIterator last)
    -> std::tuple<size_t, MinMaxType, MinMaxType> {
  if (first == last) { return {0, 0, 0}; }
  size_t count{1};
  auto minimum(*first);
  auto maximum(minimum); // only evaluate *first once
  while (++first != last) {
    ++count;
    auto const value = *first;
    if (value < minimum) {
      minimum = value;
    } else if (maximum < value) {
      maximum = value;
    }
  }
  return {count, minimum, maximum};
}
```

7 Return value optimization, while explicitly provisioned for by the standard, is not understood by the majority of programmers and is not applicable in all situations.

The function returns three values, is as efficient as can be, and does not mutate any external state.

The following example shows how to use it:

Example 5-9. Use the lenminmax function

```
vector<int const> samples{5, 3, 6, 2, 4, 8, 9, 12, 3};
int min, max;
tie(ignore, min, max) = lenminmax(samples.begin(), samples.end());

cout << "minimum: " << min << "\n"
     << "maximum: " << max << "\n";
```

Notice how we used `tie` to assign to min and max and ignore the length.

There is one more thing to consider: how did we discover the type of the tuple that `lenminmax` returns? We seem to use the magical `iterator_value_type` to infer the type the iterator returns.[8] Logically, the type of minimum and maximum is whatever type the iterator's reference is: the type of `*first`.

We can get that type with `decltype(*first)`. Still, that's not quite right, because `*first` returns a reference, and we need a value type. Fortunately, there is a way to get rid of references in types: `std::remove_reference<T>::type`. We just need to supply the type, which makes `std::remove_reference<decltype(*first)>::type`. This is quite a mouthful, so we would be better served if we could make a type alias for it. However, in the type alias, we cannot use "`first`" because the name is not defined outside of the function. Still, we need some kind of value inside the `decltype` to dereference. Again, the standard library has what we need: `declval<T>()`. `declval<T>()` gives us a fictional reference to use inside `decltype` in place of `first`. The result is in the next example.

8 In the olden days, we would rely on iterator traits, which generated much confusion in teaching, and required library authors to provide them for their structures, which did not always happen.

Example 5-10. How to get the type of the value that any iterator points to

```
template <typename T>
using iterator_value_type = typename std::remove_reference<
                              decltype(*std::declval<T>())>::type;
```

Now, we can just use it everywhere, like in the definition of the `len minmax` function.

Inner Functions with Lambdas

Sometimes, an algorithm requires an action to always be performed in a particular way. Measurement is often such a thing. For instance, every time one writes into an output iterator, one must increment it (see `push` in the next example).

Merge sort is an algorithm that sorts a sequence of items by first splitting it into already sorted subsequences[9] and then merging them two by two into successively longer sequences, until only one is left. The merge algorithm works by comparing the heads of both input sequences and moving the smaller one to the end of the output sequence. It repeats this process until both input sequences have been consumed.

This is conceptually very simple. When writing the merge routine, it very quickly crystallizes that we shall have to write two nearly identical pieces of code — one for when the head of the first sequence is to be moved, and one for the second sequence. However, with an inner function (`advance`), we can write this piece of code once, and just call it twice, with the roles of both sequences reversed.

Example 5-11. Using lambdas for inner functions to simplify algorithms

```
template <typename InputIterator1, typename InputIterator2,
          typename OutputIterator>
auto move_merge(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator&& out) -> OutputIterator {
  using std::move; using std::forward;
```

9 Note that a sequence of zero or one items is, by definition, sorted.

```
  auto drain = [&out](auto& first, auto& last){
    return move(first, last, forward<OutputIterator>(out));
  };
  auto push = [&out](auto& value) { *out = move(value); ++out; };
  auto advance = [&](auto& first_a, auto& last_a, auto& value_a,
                     auto& first_b, auto& last_b, auto& value_b) {
    push(value_a);
    if (++first_a != last_a) {
      value_a = move(*first_a);
      return true;
    } else { // the sequence has ended. Drain the other one.
      push(value_b);
      out = drain(++first_b, last_b);
      return false;
    }
  };

  if      (first1 == last1) { return drain(first2, last2); }
  else if (first2 == last2) { return drain(first1, last1); }
  auto value1(move(*first1));
  auto value2(move(*first2));
  for (bool not_done = true; not_done;) {
    if (value2 < value1) {
      not_done = advance(first2, last2, value2,
                         first1, last1, value1);
    } else {
      not_done = advance(first1, last1, value1,
                         first2, last2, value2);
    }
  }
  return out;
}
```

Also notice that we were able to give the part of the algorithm that drains the final remaining sequence into the output sequence a name, even though the actual implementation is only one line. Every part of the algorithm reads cleanly.

This is also a great example of the difference in style between inner functions and regular functions. This much mutation, in-out parameters, etc., are extremely poor style when designing function signatures, because the implementation might be far from the point of use. In most contexts, in-out parameters cause higher cognitive load for the programmer, making bugs and slowdowns more probable.

However, inner functions are not public, and their implementation is close at hand—it is in the same scope! Here, in-out parameters do not cause higher cognitive load. Instead, they help us understand

that the algorithm has the same structure for both branches after the comparison and make sure that it is in fact the same both times.

We defined `drain` in order to omit the rather cumbersome forwarding syntax from the call of move in order to make the names of the sequence iterators stand out better where it is called.

The purpose of `push` is that output iterators have to be incremented every time they are dereferenced, and some such iterators are rather heavy to copy. In order to be able to use pre-increment, two lines are needed.

Finally, `advance` is the meat of the algorithm. The reason for its definition is the aforementioned fact that, after we compare the heads of sequences and thus determine which head to move, moving one head looks exactly the same as moving the other.

Lambdas as a Scope with a Return Value

Lambda expressions can also be directly evaluated, finally allowing for some logic in initializing references and variables that are not default-constructible because they hold resources. Let's take a look at a very simple implementation of a multithreaded producer-consumer queue.

For performance in using synchronized data structures, one should release a lock as soon as possible. At (1) in the next example, the element is returned as soon as it has been popped off the queue; the function then ends, the lock is released, and the element is processed afterward.

Example 5-12. Returning from a scope

```
deque<int> queue;
bool done = false;
mutex queue_mutex;
condition_variable queue_changed;

thread producer([&]{
  for (int i = 0; i < 1000; ++i) {
    {
      unique_lock<mutex> lock{queue_mutex};
      queue.push_back(i);
    }
    // one must release the lock before notifying
    queue_changed.notify_all();
```

```
  } // end for
  {
    unique_lock<mutex> lock{queue_mutex};
    done = true;
  }
  queue_changed.notify_all();
});
thread consumer([&]{
  while (true) {
    auto maybe_data = [&]()->boost::optional<int>{ // (1)
      unique_lock<mutex> lock{queue_mutex};
      queue_changed.wait(lock,
                         [&]{return done || !queue.empty();});
      if (!queue.empty()) {
        auto data = move(queue[0]);
        queue.pop_front();
        return boost::make_optional(move(data));
      }
      return {};
    }(); // release lock
    // do stuff with data!
    if (maybe_data) { std::cout << *maybe_data << '\n'; }
    else { break; }
  }
});
producer.join();
consumer.join();
```

Most often, you will want to encapsulate this logic into a class, but it is exceedingly hard to design general queuing interfaces that are nevertheless fast in all scenarios. Sometimes, an ad hoc approach is exactly what is needed, such as in a book, where the limit is a page. Without the gratuitous use of lambda functions, this code would be longer and less clear.

CHAPTER 6
The Future of C++

We don't claim any particular gifts at seeing the future, but we can peak at where the standard is going, and then we'll point out some of the trends that we see today that we think will continue to be important, perhaps even more important in the future.

Setting the Standard

What will be in C++17? Of course we won't have a complete answer until it is finished, but we can get a hint about where the language is going by looking at what the committee is working on now. One clue is to look at the working groups that were formed after the 2011 release and what they are working on.

The new groups include:

Concurrency
: Multithreading libraries have always been a part of C++, but only with the 2011 release has concurrency been part of the standard. But concurrency is a huge topic, and only basic building blocks were provided in C++11/14. There is room for standard thread pool and other concurrency tools. This is also the group that looks at vectorization and how to exploit GPUs, a much needed enhancement to C++.

Modules
: The header-based include model needs to be replaced by a system that explicitly defines what a module wants to make public.

This can support dramatically faster builds and better encapsulation of libraries.

File System
: It currently isn't possible to read the contents of a directory in a standard portable way. This group is looking at the Boost FileSystem library.

Networking (Inactive[1])
: Currently all networking done in C++ (and there is a lot) is done with nonportable libraries. It is time that networking is part of the standard.

Transactional Memory
: The future is concurrency, but "locks" don't compose or scale. Transactional memory is one possible solution.[2]

Numerics
: Fortran is often seen as the language of choice for programs requiring extensive numeric computations, particularly if matrixes are required. The Standards Committee isn't willing to cede this domain.

Reflection
: C++ reflection is limited. It is likely to continue to be restricted to compile time, but there are a lot of opportunities for improvement even with this limitation.

Concepts
: Investigating how to define constraints on types used in generic programming.

Ranges
: Investigating how to update the standard library with a range concept rather than iterator pairs and how to extend this to include containers and range-based algorithms.

Feature Test
: Standard features are rolled out as fast as tool vendors can provide them. This group is looking at how to define a portable standard way to check for the presence of new features.

---

1 Its work is complete.

2 *http://research.cs.wisc.edu/trans-memory/*

Databases (Inactive[3])
: Database-related library interfaces.

Undefined and Unspecified Behavior
: This group is reviewing all the areas that the standard calls out as undefined or unspecified behavior in order to recommend a set of changes to how these are called out.

I/O
: This group is looking into standardizing low-level graphic/pointing I/O primitives.

One of the committee's highest priorities is backward compatibility. Any existing standard-compliant code must continue to compile and mean the same thing after any change to the standard.

The practical implication of this is that the committee (and the broader community) must live forever with any errors in the standard.[4] Any feature released in the standard will have code written that relies on that feature as specified. If that feature was specified incorrectly, correcting it in a subsequent standard would break existing code. In general, proposals that break existing code are nonstarters with the committee.

Making this even more difficult is the fact that for some features, real-world experience is the only good way to know the best way to specify the feature. To address the problem, the committee is beginning to use technical specifications (TS).[5] A TS is way of releasing a set of features (either core language, library, or both) that are considered standard-candidate material.

Tool vendors can implement a TS and provide it to users as a nonstandard extension for experimental use. This allows the committee to gather real-world user experience before adding the feature to the standard.

---

3 Currently handled directly by Library Evolution working group.

4 We are looking at you, `vector<bool>`.

5 The difference between an ISO Technical Report, of which the Committee released one in 2005, and an ISO Technical Specification, which will be used by the Committee going forward, is not very interesting. The TR1 should probably have been released as a TS.

How well does this process work? Well, we don't know yet.[6] In order for the TS approach to work, vendors must implement, and users must experiment with each TS released.[7] This seems very likely, but only one TS has been released,[8] and it is too early to know how this will work out.

Here are the technical specifications that are currently in pipeline:

File System
: Work based on Boost.Filesystem v3, including file and directory iteration.

Library Fundamentals
: A set of standard library extensions for vocabulary types like optional<> and other fundamental utilities.

Networking
: A small set of network-related libraries including support for network byte order transformation and URIs.

Concepts
: Extensions for template type checking.

Arrays
: Language and library extensions related to arrays, including runtime-sized arrays and `dynarray<>`.

Parallelism
: Initially includes a Parallel STL library with support for parallel algorithms to exploit multiple cores, and vectorizable algorithms to exploit CPU and other vector units.

Concurrency
: Initially includes library support for executors and non-blocking extensions to `std::future`. Additionally may include language extensions like `await`, and additional libraries such as concurrent hash containers and latches.

---

6 It worked well with the TR1 from 2005 that was incorporated into the 2011 release with some minor changes. But that was library-only, and almost all of it was already implemented and in wide usage as Boost libraries.

7 The approach will also fail if users embrace the TS in such a way that it becomes a de facto standard of its own.

8 File System

Transactional Memory
: A promising way to deal with mutable shared memory that is expected to be more usable and scalable than current techniques based on atomics and mutexes.

There is absolutely no guarantee that any of these will be in the 2017 standard.[9] But knowing that this is what the committee is working on gives us a sense of its priorities and ambition for the evolving standard for C++.

"Never Make Predictions, Especially About the Future" (Casey Stengel)

Stein's Law is that trends that can't continue won't.[10] The trick is to figure out which trends will continue indefinitely. Here are some that we see.

Performance

Mobile and cloud computing has rekindled the interest in performance, and we think performance will always be important. No computer will ever be powerful or energy efficient enough that performance won't matter, at least for some very important applications. This looks good for a language that has always been uncompromising in its approach to performance.

New Platforms

As the cost of hardware falls, more and more computing devices will be created. These new devices will mean new environments, some with very tight memory footprint requirements. This looks good for a highly portable systems language with a "you don't pay for what you don't use" approach to features.

Scale

At the top end, the falling cost of hardware will lead to the design and implementation of systems of a size that are difficult for us to imagine now. To implement these systems, engineers are going to

9 There is not even any guarantee that next standard will be released in 2017.

10 "If something cannot go on forever, it will stop." — Herbert Stein

look for a language that scales with additional users and teams and supports the creation of very large systems. Such a language will need to have high performance, but also support the high-level abstractions necessary to design systems at that scale. It will also need as much compiler-aided bug-catching as possible, which is heavily aided by an expressive type system that C++ supports.

Software Ubiquity

Our world is going to be more and more one in which we are surrounded by software. Will all of this software need to be highly portable, low-memory, high-performance code? Of course not. There will be great demand for applications that do not require software to be pushed to the limit. But these applications will always run on infrastructure where performance will be in demand. A lot of this infrastructure is currently written in C, but when infrastructure code requires high-level abstractions, that code is and will usually be written in C++.

It may be that the software industry as a whole will grow faster than the C++ community and that shrinking market share may make C++ appear to be less important. But the fact is that high performance infrastructure makes it possible to create applications in a less demanding programming environment. More programmers working in high-level, nonsystems languages just increases the demand for and value of the systems-programming projects that make their work possible.

Powerful Tools

The philosophy of C++ has been to rely more and more on a powerful compiler to do the heavy lifting of making high-performance applications. At times, that has pushed our compilers to the breaking point.[11] We think this is the correct direction for tool development: designing tools that let programmers focus on expressing their ideas as clearly as possible, and let the tools do the hard work to implement these ideas efficiently.

11 Early users of code that pushed the envelope on templates sometimes found that their compilers seemed to grind to a halt. Advances in compiler technology and computing power generally overcame this limitation.

We will see the language definition evolve toward making more demands on the compiler. We'll also see more and more creative tools built with the Clang toolkit.

The world of computing technology can change quickly, dramatically, and sometimes unexpectedly, but from where we sit, it looks like C++ is going to continue to play an important role for the foreseeable future.

Bibliography

[JIC] Allain, Alex. *Jumping into C++*. Cprogramming.com, 2013 (ISBN: 9780988927803)

[TCSL] Josuttis, Nicolai. *The C++ Standard Library: A Tutorial and Reference. 2nd ed.* Addison-Wesley Professional (ISBN: 9780201543308)

[CP] Lippman, Stanley, Josée Lajoie, Barbara Moo. *C++ Primer. 5th ed*. Addison-Wesley Professional, 2012 (ISBN: 9780321714114)

[EMC] Meyers, Scott. *Effective Modern C++*. O'Reilly Media Inc., 2014 (ISBN: 9781491903995)

[FM2G] Stepanov A. A. and D. E. Rose. *From Mathematics to Generic Programming*. Pearson Education, 2014 (ISBN: 9780133491784)

[DEC] Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley Professional, 1994 (ISBN: 9780201543308)

[TCPL] Stroustrup, Bjarne. *The C++ Programming Language. 4th ed.* Addison-Wesley Professional, 2013 (ISBN: 9780321563842)

[PPPUC] Stroustrup, Bjarne. *Programming: Principles and Practice Using C++. 2nd ed.* Addison-Wesley Professional, 2014 (ISBN: 9780321992789)

[ATOC] Stroustrup, Bjarne. *A Tour of C++* Addison-Wesley Professional, 2013 (ISBN: 9780321958310)

[CCIA] Williams, Anthony. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, 2012 (ISBN: 9781933988771)

About the Authors

Jon Kalb does on-site training on C++ best practices and advanced topics. He is an Approved Outside Training Vendor for Scott Meyers' training materials and is an award-winning conference speaker. For information on course content, dates, and rates, please email *jon@cpp.training*.

Jon has been programming in C++ for two and a half decades. He is currently working on Amazon's search engine at A9.com. During the last 25 years, he has written C++ for Amazon, Apple, Dow Chemical, Intuit, Lotus, Microsoft, Netscape, Sun, Yahoo!, and a number of companies that you've not heard of.

Gašper Ažman is an undercover mathematician masquerading as a software engineer. On his quest to express ideas precisely, concisely, and with great care for simplicity, he likes to study emerging programming languages for new tricks to apply in his C++. He is currently taking a hiatus from teaching to work on the Amazon search engine at A9.com. In his free time, he makes music and bread.

About the Cover

The image on the cover is a Japanese artist's[12] illustration of an ancient Chinese legend about an Old Master who asked his disciples to describe a language that he gave them. The first student said, this language is an improvement on portable assembler. The next student said, this is a language for constructing beautiful libraries. Another student said, no, this language is for constructing hierarchies of objects. No, said the next student, this language is for expressing mathematical functions in the real world. You've missed its power, said another, it is for expressing generic truths about any type...

And as the students argued on, the Old Master smiled to himself.

12 Hanabusa Itchō