Thriving in a Crowded and Changing World: C++ 2006–2020

BJARNE STROUSTRUP, Morgan Stanley and Columbia University, USA

Shepherd: Yannis Smaragdakis, University of Athens, Greece

By 2006, C++ had been in widespread industrial use for 20 years. It contained parts that had survived unchanged since introduced into C in the early 1970s as well as features that were novel in the early 2000s. From 2006 to 2020, the C++ developer community grew from about 3 million to about 4.5 million. It was a period where new programming models emerged, hardware architectures evolved, new application domains gained massive importance, and quite a few well-financed and professionally marketed languages fought for dominance. How did C++ – an older language without serious commercial backing – manage to thrive in the face of all that?

This paper focuses on the major changes to the ISO C++ standard for the 2011, 2014, 2017, and 2020 revisions. The standard library is about 3/4 of the C++20 standard, but this paper's primary focus is on language features and the programming techniques they support.

The paper contains long lists of features documenting the growth of C++. Significant technical points are discussed and illustrated with short code fragments. In addition, it presents some failed proposals and the discussions that led to their failure. It offers a perspective on the bewildering flow of facts and features across the years. The emphasis is on the ideas, people, and processes that shaped the language.

Themes include efforts to preserve the essence of C++ through evolutionary changes, to simplify its use, to improve support for generic programming, to better support compile-time programming, to extend support for concurrency and parallel programming, and to maintain stable support for decades' old code.

The ISO C++ standard evolves through a consensus process. Inevitably, there is competition among proposals and clashes (usually polite ones) over direction, design philosophies, and principles. The committee is now larger and more active than ever, with as many as 250 people turning up to week-long meetings three times a year and many more taking part electronically. We try (not always successfully) to mitigate the effects of design by committee, bureaucratic paralysis, and excessive enthusiasm for a variety of language fashions.

Specific language-technical topics include the memory model, concurrency and parallelism, compile-time computation, move-semantics, exceptions, lambda expressions, and modules. Designing a mechanism for specifying a template's requirements on its arguments that is sufficiently flexible and precise yet doesn't impose run-time costs turned out to be hard. The repeated attempts to design "concepts" to do that have their roots back in the 1980s and touch upon many key design issues for C++ and for generic programming.

The description is based on personal participation in the key events and design decisions, backed by the thousands of papers and hundreds of meeting minutes in the ISO C++ standards committee's archives.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **General programming languages**;

Additional Key Words and Phrases: C++, programming language design and evolution, standardization, generic programming, resource management, concurrency and parallelism, simplification of language use

ACM Reference Format:
Bjarne Stroustrup. 2020. Thriving in a Crowded and Changing World: C++ 2006–2020. *Proc. ACM Program. Lang.* 4, HOPL, Article 70 (June 2020), 167 pages. https://doi.org/10.1145/3386320

Author's address: Bjarne Stroustrup, Morgan Stanley and Columbia University, USA, Bjarne@Stroustrup.com.

2475-1421/2020/6-ART70
https://doi.org/10.1145/3386320

Contents

1 INTRODUCTION

Originally, I designed C++ to answer to the question "How do you directly manipulate hardware and also support efficient high-level abstraction?" Over the years, C++ has grown from a relatively simple solution based on a combination of facilities from the C and Simula languages aimed at systems programming on 1980s computers to a far more complex and effective tool for an extraordinary range of applications. It retains its dual focus on:

- *Direct mapping of language constructs to hardware facilities*
- *Zero-overhead abstraction*

This combination is the defining characteristic that sets C++ apart from most languages.
"Zero overhead" was explained like this [Stroustrup 1994]:

- *What you don't use, you don't pay for (aka "no distributed fat").*
- *What you do use, you couldn't hand-code any better.*

Abstractions are represented in code as functions, classes, templates, concepts, and aliases.

C++ is a living language, so it changes to meet new challenges and the styles of use evolve. These challenges and changes in the 2006-to-2020 time-frame are the focus of this paper. Of course, a language doesn't change by itself; it is changed by people. So this is also the story of the people involved in the evolution of C++, the way they perceived the challenges, interpreted the constraints on solutions, organized their work, and resolved their inevitable differences. When presenting a language or standard-library feature, I do so in the context of the general evolution of C++ and the concerns of the individuals involved at the time. For many features accepted early in the time period, we now have the benefit of hindsight from massive industrial use.

C++ is primarily an industrial language, a tool for building systems. For a user, "C++" is not just a language as defined by a specification; it is part of a tool set with many parts:

- The language
- The standard library
- Many other libraries
- Massive – often old – code bases
- Tools (including other languages)
- Teaching and training
- Community support

Where possible and relevant, I will consider the interactions among those "parts."

There is a myth, a very popular myth, that programmers want their languages to be simple. That's obviously the case when you have to learn a new language, have to design a programming course or curriculum, or describe a language in an academic paper. For such uses, having a language cleanly embody a few clear principles is an obvious advantage and the ideal. When the focus shifts from learning to delivering and maintaining significant applications, the demands from developers shift from simplicity to comprehensive support, stability (compatibility), and familiarity. People invariably confuse familiarity with simplicity and prefer familiarity over simplicity if given a choice.

One way of looking at C++ is as the result of decades of three contradictory demands:

- *Make the language simpler!*
- *Add these two essential features now!!*
- *Don't break (any of) my code!!!*

I added the exclamation marks because these points are often delivered with a fair bit of emotion.

I wanted to make simple things simple and to ensure that complex things are not impossible or unnecessarily hard. The former is essential for developers who are not language lawyers; the later

for implementers of foundational code. Stability is an essential property for all systems meant to last for decades, yet a living language must adapt to a changing world.

There are overarching ideals for C++. I articulate some (e.g., *The Design and Evolution of C++* [Stroustrup 1994] (§2), design principles (§9.1), and the C++ model (§11.1)) and try to make the evolution of the language follow them. However, long lists of new features and very detailed practical concerns are the main focus of C++'s development as controlled by its ISO standards committee. That's what the most vocal and influential people in the community insist on, and it would be foolhardy to deem their concerns and opinions wrong solely based on philosophical or theoretical views.

1.1 Chronology

To give a quick overview, here is a rough chronology. If you are not familiar with C++, many of the terms, construct, and libraries will be obscure; most are explained in length in the previous HOPL papers [Stroustrup 1993, 2007] or in this paper.

- 1979: Start of work on "C with Classes" that became C++; first non-research user;
  - Language: **class**es, constructors/destructors, **public**/**private**, simple inheritance, function argument type checking
  - Library: **task**s (coroutines and simulation support), **vector** parameterized with macros
- 1985: First commercial release of C++; TC++PL1 [Stroustrup 1985b]
  - Language: **virtual** functions, operator overloading, references, **const**
  - Library: **complex** arithmetic, stream I/O
- 1989-91: ANSI and ISO standardization start; TC++PL2 [Stroustrup 1991]
  - Language: abstract classes, multiple inheritance, exceptions, templates
  - Library: iostreams (but no **task**s)
- 1998: C++98, the first ISO C++ standard [Koenig 1998], TC++PL3 [Stroustrup 1997]
  - Language: **namespace**s, named casts, **bool**, **dynamic_cast**
  - Library: the STL (containers and algorithms), **string**, **bitset**
- 2011: C++11 [Becker 2011], TC++PL4 [Stroustrup 2013]
  - Language: memory model, **auto**, range-**for**, **constexpr**, lambdas, user-defined literals, ...
  - Library: **thread**s and locks, **future**, **unique_ptr**, **shared_ptr**, **array**, time and clocks, random numbers, unordered containers (hash tables), ...
- 2014: C++14 [du Toit 2014]
  - Language: generic lambdas, local variables in **constexpr** functions, digit separators, ...
  - Library: user-defined literals, ...
- 2017: C++17 [Smith 2017]
  - Language: structured bindings, variable templates, template argument deduction from constructors, ...
  - Library: file system, **scoped_lock**, **shared_mutex** (reader-writer locks), **any**, **variant**, **optional**, **string_view**, parallel algorithms, ...
- 2020: C++20 [Smith 2020]
  - Language: **concept**s, **module**s, coroutines, three-way comparisons, improved support for compile-time computation, ...
  - Library: concepts, ranges, dates and time zones, **span**, formats, improved concurrency and parallelism support, ...

Note the poverty of libraries in the early years. There were, in fact, many libraries (including GUI libraries), but few were widely used and many were proprietary. This was before open-source development became widespread. This left the C++ community without a significant shared

foundation library. In the retrospective of my HOPL2 paper [Stroustrup 1993], I deemed that the worst mistake of early C++.

The task library [Stroustrup 1985a,c] was a coroutine-based library with some support for event driven simulations (e.g., random number generation) that was very efficient when compared to alternatives, even on tiny computers. For example, I ran simulations with 700 tasks in a 256KB memory. The task library was immensely important in C++'s early years as the base of many important applications in Bell Labs and elsewhere. However, it was a bit ugly and couldn't easily be ported to Sun's SPARC architecture so it wasn't supported by most post-1989 implementations. In 2020, coroutines are only just coming back (§9.3.2).

Basically, the feature set grows contiguously. The ISO committee deprecated a few features in attempts to clean up the language but given the massive use of C++ ("many billions of lines of code"), nothing significant ever goes away. Stability is a key feature. One way of addressing problems related to the growing size and complexity is through coding guidelines (§10.6).

1.2 Overview

The paper is organized in rough chronological order around the sequence of ISO standard releases.

- §1: Introduction
- §2: Background: C++ 1979-2006
- §3: The C++ standards committee
- §4: C++11: It feels like a new language
- §5: C++14: Completing C++11
- §6: Concepts
- §7: Error handling
- §8: C++17: Lost at sea
- §9: C++20: A struggle for direction
- §10: C++ in 2020
- §11: Retrospective

Where a topic, such as "concepts" and the standards process, spans a longer period of time, I cover it in one place, giving the contents priority over the chronological format.

This paper is extraordinarily long, a monograph really. However, from 2006 to 2020, C++ went through two major revisions, C++11 and C++20, and early readers all requested more information; that led to almost doubling the page count. Even at the current size, readers will find important topics, such as concurrency and the standard library, underrepresented.

2 BACKGROUND: C++ 1979–2006

The history of C++ from 1979 to 2006 is documented in my HOPL papers [Stroustrup 1993, 2007]. In that period, C++ grew from a one-person research project to a community of about 3 million programmers.

2.1 The First Decade

The work on what became C++ started in April 1979 under the name of *C with Classes*. I wanted a tool that combined the ability to deal directly and efficiently with hardware (e.g., to write memory managers, process schedulers, and device drivers) with Simula-like facilities for organizing code (e.g., "strong" static extensible type checking, classes, class hierarchies, and coroutines). I wanted that tool to write a version of the Unix kernel that could work on multiple processors connected through a local area network or a shared memory.

I chose C as a base for my work because it was good enough and very well supported locally: my office was just across the corridor from Dennis Ritchie's and Brian Kernighan's. However, C wasn't the only language I considered. I was very attracted by Algol68 and was at the time quite expert at BCPL and a few other machine-level languages. The later success of C was at that time nowhere near certain, but Brian Kernighan and Dennis Ritchie's superb introduction and manual [Kernighan and Ritchie 1978] had just appeared and Unix was starting its victory run.

The initial implementation was a preprocessor that translated "C with Classes" into C more-or-less line-for-line. In 1982, that approach proved unmanageable as the "C with Classes" user population grew to several dozen people. So I wrote a conventional compiler, called *Cfront*, that was first used by others in October 1983. Cfront was a traditional compiler in that it had a lexical analyzer, a syntax analyzer that built an abstract syntax tree, a type checker that decorated that tree with types, and a high-level optimizer that rearranged the AST to improve the run-time efficiency of the generated code. There has been a lot of confusion about the nature of Cfront because it then finally output C (optimized and not particularly human-readable C). I generated C so that I did not have to directly deal with the myriad of (non-standardized) linkers and optimizers then in current use. Cfront was nothing like a traditional preprocessor, though. You can find a Cfront source with documentation in the Computer History Museum's source code collection [McJones 2007–2020]. Cfront was bootstrapped to C++ from C with Classes, so the first C++ compiler was written in (simple) C++ for tiny computers (less than 1MB of memory and less than 1MHz of processor speed).

The first feature added to C for "C with Classes" was classes. I knew their power from earlier use in Simula, where they were key to the strict static, but extensible type system. I immediately added constructors and destructors. They were novel, but from my machine architecture and operating systems background, I considered it obvious that I needed a mechanism to set up a working environment (a constructor) and an inverse operation to release resources acquired while running (a destructor). From my 1979 lab book:

- *A "new function" creates the run-time environment for member functions*
- *A "delete function" reverses that*

The terms "new function" and "delete function" were the original terms for "constructor" and "destructor." To this day, I consider constructor/destructor pairs the real heart of C++. See also (§2.2.1) and (§10.6).

At the time, essentially every language except C had proper function-argument type checking. I didn't think I could do anything significant without that. So, with the encouragement of my department head, Alexander Fraser, I immediately added (optional) function argument declarations and argument checking. That's what in C is now called function prototypes. In 1982, after seeing the effects of leaving the function argument checking optional, I made it compulsory. That caused a decade or two's loud howls of complaints about incompatibility with C. People wanted their type errors, or at least many loudly said they didn't want checking and used that as an excuse for not using C++. This factoid may give people an idea of the problems involved in evolving a language in significant use.

Given the occasional nasty words exchanged between overly parochial C and C++ aficionados, it may be worth pointing out that I was always friends with Dennis Ritchie and Brian Kernighan, eating lunch with them most days for 16 years. I learned a lot from them and still see Brian regularly. I credit both with contributions to C++ [Stroustrup 1993] and I am a major contributor to C myself (e.g., the function definition syntax, function prototypes, **const**, and //-comments).

To be able to think rationally about the growth of C++, I devised a set of design rules. These are featured in [Stroustrup 1993, 1994], so here I will just mention a small sample:

- *Don't get involved in a sterile quest for perfection.*
- *Always provide a transition path.*
- *Say what you mean (i.e., enable direct expression of higher-level ideas) .*
- *No implicit violations of the static type system.*
- *Provide as good support for user-defined types as for built-in types.*
- *Preprocessor usage should be eliminated.*
- *Leave no room for a lower-level language below C++ (except assembler).*

These were not unambitious goals. Some, I am still working on in 2020. In the early-to-mid-1980s, I added more language facilities to C++:

- *1981*: **const** – to support immutability in interfaces and symbolic constants.
- *1982*: virtual functions – to offer run-time polymorphism.
- *1984*: References – to support operator overloading and simplify argument passing.
- *1984*: Operator and function overloading – including allowing the user to define = (assignment), **()** (application; enabling "function objects" (§4.3.1)), **[]** (subscripting), and -> (smart pointers) in addition to the arithmetic and logical operators.
- *1987*: Type-safe linkage – to eliminate many errors coming from inconsistent declarations in separate translation units.
- *1987*: abstract classes - to offer pure interfaces.

In the late 1980s, as the power of computers increased dramatically, I got more interested in larger-scale software and added

- Templates – to better support generic programming after years of suffering with writing generic programming using macros.
- Exceptions – to try to bring some order to the chaos of error-handling; RAII (§2.2.1) was articulated for that design.

These later facilities were not universally well received (e.g., see (§7)). Part of the reason was that the community had grown large and unmanageable. The ANSI standardization had started so I was no longer able to implement and experiment in private. People insisted on large elaborate designs and on debating them extensively before serious implementation. I could no longer start with a minimal proposal and grow it into a more complete facility while knowing that it isn't possible to please everybody. For example, people insisted on the "heavy" template syntax with the **template<class T>** prefix everywhere.

In the late 1980s, the "object-oriented" hype became deafening and stole the message of C++ from me. My opinion of what C++ was and was meant to become was widely ignored – many never heard it. All new languages were to be "pure object-oriented," for some definition of "object oriented." Not being "truly OO" was deemed bad without the need for argument.

The fact that I never used the phrase "C++ is an object-oriented programming language" was not known or ignored as a bit embarrassing. At the time, my standard description was

> *C++ is a general-purpose programming language with a bias towards systems programming that*
> - *is a better C*
> - *supports data abstraction*
> - *supports object-oriented programming*
> - *supports generic programming*

This was (and is) accurate, but not as exciting as slogans such as "Everything is an object!"

2.2 The Second Decade

The ANSI C++ committee was founded at a meeting in Washington D.C. in December of 1989, just over 10 years after the first beginnings of "C with Classes." About 25 C++ programmers were present. I was there, as were a small handful ISO C++ standard committee members who were still active in the 2010s.

The committee delivered its first standard, C++98, after the conventional about a decade's work. Naturally, I – and many others – would have preferred a standard sooner, but committee rules, over-ambition, and various delays brought us into line with the schedules of Fortran, C, and other formally standardized languages.

The work that led to C++98 is the core of the HOPL3 paper [Stroustrup 2007], so here I only briefly summarize.

2.2.1 *Language Features.*

The major language features of C++98 were:

- Templates – unconstrained, Turing-complete, compile-time support for generic programming following up on my early work (§2.1) with many elaborations and refinements; that work continues (§6).
- Exceptions – a mechanism for returning error-values on a separate ("invisible") path to be handled by code "elsewhere" up the stack of callers; see (§7).
- **dynamic_cast** and **typeid** – a very simple form of run-time reflection ("Run-time Type Identification" aka RTTI).
- **namespace**s – allowing programmers to avoid name clashes when composing larger programs out of separate parts.
- Declarations in conditions – tightening up notation and limiting scope of variables.
- Named casts (**static_cast**, **reinterpret_cast**, and **const_cast**) – eliminating ambiguities from C-style casts and making explicit type conversion far more visible.
- **bool** – a Boolean type that proved surprisingly useful and popular; C and C++ had used integers as Boolean variables and constants.

Consider a simple C++98 example. The **dynamic_cast** is C++'s version of what is often called something like "isKindOf" in object-oriented languages:

```
void do_something(Shape* p)
{
    if (Circle* pc = dynamic_cast<Circle*>(p)) {  // is p a kind of Circle?
        // ... use the Circle pointed to by pc ...
    }
    else {
        // ... it wasn't a Circle, do something else ...
    }
}
```

The **dynamic_cast** is a run-time operation relying on data stored in the **Shape**'s virtual function table. It is general, easy to use, and about as efficient as equivalent facilities in other languages. However, **dynamic_cast** became quite unpopular because its implementations tended to be complicated and special cases can be hand-code more efficiently (so that **dynamic_cast** arguably violated the zero-overhead principle). The use of declarations in conditions was novel, though at the time, I thought that I had just copied the idea from Algol68.

A simpler variant uses references instead of pointers:

```
void do_something2(Shape& r)
{
    Circle& rc = dynamic_cast<Circle&>(r);          // r is a kind of Circle!
    // ... use the Circle referred to by rc ...
}
```

This simply asserts that **r** refers to a **Circle** and throws an exception if it does not. The idea was to use pointers and tests if the "error" could reasonably be handled locally and to rely on references and exceptions if not.

One of the most important techniques in C++98 was RAII (*Resource Acquisition Is initialization*). That was my clumsy name for the idea that every resource should have an owner represented by a scoped object: A constructor acquires the resource and a destructor implicitly releases it. This idea was present in the earliest C with Classes (§2), but not named until a decade later. Here is an example I often used to illustrate the idea that not every resource is memory:

```
void my_fct(const char* name)    // C-style resource management
{
    FILE* p = fopen(name,"r");    // open File 'name' for reading
    // ... use p ...
    fclose(p);
}
```

The problem here is that if (between the calls of **fopen()** and **fclose()**) we **return** from the function, **throw** an exception, or use C's **longjmp**, the file handle pointed to by **p** is leaked. Leaking file handles exhausts an operating system even faster than memory leaks. That file handle is an example of a *non-memory resource*.

The solution is to represent the file handle as a class with a constructor and a destructor:

```
class File_handle {
    FILE* p;
public:
    File_handle(const char* name, const char* permissions); // open file
    ~File_handle(); // close file
    // ...
};
```

We can now simplify our use:

```
void my_fct2(const char* name)   // RAII-style resource management
{
    File_handle p(name,"r");    // open File 'name' for reading
    // ... use p ...
} // p is implicitly closed
```

With the introduction of exceptions, such resource handles became pervasive. In particular, the standard-library file stream is such a resource handle, so using the C++98 standard library, this example becomes:

```
void my_fct3(const string& name)
{
    ifstream p(name);   // open File 'name' for reading
    // ... use p ...
} // p is implicitly closed
```

Note that the RAII code differs from the traditional use of functions by allowing the "cleanup" to be defined once-and-for-all in the library, rather than having to be remembered and explicitly written by the programmer for each use of a resource. It is critical that the correct and robust code is simpler, shorter, and at least as efficient as the conventional style. Over the next 20 years, RAII permeated C++ libraries.

The implication of having non-memory resources is that garbage collection isn't by itself sufficient for resource management. In addition, RAII plus smart pointers (§4.2.4) eliminate much of the need for Garbage collection. See also (§10.6).

2.2.2 Standard-Library Components. The C++98 standard-library provided:

- The STL – the innovative, general, elegant, and efficient framework of containers, iterators, and algorithms by Alexander Stepanov.
- Traits – sets of compile time properties useful for programming with templates (§4.5.1).
- **string** – a type for holding and manipulating a sequence of characters. The character type is a template parameter defaulted to **char**.
- **iostream**s – an elaboration by Jerry Schwartz and the standards committee of my simple 1984 streams library to handle a wide variety of character types, locales, and buffering strategies.
- **bitset** – a type for holding and manipulating sets of bits.
- **locale**s – an elaborate framework of cultural conventions, mostly related to I/O.
- **valarray** – a numeric array with optimizable vector operations that unfortunately didn't see much use.
- **auto_ptr** – an early pointer representing exclusive ownership; in C++11, it was replaced by **shared_ptr** (for shared ownership) and **unique_ptr** (for exclusive ownership) (§4.2.4).

The STL framework was by far the most important standard-library component. I think it fair to say that it – and the generic programming techniques it pioneered – saved C++ as a living modern language. Like all the C++98 facilities, the STL has been extensively described elsewhere (e.g., [Stroustrup 1997, 2007]) so here I will present just a single, short example:

```
void test(vector<string>& v, list<int>& lst)
{
    vector<string>::iterator p
         = find_if(v.begin(), v.end(), Less_than<string>("falcon"));
    if (p != v.end()) {    // p points to 'falcon'
        // ... use *p ...
    }
    else {                 // 'falcon' not found
        // ...
    }

    vector<int>::iterator q
         = find_if(lst.begin(), lst.end(), Greater_than<int>(42));
    // ...
}
```

The standard-library algorithm **find_if** traverses a sequence (delimited by a **begin**/**end** pair) looking for an element for which a predicate is true. The algorithm is generic in three dimensions:

- The way elements of the sequence are stored (here, **vector** and **list**)
- The type of elements (here, **string** and **int**)
- The predicate used to determine when an element is found (here, **Less_than** and **Greater_than**)

Note the absence of object-oriented techniques. This is generic programming relying on templates, sometimes referred to as *compile-time polymorphism*.

The notation was still primitive, but from about 2017, I could use **auto** (§4.2.1), ranges (§9.3.5), and lambdas (§4.3.1) to simplify that code:

```
void test2(vector<string>& v, list<int>& lst)
{
    if (auto p = find_if(v,[](const string& s) { return s<"falcon"; })) {
        // ...
    }
    // ...
    if (auto q = find_if(lst,[](int x) { return x>42; })) {
        // ...
    }
    // ...
}
```

2.3 C++ in 2006

In 2006, I and most other members of the ISO C++ committee had high hopes for a feature-rich C++0x standard. A feature freeze was planned for 2007, so we had a reasonable expectation that C++0x would be C++08 or C++09. In fact, C++0x became C++11, causing jokes about the hexadecimal C++0xB.

In my 2006 HOPL paper [Stroustrup 2007], I listed 39 proposals and predicted that the first 21 would make it into C++0x. Interestingly, 24 of the first 25 proposals on my list made it into C++11. Proposals 22-25, I listed as "being developed aiming for votes in 2007." To my surprise – and immense pleasure – they all made it. None of proposals 26-39 even made it into C++17. That leaves proposal 10, "concepts", which has its own long sad story with a happy ending in C++20 (§6).

I and many others were frustrated by the delays of C++0x and feared that an unimproved C++ might not survive as a living language in the face of competition from more modern and better financed alternatives. In 2006, Java use was still increasing and Microsoft's C# was heavily supported and marketed. My estimate in 2006 was that C++ use had – for the first time – declined slightly over the previous 4 years. It is hard to obtain real numbers and my best estimate (a 7% decline) is well within the error margins, but there was certainly reason to worry. Languages, such as Java and C#, were based on the assumption – often loudly proclaimed – that C++ didn't have an ecological niche:

- "Low-level stuff" could be handled by a small amount of C or assembler.
- "High-level stuff" was better, cheaper, and more efficiently done in a safer, smaller, garbage-collected language with a huge run-time support system.
- Managed languages, such as Java and C#, using garbage collection and consistent runtime range checking, made less-expert programmers more productive and highly-skilled developers less needed.
- Deep integration of a programming language into a platform and supported by an integrated tool set were essential for productivity and the construction of large systems.

Obviously, I and many others didn't agree, but these were (and are) serious arguments, that (if correct) should lead to the abandonment of C++. C++ is based on the traditional model of a programming language separate from the underlying operating system and supported by a multitude of independent tool suppliers. The managed languages tended to be proprietary; only a large and rich organization could develop the massive infrastructure and libraries required. I and

many others in the C++ community prefer languages to be free of corporate control; this was one reason I took part in the ISO standards effort.

In retrospect, 2006 might have been the nadir for C++, but important technological events had just happened: For the first time in history, in about 2005, single-processor (single-core) performance stopped improving and energy efficiency ("Performance per watt') became a key measure (especially for server farms and hand-held devices). The economics of computing shifted to favor better software. No longer could inefficiencies in languages or programming techniques be completely hidden by hardware advances. Now, a highly skilled developer using a "sharp tool" could (again, after a decade or so) gain an economic order-of-magnitude advantage over weaker programmers or programmers hobbled by overheads in their tool chains. Even today, these facts have not yet worked their way through all the educational and management systems, but there are now many significant tasks for which spending time on carefully crafting performant code pays off massively.

Another turning point came from vendors trying to impose their favorite language on all users by defining standard interfaces to, say GUI, that could be met only by using their favored – and often proprietary – languages. Examples were Google's use of Java for Android, Apple's Objective-C for iOS, and Microsoft's C# for Windows. Application vendors could try to dodge the lock-in by using dialects, such as Objective C++ [Objective C++ Wikipedia 2020] or C++/CLI [ECMA International 2005], but the resulting code was still not portable. Many organizations, such as Adobe, Google, and Microsoft, responded by writing the major parts of their demanding applications in C++ and then using thin interface layers for the various platforms (e.g., Android, iOS, and Windows). In 2006, this trend was barely noticeable.

On portable devices (in particular, smartphones), the need for energy efficiency and platform independence combined. One effect is that by my best estimates in 2018, the number of C++ programmers was up about 50% since 2006 to about 4.5 million developers [Kazakova 2015]. That's a 150,000 developers/year increase; about 4%/year for a decade.

In 2006, few people had spotted the significant hardware trends feeding into C++'s inherent strengths. Instead, the community and the standards committee were focusing on novel language features and libraries to increase C++'s usefulness and to raise some enthusiasm. Some committee members, including me, felt an urgency and a dire need for significant improvements. Others were more focused on stabilizing the language and improving its implementations. A standards committee needs both groups, but the constant tug of war between innovation and retrenchment is a source of tension. As in any large organization, there is an organizational advantage to people defending status quo and serving current users. In *The C++ Programming Language (3rd Edition)* [Stroustrup 1997], I cited Niccolò Machiavelli on that topic:

> *"there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer makes enemies of all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order."*

My opinion was that C++ needed significant improvement to serve its community well. C++ applications were massively deployed, but new projects often chose more fashionable languages and some successful C++ projects were being rewritten into such languages. For example, much of Google's large-scale applications, such as search, was (and is) based on their map-reduce framework [Dean and Ghemawat 2004, 2008]. That's a C++ program. However, as it is proprietary for commercial reasons, people replicated it, but – sadly for the C++ community – the open-source map-reduce framework (Hadoop) was for a variety of reasons done in Java.

Another significant reason for development moving to other languages is that the flexibility of interfaces offered by templates makes it extremely difficult to provide a stable ABI using all C++ features: you can be flexible or you can offer stable binary interfaces, but it is beyond most organizations to do both. I consider this a contributing reason that people require C, Java, C#, etc., interfaces to programs written in C++. ABI stability for C++ is a genuinely hard technical problem, especially as the C++ standard has to be platform independent.

To add to the problems of the C++ community, by 2006, most professional software magazines covering C++ had died as publishing on paper was declining and journalists followed fashion and advertising revenues. Dr. Dobbs Journal lasted another few years (stopped in print in February 2009). The C++ conferences were being absorbed into "Object-oriented" or general software development conferences, depriving the C++ community of venues for exposure of new developments. Books were still being written, but programmers were reading fewer books (or at least buying far fewer as pirating was becoming easier and consequently statistics were becoming less reliable) and on-line sources were becoming more popular.

An even more serious problem was that C++'s role in education was sharply decreasing. C++ was no longer "new and interesting" and Java was being marketed directly to universities as an easier and more powerful language. The US high-school Computer Science test suddenly changed from C++ to Java. The use of Java as the introductory language in universities increased dramatically. The quality of C++ teaching was also decreasing with most courses choosing a C-first approach or taking the view that Object-Oriented programming relying heavily on class hierarchies was the one true way. Both approaches minimized C++ strengths and required heavy use of macros. The standard library (relying on generic programming; (§2.2)) and RAII (relying on constructor/destructor pairs (§2.2.1)) were often completely left out of foundational courses or delegated to an "advanced features" section that most students either never reached or considered scary. Textbooks often bogged down in obscure details. There were exceptions, of course, but on average the C++ presented to students was far inferior to the best industrial practices. In 2005, I accepted the challenge to teach programming to first-year university students. I surveyed about twenty of the most popular C++ programming textbooks and ended up loudly complaining:

> "*If that's C++, I don't like it either!*"

After teaching a year using a well-reputed textbook, I changed to using just my own notes, and in 2008 published *Programming: Principles and Practice using C++* [Stroustrup 2008a] but to this day, much C++ teaching has a 1980s flavor.

Despite this, C++ usage was starting to increase again. I think the reason was the fundamental technology trends again favored C++ and towards the end of the decade the beginnings of C++11 were helping.

The Boost libraries and the Boost organization were important [Boost 1998–2020]. In 1998, Beman Dawes, an experienced developer and influential member of WG21, had started a "C++ library repository web site" [Dawes 1998] with the explicit aim of developing C++ libraries to establish existing practice that future standardization could build upon. Before that, C++ had never even had a common repository for libraries. Boost slowly grew into an active organization with peer review of new libraries and a yearly conference. The Boost libraries became very widely used and the most popular were absorbed into the standard (e.g., **regex** (§4.6), **thread** (§4.1.2), **shared_ptr** (§4.6), **variant** (§8.3), and file system (§8.6)). It was important for the C++ community that the Boost libraries were available more than a decade earlier than their ISO standard versions but were trusted as a kind of "junior standard." Many committee members, notably, Dave Abrahams, Doug Gregor, Jaakko Järvi, Andrew Sutton, and of course Beman Dawes, were involved with Boost.

By 2006, C++ was no longer new and exciting in industry, but spread over many industries. Usage was – and still is – strong in the telecommunications industry where C++ was born. From there, it had spread into the games (e.g., Unreal, PlayStation, Xbox, and Douglas Adams' "Spaceship Titanic"), finance (e.g., Morgan Stanley and Renaissance Technologies), microelectronics (e.g., Intel and Mentor Graphics), movies (e.g., Pixar and Maya), aerospace (e.g., Lockheed-Martin and NASA), and many other industries.

Personally, I was particularly fond of C++'s widespread use in science and engineering, such as High Energy Physics (e.g., CERN, SLAC, FermiLab), biology (e.g., the human genome project), space exploration (e.g., Mars Rovers and the deep space communication network), medicine and biology (e.g., tomography, general imaging, the human genome project, and monitoring equipment), and much more.

2.4 Other Languages

People often look for direct technical influences from other programming languages on C++. There are few. Typically, influences bubble up through history from common ancestors and shared ideas. Decisive arguments for extending C++ tend to relate to observed problems in the C++ community. Direct borrowing from a fashionable language is rare and far more difficult than people imagine. Most members of the standards committee master many languages and keep an eye out for useful facilities, libraries, and techniques.

Consider some real and conjectured influences on C++ in the 2000s:

- **auto** – the ability to deduce a type from an initializer. This is popular in modern languages, but old as the mountains. I don't really know where it came from, but I implemented it in 1983 and didn't consider it novel then (§4.2.1).
- **tuple** – many languages, especially from the functional programming tradition have tuples, usually as a built-in type. The C++ standard-library **tuple** and many of its uses are inspired from those. The **std::tuple** was derived from the **boost::tuple** [Boost 1998–2020] (§4.3.4).
- **regex** – The standard library **regex** added in C++11 was copied (via Boost with proper acknowledgements) from facilities in Unix and JavaScript (§4.6).
- Functional programming – there are many obvious similarities between FP features and C++ constructs. Most are not simple language features, but programming techniques. The STL was inspired by functional programming and first tried (unsuccessfully) in Scheme [Stepanov 1986] and Ada [Musser and Stepanov 1987].
- **future** and **promise** – from Multilisp via other Lisp dialects (§4.1.3).
- Range-**for** – it has equivalents in many languages, but one direct inspiration was STL sequences (§4.2.2).
- **variant**, **any**, and **optional** – clearly inspired by a variety of languages (§8.3).
- Lambdas – clearly the use of lambda expressions in functional languages was part of the inspiration. However, in C++, the roots of lambdas also include blocks of code used as expressions going back to BCPL, local functions (which had been repeatedly rejected for C and C++ because they were seen as error-prone and adding complexity), and (most importantly) function objects (§4.3.1).
- **final** and **override** – for more explicit management of class hierarchies and present in many object-oriented languages. They had been considered – and considered unnecessary – from the earliest days of C++.
- The three-way comparison operator, <=>, was inspired by C's **strcmp** and operators in various languages, including PERL, PHP, Python, and Ruby (§9.3.4).

- **await** – the original C++ coroutines (§1.1) were inspired by uses of Simula, but offered as a library rather than as a language feature to leave room for several alternative ways of expressing concurrency. The idea for the C++20 stackless coroutines came primarily from F# (§9.3.2).

Even when a feature is borrowed from another language in a pretty direct manner, it mutates. Typically, the syntax changes quite a bit to fit into C++. When borrowing from a garbage-collected language, lifetime issues must be dealt with, and often the C++ distinction between objects and references to objects needs to be addressed in ways that differ from the original. Often novel uses are discovered during the "translation" into C++. The story of the introduction of "lambdas" into C++ offers examples of most of these phenomena (§4.3.1).

Many people imagine that I (and others involved with C++) sit around all day strategizing in a complicated war for dominance among popular languages. In fact, I spend no time on such. Most days, I don't think about other languages except when I happen to study one out of general technical interest or use one to get some work done. What I do is to talk with software developers, consider problems that people encounter using C++, and consider the flood of suggested improvements in the standards committee. Of course, I also write code to experience problems and test out ideas for improvements. The problem is to find time to calmly consider what's fundamental, what's just fashion, and what would do harm.

Similarly, C++'s contributions to other languages are hard to pinpoint. Often, similar features are parallel evolution or have common roots. Consider:

- Generics in Java and C# – they modeled their generics from other languages but took the C++ syntax and added generics only after C++ demonstrated the utility of generic programming on a large scale.
- The dispose idiom in Java, Python, etc. – that is about the best you can do to approximate destructors in a garbage-collected language.
- Compile time evaluation in the D programming language – I explained the early **constexpr** design to Walter Bright.
- C++'s model of object lifetime based on constructors and destructors was part of the inspiration for Rust. Amusingly, these days, C++ is often accused of having borrowed such ideas from Rust.
- C adopted the C++11 memory model, the function declaration and definition syntax, declarations as statements, **const**, //-comments, **inline**, and initializers in **for**-loops.

Many differences between C++ and other languages stem from C++'s use of destructors. That makes it hard for garbage-collected languages to borrow directly from C++.

3 THE C++ STANDARDS COMMITTEE

The international C++ standards committee, officially named *ISO/IEC JTC1/SC22/WG21*, is central to the development of C++. This has been the case since its founding in 1991 and before that the focus of C++ development was the ANSI C++ standards committee from 1989 [Stroustrup 1993]. C++ has no rich owner or other significant funding sources, so the community relies on corporate development and open-source projects. WG21 and the National Standards committees are the only venues where people from otherwise competing organizations can meet to jointly solve problems.

The members are all volunteers – there is no paid secretariat – even if many members do come as representatives of the organizations they work for. At every meeting, there are people who proudly claim to represent "self." That is, they are not sponsored and represent only themselves. It is not uncommon for someone turning up to represent a new organization after a job change. There are many examples of people making "attendance of the C++ standards committee" as a condition for

accepting a new job. People have joined the committee to learn about C++ and "member of the C++ committee" has been quoted as a qualification (not always truthfully).

Some participate only for a couple of meetings or infrequently. On the other hand, there are people who have been at most meetings over decades. In the beginning and currently, there are three meetings a year. During a few years after the 1998 standard, we met only twice a year. Currently, the face-to-face meetings are supplemented with several teleconferences and many, many emails every day.

Here, I describe

- The role of the standard (§3.1)
- The organization of the committee (§3.2)
- The impact of the committee structure on the design of C++ (§3.3)

3.1 The Standard

The purpose of the standards committee is to write a standard. One official rationale for standards is "*facilitating trade, particularly in reducing technical barriers and artificial obstacles to international trade*" and "*providing a framework for achieving economies, efficiencies and interoperability.*" A standard is a specification, not an implementation. Its purpose is to keep multiple implementations in agreement, and to decide exactly what "agreement" means in a world where diverse underlying hardware must be exploited effectively. Many programmers have a problem understanding that. They consider their current compiler the definition of the language or have trouble understanding why it is hard to get 100% agreement among many separate – and typically competing – organizations. In the 1990s, the committee considered formal specification but after consulting with world-class experts concluded that neither specification technology nor the committee members were up to a formal specification of C++. Obviously, the idea of a reference implementation was considered, but the complexity of the language and especially issues related to hardware use and optimization have defeated such ideas. It would be too complicated and too expensive. Alternatively, it would be simplified to the point where it couldn't help with the hardest problems where it would be most needed. Also, a complicated reference implementation would be as likely to hide surprises as N competing implementation teams documenting their decisions, running extensive conformance tests, and discussing where they differ. For C++, N is at least four when it comes to front-ends (Clang, EDG, GCC, and Microsoft) and at least a dozen for backends.

So, the standards committee is grappling with the problems of having many implementations. The alternative would be to take the risks of a monoculture. If a single organization was the source of C++ technology, everybody would get the same, for good and bad. An organization controlling the "one true implementation" would have a dominant voice in the community and problems there would affect all. In particular, funding problems, commercial concerns, political opinions, and technical single-mindedness could seriously damage the language and its community.

For good and bad, the C++ community chose the semi-organized chaos of a large committee plus multiple compiler, tools, and library suppliers over a unified ownership/dictatorship model.

3.2 Organization

For the work on C++17 and C++20, as many as 250 people turned up at each face-to-face WG21 meeting, out of a membership of about twice that. In addition, there are national standards committees and C++ standards interest groups supporting members in a dozen or more countries, including Canada, Finland, France, Germany, Russia, Spain, the UK, and the USA. The members represent more than a hundred organizations. To give an idea, here is a selection: Apple, Bloomberg, CERN, Codeplay, EDG (Edison Design Group), Facebook, Google, IBM, Intel, Microsoft, Morgan Stanley,

Nvidia, Qt, Qualcom, Red Hat, Ripple, Sandia National Labs, University of Applied Sciences HSR Rapperswil, and University of Carlos III, Madrid. There is a solid representation from compiler suppliers, hardware suppliers, finance, games, library providers, platform suppliers, national labs (physics), and more. The telecom presence that was prominent in early C++ has decreased, whereas university presence, which used to be minimal, seems to be on the increase.

Obviously, such a large group of organizations and individuals, representing widely varying interest and technical backgrounds, need an organizational structure to function. The meetings are organized around working groups (WGs) and study groups (SGs). In the summer of 2019, we had:

- *Core WG* (CWG) – writes the final standards text for the language – chair, Michael Miller (EDG).
- *Library WG* (LWG) – writes the final standards text for the standard library – chair, Marshall Clow (The C++ Alliance, formerly Qualcom).
- *Evolution WG* (EWG) – processes language proposals – chair, Ville Voutilainen (Qt, formerly Symbio).
- *Library Evolution WG* (LEWG) – processes standard-library proposals – chair, Titus Winters (Google).

Study groups explore new areas and designs for possible standardization:

- *SG1, Concurrency* – concurrency and parallelism topics – chair, Olivier Giroux (Nvidia).
- *SG5, Transactional Memory* – Exploring transactional memory constructs – chair, Michael Wong (Codeplay, formerly IBM).
- *SG6, Numerics* – including but not limited to fixed point, decimal floating point, and fractions – chair, Lawrence Crowl ("self," formerly Google and Sun).
- *SG7, Compile-time programming* – Initially focused on compile-time reflection, then expanded to compile-time programming in general – chair, Chandler Carruth (Google).
- *SG12, Undefined behavior and Vulnerabilities* – a systematic review of vulnerabilities and undefined/unspecified behavior – chair, Gabriel Dos Reis (Microsoft, formerly Texas A&M University).
- *SG13, Human/Machine Interface and I/O* – selected low-level output (e.g., graphics, audio) and input (e.g., keyboard, pointing) I/O primitives – chair, Roger Orr (British Standards (BSI)).
- *SG14, Game Development and Low Latency* – topics of interest to game developers and others with low-latency requirements – chair, Michael Wong (Codeplay, formerly IBM).
- *SG15, Tooling* – topics related to creation of developer tools for standard C++, including but not limited to modules and package management – chair, Titus Winters (Google).
- *SG16, Unicode* – topics related to Unicode text processing in C++ – chair, Tom Honermann (Synopsis).
- *SG19, Machine Learning* – chair, Michael Wong (CodePlay, formerly IBM)
- *SG20, Education* – looking for ways to support learners and teachers approach C++ as it is today – chair, Jan Christiaan van Winkel (Google)
- *SG21, Contracts* – trying to design a contact system after the failure to do so for C++20 (§9.6.1) – chair John Spicer (EDG).

In 2017, a small group was established to address problems to do with the lack of direction in the design of the language and standard library [Dawes et al. 2018]. The members of this *Direction Group* (DG) are appointed by the convener in consultation with the WG chairs. Its members are long-term contributors to the committee, language, and standard library. The initial members were Beman Dawes, Howard Hinnant, Bjarne Stroustrup, David Vandevoorde, and Michael Wong; later, Beeman retired and Roger Orr joined). The DG chairmanship is rotating, starting with me. The DG

is advisory and has the policy of only giving opinions when its members agree unanimously. It maintains a document presenting its recommendations [Dawes et al. 2018; Hinnant et al. 2019]

The WGs persist over decades with slowly changing membership. The SGs come and go as interest dictates and/or they complete their work and hand over proposals to WGs for final processing. For example, four of the most significant SGs have declared victory and disbanded:

- *SG2, Modules* – chair, Gabriel Dos Reis (Microsoft, formerly Texas A&M University).
- *SG3, File system* – chair, Beman Dawes ("self").
- *SG8, Concepts* – chair, Andrew Sutton (University of Akron, Ohio, formerly Texas A&M University).
- *SG9, Ranges* – updating the STL to use concepts, simplify notation, and provide infinite sequences and pipelining – chair, Eric Niebler (FaceBook).

SG4, Networking is dormant as its results are waiting to be merged into the standard (§8.8.1). Another SG, *SG11, databases*, disbanded for lack of consensus and lack of critical mass of volunteers to get the work done.

Some SGs produce Technical Specifications (TSs) that can be significant documents in the style of the standard itself. They have some official (ISO) standing but don't offer the long-term stability of an international standard (IS). The Concurrency SG (SG1) has been active since 2006, headed by Hans-J Boehm (Google, formerly HP Labs, formerly SGI) for most of its existence, and has a standing very similar to the WGs.

In addition to these groups, there is a semi-official C/C++ liaison group consisting of people who are members of both the C++ committee and the C committee (ISO/SC22/WG14). This group tries to minimize C/C++ incompatibilities and the C++ standard documents every incompatibility. C and C++ are far more compatible than they would have been without the constant effort of the liaison group, but even then, most of the many features imported into C from C++ were modified so that they introduced some incompatibility.

There are just three official officers required by and recognized by the ISO:

- *Convener* – chairs the WG, sets the WG meeting schedule ("convenes" meetings), appoints Study Groups, and is responsible to higher levels of ISO (SC22, JTC1, and ITTF) for the WG's work – Herb Sutter (Microsoft) who has held that position since 2002 with a break from 2008-2009 when PJ. Plauger (Dinkumware) held it.
- *Project Editor* – ultimately responsible for applying committee-approved changes to the standard's working draft – Richard Smith (Google); for C++11, Pete Becker (Dinkumware); for C++14, Stephanus Du Toit (Intel).
- *Secretary* – responsible for taking and distributing minutes of WG21 meetings – Nina Ranns (Edison Design Group, formerly Symantec).

The National standards committees have their own officers and procedures.

Obviously, the positions are held by various people over the years, but it is rare for someone to hold a position for less than 5 years, despite the typically heavy workload. I was chair of the EWG for 24 years before handing over to Ville Voutilainen in 2014.

In general, smaller proposals go directly to EWG and/or LEWG and larger proposals start out in a SG. Proposals need to come in writing and be presented by someone. Typically, processing of a significant proposal takes several meetings (often years) and requires several papers, revisions of papers, and repeated presentations. Finally, a proposal that has gained strong support will be presented to the committee as a whole for a final vote. The convener looks at the vote and makes the call whether there is consensus. Consensus is not just a majority. The committee prefers unanimity after the WG processing and votes, and if that's not the case, a 9:1 or 8:2 majority is usually required. The convener may very well deem a 8:2 majority "not consensus." This happens if heads of national

standards bodies or several prominent members voice strong objections, so that there is a danger that issues will linger or lead to patchy adoption.

A standards meeting is exhausting. Usually members talk shop from breakfast until midnight, with formal sessions 8:30-12:30 and 14:00-17:30 plus evening sessions (19:00-22:00) most days. Members who are preparing proposals work even longer hours. WG and SG chairs typically have meetings over most mealtimes. Monday to Friday are full days, but if nothing surprising happens, most members are done by about 15:00 on Saturday. Still, when meetings are in nice places, such as Kona Hawaii, nobody outside the committee seems to be willing to believe that a meeting isn't some sort of vacation.

Voting in WGs and SGs is by one vote to everyone present. Formal voting in full committee is one vote to every organization present (so that large organizations don't get multiple votes) plus a count of the national body positions. The "technical vote" and the national body vote must agree for a consensus to be established.

The history of the committee before 2006 is documented in [Stroustrup 1993, 1994, 2007]. The C++ Foundation (§10.2) keeps a reasonably up-to-date description of the organization, key individuals, and processes of the committee on its website (isocpp.org/std).

An almost complete collection of committee papers from 1989 onwards is available [WG21 1989–2020]. Currently that collection increases by more than 500 papers a year. In addition, many of the committee's discussions are on archived mailing lists (called "reflectors"). There can be more than a hundred messages a day. It is very hard to keep up with all that goes on in the committee, especially as much require specialized technical knowledge to follow. I keep the collection of my WG21 papers on my home pages [Stroustrup 1990–2020].

Traditionally, ISO standards were revised every ten years or so. For example, we have C89, C99, and C11. The problem with such a long revision cycle is that if a new feature misses a feature freeze, we must wait another 12 years or so for it to become standard. Naturally, people then argue for slipping the next standard by a year or two: "This feature is so important that it can't wait, so we must delay the standard!" That was how C++0x became C++11, 13 years after C++98.

After C++11, several members wanted a shorter cycle and Herb Sutter, the convener, suggested we adopt a "train model." That is, "the train leaves at its scheduled time and anyone not on board will have to wait for the next scheduled departure." People liked that and there was a long discussion about what would be the right interval between standards revisions. I argued for a short interval, 3 years, because anything longer (e.g., 5 years) would be vulnerable to the "this feature is too important to wait" argument for delay. We agreed on a three-year "release cycle" and Herb Sutter added that we should adopt the "Intel tick-tock" model of alternating major and minor releases. That too was agreed, so three years after C++11 (§4) we delivered C++14 (§5) adopting delayed features and remedying minor problems discovered in early use. C++17 was also delivered on time, but sadly not as a major upgrade (§8). C++20 was voted feature complete in February 2019 and the final technical vote was done in Prague in February 2020.

3.3 Impact on Design

How do this organization of work, the elaborate decision processes, and the large number of participants affect the development of C++? Looking at the size of the committee, its composition, and its processes, I consider it amazing that anything constructive ever emerges. This is not just "design by committee;" it is "design by a confederation of committees."

In addition, the committee has only the weakest of management structures, lacking even the most basic management tools:

- There are no qualifications (say, of education or of practical experience) required for membership, speaking, or voting. Pay the ISO membership fee ($1280 for US members in 2018) and attend two meetings and you are a full voting member. In SGs and WGs anyone can speak and vote, even at their first meeting.
- There are no rewards beyond getting a proposal accepted and the satisfaction of seeing an improved standard. That is a major motivator, though.
- There are no real ways to discourage disruptive behavior. All the unofficial committee management can do is to politely encourage people not to do what others consider disruptive. Members have different opinions about what is disruptive.

Before considering the problems of evolving a language in a large committee, please remember that most of the time and work in the committee is done to resolve "minor problems;" that is, problems that don't rise to the level of language design philosophy, academic publication, or conference presentation. They are essential for keeping the language and its standard library from fragmenting into dialects and for portability across compilers and platforms. These issues include naming, name lookup, overload resolution, grammar details, exact meaning of constructs, lifetime of temporaries, linkage, and much, much more. Many are tricky to resolve and poor resolutions can have surprising and damaging consequences. Solutions tend to be crafted to minimize breakage of existing code. The committee resolves hundreds of issues a year. I estimate that the fraction of the committee members' time and effort spent at this is at least a third and maybe as high as two thirds. This work tends to be overlooked and underappreciated. If you have used a computer or a computerized gadget (e.g., a phone or a car), you can thank the people in the CWG and LWG for it working.

When focusing on problems caused by a huge committee, please also remember that these problems are caused by an embarrassment of riches: The C++ standards process is driven by hundreds of enthusiastic people from a wide variety of backgrounds, with a wealth of diverse experience, and a massive dose of idealism.

The committee is a filter supposedly preventing bad proposals from getting into the standard while improving the quality of the proposals that make it through. The existence of the committee encourages people to make proposals and come forward to help. However, there is no formal request-for-proposal process.

There are no full-time C++ designers, though there are many full-time C++ compiler, library, and tool implementers. Until recently, there were relatively few application builders on the committee. This is a problem because it biases the committee towards language lawyering, advanced features, and implementation issues, rather than directly addressing the needs of the mass of C++ developers, which many committee members know only indirectly. This problem may be partly alleviated by the recent sharp increase of new members.

There are relatively few educators on the committee. This can be a problem because the committee (rightly) prioritizes "ease of learning" highly but members have very different ideas (and often strong opinions) of what that means. This often confounds discussions about "simplicity" and "ease of use."

When considering the impact of organizational problems on the development of C++, please remember that the ISO process was not designed for 200-people meetings – the typical ISO programming language committee is one or two dozen people. On average we manage, partly by recognizing the problems and addressing them. Consider some observed problems:

- *Delays*: The multi-stage process offers many opportunities for delays, blocking of proposals, and for proposals to mutate. Dozens of members will insist that their needs be met, often by elaboration, extension, and special cases. One person's excessive delay is another's due diligence.
  Examples: Concepts (6 years for the current approach (§6)), contracts (6 years from start to failure (§9.6.1)), networking (15 years and still in progress (§8.8.1)) and constexpr (5 years (§4.2.7)). Even getting **nullptr** accepted took three years (§4.2.6).
- *Isolated features*: Most committee members like to see improvement; that is, to see features added. On the other hand, they are – quite reasonably – terrified of breaking existing code. This gives a systematic advantage to *isolated features*, small proposals that supposedly don't affect the rest of the language or the standard library. Such proposals rarely have major impact on how the language is used but add to the complexity of learning and implementation. Also, they often turn out to have surprising feature interactions after all.
  Examples: mostly features that were not worth mentioning in this summary of language evolution. Structured bindings (§8.2) and operator <=> (§8.8.4) both required many meetings to complete.
- *Late alternatives*: When – sometimes after years of work – a proposal comes near a vote, members who have so far not taken an interest enter the discussions and alternative proposals are made. Such proposals can be dramatically different from the original proposal or just a stream of requests for minor changes. This usually leads to delays, confusion, and sometimes acrimony as issues considered settled are resurrected and close-to-equal weight is given to untried (and usually unimplemented) new ideas and to proposals that are the result of years of work. For an older proposal, imperfections will have been uncovered and technical tradeoffs worked out. It is far too easy to imagine the benefits of something new and forgetting the law of unintended consequences: *there will be unintended consequences*. The new and relatively unexamined always looks better than the old. This makes the proponents of the older proposals defensive and diverts from efforts to refine the "old proposal." Here "old" could mean just a couple of years, or – as in the case of concepts (§6) a dozen years. Sometimes untried late changes ("improvements") are accepted to appease opposition; this often has unintended consequences. People entering into a discussion late often don't see "the need to rush" and naturally want to see their ideas seriously considered (often without seriously considering the details and rationale of an older proposal). This can cause friction with people who have already invested years of work on the older proposal.
  Examples: Structured bindings (syntax change, added support for bitfields, clumsy **get()** (§8.2)), concepts (§6). Digit separators (§5.1), operator dot (§8.8.2), modules (§9.3.1), coroutines (§9.3.2), contracts (§9.6.1).
- *Enthusiasm favors the new*: It is easier to summon enthusiasm for something new than for opposing it. Every proposal will solve something for someone and its proponents are willing to spend massive amounts of time demonstrating its value. To oppose, someone has to say things like
  - *No, this problem isn't all that important.*
  - *No, this solution has flaws.*
  - *No, you have not documented your solution sufficiently.*
  - *No, you have not examined the alternatives carefully.*

  However politely phrased, this can easily make an opponent appear to be a "the bad guy" blocking progress and denying the validity of the proponents' needs. Worse, the proponents invariably spend much more time preparing papers and presentations than opponents. Most

people prefer to work constructively on something they believe in, rather than carefully demolishing other peoples' work. So, the proponents are usually enthusiastic and well-prepared, whereas the opponents can easily end up sounding vague and demonstrating ignorance of details. Yet every new feature has a cost: e.g., design, specification, implementation, revision, deployment, and teaching (§9.5). I fear Thursday afternoons in the evolution working group. That's when EWG members are tired after working hard on major proposals for days, many of the regulars (such as me) have been dragged into other groups, and members are impatient to see something done. This is where minor proposals slip by with relatively minor scrutiny. Examples: explicit tests in conditions (§8.7), **inline** variables (§8), late changes to structured bindings (§8.2).

- *Overconfidence*: Relative to the complexity of the complete language, the complete standard library, and especially the complexity of the problems facing the users of C++ in diverse application areas, an individual's experience from everyday work is inadequate. Not all committee members see that or adequately compensate by doubting the wider relevance of their own experience. This can lead to proposals of limited generality being pushed too hard. Worse, it can lead to proposals being vigorously opposed by members who can't see the need to solve the problems addressed. Language design requires some intellectual humility [Stroustrup 2019b]. First solutions are rarely the best and off-the-cuff objections and suggestions rarely lead to improvements without further serious thought.
  Examples: No examples, to protect the guilty.
- *Poor implementation timing*: Implementing a proposal late in the standards process risks a seriously flawed feature, potentially with unimplementable parts and lack of feedback from use. Implementing a proposal early risks that the feature gets frozen in an incomplete, suboptimal, and hard-to-use form. This is a hard, practical dilemma: many in the committee will not vote for a proposal that has not been implemented or is implemented in a way they do not trust. On the other hand, many implementers are unwilling to devote implementation resources on a proposal that has not been approved by the committee. The committee often hear the question "has it been implemented?" Frequently, "Has it been designed?" and "How will this be used?" are more important questions. People easily get lost in details. My suggested way out of this dilemma is to agree on a direction, a general scope of a proposal, then start with the detailed design and implementation of a relatively small subset guided by critical use cases. That way, we can gain user experience relatively early and see how the feature interacts with other features. This requires a long-term view of what that language should be [Stroustrup 1993, 1994, 2007] (§1), (§11.2) or it deteriorates to mere opportunistic hacking. When it works, the language benefits from feedback and organic growth.
  Examples: modules (§9.3.1), C++0x concepts (§6), and <=> (§8.8.4).
- *Feature interaction*: One of the hardest issues to deal with is the use of features in combination. This is partly a technical issue of specification and implementation. As such, it soaks up much committee time. From a design perspective, the harder problem is to anticipate the use of a new feature in the context of the whole language, including other new language and library features under consideration. Every feature should be designed to be used in combination with other features. I fear that point is underappreciated. Few proposal papers offer detailed discussions and committee discussions about feature interactions tend to be short or confused. One consequence of this is that individual features tend to bloat to be usable in isolation from the rest of the language.
  Examples: **tuple** (§4.3.4) and <=> (§9.3.4). The (failed) proposals for having specialized syntax for actions in lambdas (§4.3.1).

- *Volume and distractions*: So much is going on, often concurrently, that nobody can keep up with all. Those of us who try, can easily get distracted from important issues by what turns out to be unimportant. There are now more than 500 committee papers a year, some are dozens or even hundreds of pages long. This represents about a doubling of the volume of documents compared to the early 2010s. I noted that the fall-2018 pre-meeting mailing (collection of new papers) had three times as many words as the complete works of Shakespeare.
  The flood of email messages can be most distracting as many members like to conduct technical discussions through bursts of short messages. If you fall behind in such a discussion, you lose track of issues and an apparent consensus can emerge from a handful of people.
  That kind of discussion does not lend itself to a calm and systematic weighing of alternatives. Sometimes, it leads to unfortunate features slipping through. Sometimes, it leads to different parts of the language and standard library reflecting different design philosophies and compromising interoperability.
  Examples: The differing interfaces to **any**, **optional**, and **variant** (§8.3). Concepts (§6).
- *Precise specification*: The standard is a specification, not an implementation. However, the standard is written in English, so mathematical precision escapes us. Many members of the committee are mathematically inclined, but more are not, so mathematical notation cannot be used in the specification. The attempts to make the English text precise and comprehensive make it stilted and damage comprehensibility. I often have a hard time understanding the standard's description of my own proposals.
  Most members are programmers, more than designers, so the specification sometimes ends up looking like a program – a program written in a low-level language without a type system or a compiler. There are elaborate if-then-else explanations and few statements of invariants. Worse, much of the vocabulary is inherited from C and based on tokens in the program source text so that higher-level concepts are stated only indirectly.
  Curiously, the standard-library specification is noticeably more formal in structure than the language specification.
- *Scholasticism*: A heavy emphasis on having the text of the standard correct and precise is of course necessary. However, people sometimes forget that the standard might be wrong and discuss correctness exclusively based on arguments from the text. Thus, arguments based on the models and uses that the standard's text supposedly reflects can be ignored.
- *Direction*: Which problems are real? Important? For whom? Which are urgent? Which solutions will still be relevant in a decade's time? That something can be a problem doesn't imply that it must have a direct solution in the language. In particular, it is hard for a committee to remember that a language cannot be all things to all people. It is even harder to accept that it cannot solve even the most urgent problems of every member [Stroustrup 2018d].
  Examples: C++17 (§8) and C++20 (§9).
- *Exclusive focus*: Some members focus exclusively on one or two issues, such as language technicalities, ease of use, "teachability," efficiency, use in a single style of programming, use in a single industry, use in a single firm, a single language feature, etc. This can be a very effective technique for a member with an exclusive focus, but can make broad, balanced progress difficult. Excessive trust in theory or in personal experience are other examples of this. A good proposal offers progress in many areas but is typically not perfect along all of these axes.
- *Inappropriate application of principle*: Applying general principles to a concrete example is often difficult. Sometimes, a principle is applied rigidly without the necessary tradeoff with other principles. The need for tradeoff is one reason D&E [Stroustrup 1994] refers to design

principles as "rules of thumb." Sometimes, a principle seems to appear out of nowhere without empirical basis. Sometimes, a principle is rigidly adhered to for one proposal, yet disregarded for another. Principled design is difficult; it requires taste and experience as well as principles. Practical language design is not just an exercise in deduction from first principles. Often, principles have to be balanced against each other.

- *Pro-expert bias*: It is hard to imagine the problems of someone different from yourself. The committee members are almost all experts in something or other. In their day jobs, they are typically the persons who deal with the most subtle and intricate problems. Such problems are often rare in the billions of lines of C++ code "out there" and not the problems that the majority of C++ programmers struggle with. Yet, the expert-level problems are typically the ones that are urgent to the committee and have the easiest passage through the process.
  Examples: The ease with which support for the use of **enable_if** and type traits (§4.5.1) permeated the standard library compared to the trouble getting concepts (§6) accepted.
- *Cleverness*: The average committee member is clever and many show a weakness for clever solutions. Further, they have a hard time to decide that not every problem is worth solving and that having a solution doesn't imply that we have to put it into the standard. This leads to overelaborate features and to features that most programmers could happily live without. It is only fair to point out that many programmers are also clever and sometimes delight in overly clever language and standard-library features.
  Example: proposals that require serious template metaprogramming even for simple uses.
- *Unwillingness to compromise*: Most members have strong opinions but gaining consensus in a large group requires compromise. It can be hard to distinguish between compromise over something inessential and over fundamental principles. The latter could damage the language and should be avoided. Unfortunately, members who are firmly convinced that their concerns are fundamental and essential have a critical tactical advantage compared to people with a more open mind. People who care more about the language as a whole than about any individual issue tend to give in to people who don't. Similarly, people who never seriously question their own principles or needs can mount a ferocious attack against technical compromises that others consider necessary. Making progress requires concern for the community as a whole, self-awareness, and a dose of humility [Stroustrup 2019b].
- *Lack of priority*: From a technical point of view, all problems are equal: an imprecise specification is an imprecise specification independently of what it fails to properly specify and any error that might slip through a hole in the type system could in principle cause death and destruction. However, the real-world implications can be dramatically different. In fact, most obscure details have essentially no effect. That's hard for some people to remember when working on details of a design.
  Example: More time was spent on digit separators (§5.1) than on range-**for** (§4.2.2).
- *Perfectionism*: A standard is expected to be used by millions and be stable for decades. Naturally, people would like it to be perfect. This leads to feature bloat (too many features) and in particular to the bloat of individual features. Programmers are excellent at imagining problems and as a feature progresses through the committee, members insist that it solve them all. This can lead to serious mission creep and to features only an expert could love. It can also lead to a feature not making it into the standard.
  Examples: Operator dot (§8.8.2), networking library (§8.8.1), and exception specifications (§4.5.3).
- *Minority blocking*: The consensus process protects against quite a few kinds of mistakes and especially against the tyranny of the majority. However, it is vulnerable to individuals and small groups blocking progress. That can be good (avoiding mistakes), but when it happens

repeatedly at the various stages of the proposal pipeline or just in the last minute, it can be disruptive.
Examples: **constexpr** (§4.2.7), Operator dot (§8.8.2), modules (§9.3.1), and coroutines (§9.3.2).

- *Cohesive groups*: Many WGs and SGs have a stable core set of people who over the years develop a cohesive outlook, a shared vocabulary, and specific ways of operating. This can make it difficult for "outsiders" to communicate and contribute. It can also make it difficult to design features that cross WG boundaries, such as facilities with both library and language parts. Each group is likely to design something that fits into its own organizational domain, thus adding confirmation to the old dictum that the structure of a system resembles the structure of the organization that created it.
  Examples: range-**for** (§4.2.2) and concurrency mechanisms that might require language changes (§4.1.3). The differing interfaces to **any**, **optional**, and **variant** (§8.3).

On the positive side, actions based on personal animosity or tit-for-tat are rare. In that sense, the committee is quite professional.

Fortunately, not every proposal suffers from the effects of all of these phenomena and most of these problems are shared with many other large projects. However, C++, as represented by its ISO standard, reflects these phenomena. They are not brand new, but have been increasing since C++11. I suspect they are caused by a combination of

- the increased size of the committee
- the influx of new people
- the specialization (fragmentation) of the membership
- the decrease of knowledge of C++'s history among the members

One reason the standards process has repeatedly succeeded despite these serious problems is that many people mount a constant effort to minimize the negative effects. The creation of the Direction Group is part of the effort (§3.2) [Dawes et al. 2018; Stroustrup 2018d]. See also (§11.4). The constant efforts of working group chairs, note takers, meeting organizers, and editing groups are invisible, but essential. For example, Jens Maurer has for decades taken notes in the CWG, helped proposers write standards text, arranged for web access, organized phone access to members unable to attend, organized meeting rooms, informed members about local travel possibilities, and more.

What would be an alternative? In an ideal world, I would recommend restricting decision making to a small group (on the order of 5 people) of full-time trusted experts but have the discussions, the ability to propose, and most of the processing done by a large group (like the 350+ member committee). No, I don't see something like that happening for C++:

- Nobody likes to give up power (in this case voting power).
- Maintaining stable funding for a permanent staff of full-time experts requires non-trivial skills (and such skills haven't surfaced in the C++ community).
- Radical change doesn't happen during times of success; only a marked drop in C++ use could motivate the committee to dramatic organizational innovation (and then it would probably be too late).

I don't see corporate control as a viable alternative:

- Corporations expect return on investment.
- Corporate support often evaporates after a few years.
- Corporations tend to prefer differential advantages over progress that benefits all.

Nor do I consider fully open processing (by thousands of voters) viable:

- A cast of thousands does not have taste.
- The membership and opinions of a large group are not stable over decades.

The hierarchical approval process that works for many large open-source projects might have been at least part of an answer, but there was little experience with that when the standardization of C and C++ started. When such a system works well, the higher you get in the approval hierarchy the broader a base of knowledge of the approvers and the wider their area of concern. At the top, we find one or more people with some knowledge of just about everything and concern for all users. To contrast, the ISO process dilutes expertise and areas of concern as a proposal approaches final approval: In plenary, many members vote on proposals that they have little interest in, have limited experience with the problem area, and haven't followed closely. People try hard to do so responsibly, but that is hard and seeing each proposal as part of the bigger picture is almost impossible.

Looking at it this way, WG21 hasn't done that badly. I do worry whether this model can keep C++ coherent and relevant for much longer. The 200+ people who turn up for a C++ standards meeting is an order of magnitude larger than for other standards groups, for which the ISO process was designed. Also, the membership is far more diverse than the groups of grizzled experts, corporate representatives, and national body representatives of the past. Chaos could erupt.

I take some comfort from Winston Churchill's dictum that "democracy is the worst form of Government except for all those other forms that have been tried from time to time."

In particular, I don't see the often suggested "Benevolent Dictator For Life" model scaling, and that model has never been relevant for C++ anyway.

An individual or a small coherent group of friends is my ideal model for starting a language design project, but I don't see that approach scaling. A mature language needs dozens or even hundreds of people working on the huge variety of problems that must be faced. Even coordination with relevant standards groups and industry groups would swamp the capacity of a tiny coherent group.

3.4 Proposal Checklists

There was a "How to write a proposal" guide for C++98 [Stroustrup et al. 1992], but curiously the evolution group did not have a checklist for proposals for C++14, C++17, or C++20. There was one for standard-library proposals [Meredith 2012]. For C++20, a note from the heads of National Standards Bodies [van Winkel et al. 2017] and a paper from the Direction Group [Hinnant et al. 2019] gave some guidance. Here is a short and incomplete list of questions that were almost always raised for a proposal:

- What is the problem to be solved? What kind of users will be served? Novices? Experts?
- What is the solution? Articulate the principles that it is based on. Give simple use cases and examples of expert-level use.
- What are alternative solutions? Could a library solution be sufficient? Why are current facilities not good enough?
- Why does the solution need to be in the standard?
- What barriers to adoption are there? How long is a transition from existing techniques likely to take?
- Has it been implemented? What implementation problems were encountered or can be expected? Is there any user experience?
- Will there be significant compile-time overheads?
- Does the feature fit into the frameworks of existing tools and compilers?
- Will there be run-time overheads compared to workarounds? In time? In space?
- Will there be compatibility problems? Breakage of existing code? ABI breakage?
- How will the new feature interact with existing and other new features?

- Is the solution teachable? To whom? By whom?
- How will the standard library be affected?
- Will the proposal lead to demands for further extension in future standards?
- How does the feature fit into the wording of the standard?
- What mistakes are users likely to make with the new feature?
- Is the proposal among the top-20 in terms of benefits to the C++ community at large? Top-10?
- Is the proposal among the top-3 in terms of a specific sub-community? Which sub-community?
- Is the proposal for a general mechanism to solve a class of problems or a specific solution to a specific problem? If to a class, which class of problems?
- Is the proposal coherent with the rest of the language in terms of semantics, syntax, and naming?

Ideally, a proposal would have answers to all these questions, and more, but that rarely happened. In particular, the rationale was often very weak in an initial proposal as proposers thought that the importance of the problems addressed and their suggested solution were pretty obvious. However, follow-up papers, revisions, email discussions, and face-to-face discussions in the Evolution Group typically cover those questions, but rarely systematically or consistently across proposals. Members have a tendency to focus on technical details (e.g., grammar, ambiguities, optimization opportunities, and naming), rather than revisiting fundamental questions. Sometimes, what I consider a bad proposal slips through. The reason is usually great enthusiasm from proposers combined with distraction, politeness, and exhaustion among opponents [Stroustrup 2019b].

4 C++11: IT FEELS LIKE A NEW LANGUAGE

The release of C++11 [Becker 2011] and the relatively quick deployment of implementations led to much enthusiasm, increased use, an influx of new people into the C++ world, and much experimentation. Three complete or almost complete implementations of C++11 were available in 2013. My comment at the time, *C++11 feels like a new language* [Stroustrup 2014d], was widely perceived as accurate. Why and how did C++11 do such a good job helping programmers?

C++11 introduced a bewildering number of language features, including:

- memory model – an efficient low level-model of modern hardware as a foundation for concurrency (§4.1.1)
- **auto** and **decltype** – avoiding redundant repetition of type names (§4.2.1)
- range-**for** – simple linear traversal of ranges (§4.2.2)
- move semantics and rvalue references – minimizing copying of data (§4.2.3)
- uniform initialization – an (almost) completely general syntax and semantics for initializing objects of all kinds and types (§4.2.5)
- **nullptr** – a name for the null pointer (§4.2.6)
- **constexpr** functions – compile-time evaluated functions (§4.2.7)
- user-defined literals – literals for user-defined types (§4.2.8)
- raw string literals – literals where escape characters are not needed, mostly for regular expressions (§4.2.9)
- attributes – associating essentially arbitrary information with a name (§4.2.10)
- lambdas – unnamed function objects (§4.3.1)
- variadic **template**s – **template**s that can handle an arbitrary number of arguments of arbitrary types (§4.3.2)
- template aliases – the ability to rename a template and to bind some template arguments for the new name (§4.3.3)
- **noexcept** – a way of ensuring that an exception isn't thrown from a function (§4.5.3)

- **override** and **final** – explicit syntax for managing large class hierarchies
- **static_assert** – compile-time assertions
- **long long** – a longer integer type
- default member initializers – give a data member a default value that can be superseded by initialization in a constructor
- **enum class**es – strongly typed enumerations with scoped enumerators

And here is a list of the major standard-library components (§4.6):

- **unique_ptr** and **shared_ptr** – resource-management pointers (§4.2.4) relying on RAII (§2.2.1)
- memory model and **atomic** variables (§4.1.1)
- **thread**, **mutex**, **condition_variable**, etc. – type-safe and portable support for basic system-level concurrency (§4.1.2)
- **future**, **promise**, and **packaged_task**, etc. – slightly higher-level concurrency (§4.1.3)
- **tuple** – unnamed simple composite types (§4.3.4)
- type traits – testable properties of types for use in metaprogramming (§4.5.1)
- regular expression matching (§4.6)
- random numbers – with many generators (engines) and distributions (§4.6)
- Time – **time_point** and **duration** (§4.6)
- **unordered_map**, etc. – hash tables
- **forward_list** – a singly-linked list
- **array** – a fixed-constant-sized array that knows its size
- emplace operations – construct objects right within a container to avoid copying
- **exception_ptr** – enables transfer of exceptions between threads

There are more, but these are the most significant changes. All are described in [Stroustrup 2013] and much information is available online (e.g., [Cppreference 2011–2020]).

How could these apparently disconnected extensions make a coherent whole? How could this actually change the way we write code for the better? C++11 did achieve that. In a relatively short period of time (say, 5 years), huge amounts of C++ were upgraded to C++11 (and further to C++14 and C++17) and the presentation of C++ at conferences and blogs completely changed.

This dramatic change in the "feel" of the language and the styles of its use is not the result of a traditional careful design process guided by a master craftsman, but the result of a huge set of suggestions filtered through layers of decisions by a large and changing set of individuals.

In my HOPL3 paper [Stroustrup 2007], I correctly described many of the C++11 language features. The notable exception was "concepts" which is addressed in (§6). Instead of going into details again, I will describe a classification of facilities into "themes" based on what programmer needs they address. I think this way of looking at proposals is the root of C++11's success:

- §4.1: Support concurrency
- §4.2: Simplify use
- §4.3: Improve support for generic programming
- §4.4: Increase static type safety
- §4.5: Support for library building
- §4.6: Standard library components

These "themes" are not disjoint. In fact, my conjecture is that C++11 was a success because it adds to up a fine mesh of interrelated facilities addressing genuine needs. My favorite features belong in every theme. I suspect that my articulated aims for C++ in writing (e.g., [Stroustrup 1993, 1994, 2007]) and presentations helped the design remain reasonably focused. For me, a crucial

measure of every new feature is whether it brings C++ closer to its ideals, e.g., by making the support for built-in types and user-defined types more similar (§2.1).

Looking at C++11, we see suggested improvements from about 2002 and quite a few libraries emerging early, often as part of Boost [Boost 1998–2020]. However, complete C++11 implementations were not available until 2013. In 2020, some organizations still struggle with an upgrade to C++11 because of huge code bases, programmers stuck in the past, outdated teaching, and (especially in the embedded systems world) seriously outdated compilers. Adoption of C++17 is noticeably faster than the adoption of C++98 and C++11, though, and as early as 2018 some major C++20 features were already in production use.

As late as 2018, I have seen pre-C++98 compilers used in teaching. I consider that abuse of students, depriving them of 20+ years of progress.

What is the distant past according to the standards committee, to the major compiler vendors, and to most vocal C++ proponents is still the present – or even the future – for many. The result is a continuing confusion about what C++ really is. This confusion will last as long as C++ continues to evolve.

4.1 C++11: Support for Concurrency

C++11 had to support concurrency. It was both obvious and a common requirement from all major users and platform suppliers. C++ was (and is) heavily used as the foundation for most of the software industry, and in the first decade of 2000 concurrency was becoming pervasive. Exploiting hardware concurrency well was essential. Like C, C++ had of course always supported various forms of concurrency, but that support had not been standardized and was generally low level. Machine architectures were using increasingly subtle memory architectures and compiler writers were applying increasingly aggressive optimization techniques, making life extremely difficult for the writers of the lower levels of software. A treaty between machine architects and optimizer writers was badly needed. Only with a well-specified memory model could writers of foundational libraries have a stable base and some degree of portability.

The work on concurrency was spun off from the EWG into the concurrency group with expert membership led by Hans-J Boehm (HP, later Google). It had three main trusts:

- §4.1.1: Memory model
- §4.1.2: Threads and locks
- §4.1.3: Futures

In addition, parallel algorithms (§8.5), networking (§8.8.1), and coroutines (§9.3.2) were dealt with in separate groups and (as expected) not ready for C++11.

4.1.1 Memory Model. One of the most urgent issues was to precisely specify the rules for accessing memory in a world of multi-cores, caches, speculative execution, instruction reordering, etc. Paul Mckenney from IBM was very active on topics of memory guarantees. The research of Mark Batty from Cambridge University [Batty et al. 2013, 2012, 2010, 2011] helped our formalizations as described in [McKenney et al. 2010] by P. McKenney, M. Batty, C. Nelson, H. Boehm, A. Williams, S. Owens, S. Sarkar, P. Sewell, T. Weber, M. Wong, L. Crowl, and B. Kosnik. It was a massive and essential part of C++11.

In C11, C adopted the C++ memory model. However, in the very last minute before the C standard was put out to vote and after the last opportunity to change the C++11 standard, the C committee introduced notational incompatibilities that became a pain for C and C++ implementers and users.

Much of the memory model was motivated by the needs of the Linux and Windows kernels. It is now used there and also far more widely. The memory model is widely underrated because it

isn't seen by most programmers. To a first order of approximation, it simply makes code work as anyone would expect.

Initially, I think that most of the committee members underestimated the problem. We knew that Java had a good memory model [Pugh 2004] and hoped to adopt that. I was highly amused to find that representatives from Intel and IBM effectively vetoed that idea by pointing out that by adopting the Java memory model for C++ we would slow down all JVMs by a factor of at least two. Consequently, to preserve the performance of Java, we had to adopt a far more complex model for C++. Ironically and predictably, C++ was then criticized for having a more complicated memory model than Java.

Basically, the C++11 model is based on *happens-before relations* [Lamport 1978] and supports relaxed memory models as well as sequentially consistent [Lamport 1979] ones. On top of that and integrated with it, C++11 provides atomic types and support for lock-free programming. The details are far beyond the scope of this paper (e.g., see [Williams 2018]).

Unsurprisingly, the memory model discussions in the concurrency group got a bit heated at times. Significant interests of hardware manufacturers and compiler vendors were at stake. One of the hardest decisions was to accept both Intel's x86 primitives (a Total Store Order (TSO) model [TSO Wikipedia 2020] plus a few atomic operations) and IBM's PowerPC primitives (weak consistency plus fences) for the lowest level synchronization. Logically, only one set of primitives were needed, but Paul Mckenney convinced me that there was far too much code written using fences deep in complex algorithms for IBM to adopt something like Intel's model. One day, I literally did shuttle diplomacy between two corners of a large room. Eventually, I suggested that both had to be supported and that was what C++11 adopted. I – and others – were very pleased when later people found that fences and atomics could be used together to create better solutions than either in isolation.

A bit later, we added support for data-dependency-based consistency represented in source code through attributes (§4.2.10), such as **[[carries_dependency]]**.

C++11 introduced **atomic types** on which simple operations are atomic:

```
atomic<int> x;
void increment()
{
    x++;  // not x = x + 1
}
```

Obviously, these are widely useful. For example, using an atomic makes the notoriously tricky double-checked locking optimization trivial:

```
mutex mutex_x;
atomic<bool> init_x; // initially false.
int x;

 if (!init_x) {
    lock_guard<mutex> lck(mutex_x);
    if (!init_x) x = 42;
    init_x = true;
 }  // implicitly release mutex_x here (RAII)

// ... use x ...
```

The point of double-checked locking is to use the relatively cheap **atomic** to guard the use of the much more expensive **mutex**.

The **lock_guard** is a RAII type (§2.2.1) that guarantees the unlocking of the **mutex** it controls.

Hans-J Boehm has described the atomic types as "surprisingly popular," but I can't say I'm surprised. Being less expert than Hans, I appreciate the simplification more. C++11 also introduced the key operations for lock-free programming, such as compare and swap:

```
template<typename T>
class stack {
   std::atomic<node<T>*> head;
public:
   void push(const T& data)
   {
      node<T>* new_node = new node<T>(data);
      new_node->next = head.load(std::memory_order_relaxed);
      while(!head.compare_exchange_weak(new_node->next, new_node,
            std::memory_order_release, std::memory_order_relaxed)) ;
   }
   // ...
};
```

Even with C++11 support, I consider lock-free programming expert-level work.

4.1.2 Threads and Locks. On top of the memory model, a threads-and-locks model of concurrency was provided. I consider the threads-and-locks level of concurrency the worst model for application use of concurrency, but it is essential for a language like C++. Whatever else it is, C++ is (and always was) a systems programming language capable of interacting directly with the operating system, usable for kernel code and device drivers. Therefore, it has to support what the systems support at their lowest levels. On top of that, we can build a variety of concurrency models more suitable for specific applications. Personally, I'm particularly fond of message-based systems because they can eliminate the data races that are the root of the most subtle concurrency bugs.

C++'s support for the threads-and-locks level of programming is a type-safe variant of what POSIX and Windows offer. It is described in [Stroustrup 2013] and in greater depth in Anthony Williams' book [Williams 2012, 2018]:

- **thread** – a system's thread of execution, with **join()** and **detach()**
- **mutex** – a system's mutex with **lock()**, **unlock()**, and RAII ways of getting guaranteed **unlock()**
- **condition_variable** – a system's condition variable for communicating events between threads
- **thread_local** – thread-local storage

Compared to the C versions, the type safety makes the code much simpler and cleaner, e.g., no more **void****s and macros. Consider a simple example of having a function execute on a different thread and returning a result:

```
class F {  // traditional function object
public:
    F(const vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()();        // place result in *res
private:
    const vector<double>& v;  // source of input
    double* res;              // target for output
};
```

```
double f(const vector<double>& v);  // traditional function

void g(const vector<double>& v, double* res); // put result into *res

int comp(vector<double>& vec1, vector<double>& vec2, vector<double>& vec3)
{
    double res1;
    double res2;
    double res3;
    // ...
    thread t1 {F{vec1,res1}};             // function object
    thread t2 {[&](){res2=f(vec2);} };    // lambda
    thread t3 {g,vec3,&res3};             // ordinary function

    t1.join();
    t2.join();
    t3.join();

    cout << res1 << ' ' << res2 << ' ' << res3 << '\n';
}
```

The design of the type-safe library support relies critically on variadic templates (§4.3.2). For example, the constructor for **std::thread** is a variadic template. It can distinguish different executable first arguments and check that they are followed by a correct number of arguments of correct types.

Similarly, lambdas (§4.3.1) made many uses of the **<thread>** library much simpler. For example, the argument for **t2** is a piece of code (a lambda expression) that accesses the surrounding local scope.

It was difficult to get novel features accepted and used in the standard library in the same release of the standard. There were voices raised that doing this was too aggressive and that it could lead to long-term problems. Introducing new language features and using them at the same time was undoubtedly risky, but it added significantly to the quality of the standard by

- giving users a better standard library
- giving users examples of good use of the language features
- saving users from having to implement low-level facilities
- forcing designers of language features to cope with difficult real-world uses

The threads-and-locks model requires the use of some form of synchronization to avoid race conditions. C++11 provides standard **mutex**es for that:

```
mutex m;  // controlling mutex
int sh;   // shared data

void access()
{
    unique_lock<mutex> lck {m};    // acquire mutex
    sh += 7;                       // manipulate shared data
} // release mutex implicitly
```

The **unique_lock** is a RAII object ensuring that the user cannot forget to **unlock()** the **mutex**. These lock objects also provided a way of protecting against the most common form of deadlock:

```
void f()
{
    // ...
    unique_lock<mutex> lck1 {m1,defer_lock};  // don't yet acquire m1
    unique_lock<mutex> lck2 {m2,defer_lock};
    unique_lock<mutex> lck3 {m3,defer_lock};
    // ...
    lock(lck1,lck2,lck3);    // acquire all three mutexes
    // ... manipulate shared data ...
} // implicitly release all mutexes
```

Here, the **lock()** function acquires all **mutex**es "simultaneously" and releases all implicitly (RAII (§2.2.1). C++17 has an even more elegant solution (§8.4).

The threads library was first proposed for C++0x by Pete Becker (Dinkumware) [Becker 2004] in 2004 based on a Dinkumware implementation of the interface offered by **boost::thread** [Boost 1998–2020]. It is probably not a coincidence that this was presented at the same meeting (Redmond WA, September 2004) as the first proposal for a memory model [Alexandrescu et al. 2004].

The biggest controversy was over cancellation, which is the ability to stop a thread from running to completion. Essentially every C++ programmer on the committee wanted that in some form or other. However, the C committee objected to thread cancellation in a formal note to WG21 [WG14 2007], the only formal note ever sent from WG14 (the ISO C standards committee) to WG21. I noted, "but C doesn't have destructors and RAII for systematic resource management and clean-up." The Austin Group who manages POSIX sent representatives who were 100% against any form of the idea, insisting that cancellation was neither necessary nor possible to do safely. Observing that Windows and other operating systems offer variants of the idea and that C++ isn't C made no difference to the POSIX people. I fear that they were defending their business and their C-language world view, rather than trying to come up with the best solution for C++. The lack of standard cancellation has repeatedly been a problem. For example, in a parallel search (§8.5), the thread that first finds the answer would like to trigger cancellation (by any name) of the other such threads. C++20 provides the stop-token mechanism to support this use case (§9.4).

4.1.3 Futures. A type-safe and standard POSIX/Windows-like thread library was a major improvement over the incompatible C-style libraries in use, but that was still 1980s style low-level programming. Some members, notably me, argued that C++ badly needed something more modern and higher level. For example, Matt Austern (Google, formerly SGI) and I argued for message queues ("channels") and a thread pool. That made little progress against objections that there wasn't time to do that right. I pleaded and pointed out that if the experts in the committee didn't offer such facilities, they would end up having to use facilities "cooked up in a hurry by my students!" The committee could certainly do a much better job than that. "If you won't do that, please give me one way, just one way, to pass information between threads without explicit synchronization!"

The committee were split between members who basically wanted POSIX with improved typing (notably, PJ Plauger) and members pointing out that POSIX was basically a 1970s design and "everybody" was using higher-level facilities already. At the 2007 Kona meeting we agreed on a compromise: C++0x (still expected to become C++09) would offer **promises** and **futures** and a launcher for asynchronous tasks, **async()**, that allows but does not require a thread pool. Like most compromises, "the Kona compromise" pleased nobody and led to some technical problems. However, many users considered it a success – most didn't know it was a compromise – and over the years, improvements have emerged.

In the end, C++11 offered:

- **future** – a handle from which you can **get()** a value from a shared one-object buffer, possibly after a wait for the value to be put there by a **promise**.
- **promise** – a handle through which you can **put()** a value to a shared one-object buffer, possibly waking up a **thread** waiting on a **future**.
- **packaged_task** – a class that makes it easy to set up a function to be executed asynchronously on a **thread** with a **future** for its result and a **promise** to return the result.
- **async()** – a function that can launch a task to be executed on another **thread**.

The easiest way of using all this is to use **async()**. Given an ordinary function as an argument, **async()** runs it on a **thread** handling all the details of thread launching and communication:

```
double comp4(vector<double>& v)
  // spawn many tasks if v is large enough
{
    if (v.size()<10000)    // is it worth using concurrency?
        return accum(v.begin(),v.end(),0.0);
    auto v0 = &v[0];
    auto sz = v.size();

    auto f0 = async(accum,v0,v0+sz/4,0.0);          // first quarter
    auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);     // second quarter
    auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0);   // third quarter
    auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);     // fourth quarter

    return f0.get()+f1.get()+f2.get()+f3.get();     // collect the results
}
```

The **async** wraps the code in a **packaged_task** and manages the setup of a **future** and its **promise** to transmit the result.

Either a value or an exception can be passed from one **thread** to another through a **future**/**promise** pair. For example:

```
X f(Y); // ordinary function

void ff(Y y, promise<X>& p)    // to execute f(y) asynchronously
{
    try {
        X res = f(y);      // ... compute a value for res ...
        p.set_value(res);
    }
    catch (...) {          // oops: couldn't compute res
        p.set_exception(current_exception());
    }
}
```

For simplicity, I have not used perfect forwarding of the argument (§4.2.3).

A **get()** on the corresponding **future** will now either get a value or throw an exception – exactly as for an equivalent synchronous call of **f()**.

```
void user(Y arg)
{
    auto pro = promise<X>{};
```

```
    auto fut = pro.get_future();
    thread t {ff,arg,pro};  // run ff on a different thread
    // ... do something else for a while ...
    X x = fut.get();
    // ...
}
```

The standard library **packaged_task** automates this way of wrapping an ordinary function in a function object that handles the **promise**/**future** setup and deals with returns and exceptions.

I hoped that this would lead to a work-stealing implementation supported by a thread pool, but I was disappointed.

See also (§8.4).

4.2 C++11: Simplifying Use

C++ is "expert friendly." I think I was the first to use that phrase as a gentle criticism and to popularize the slogan *Make simple things simple!* in the context of C++. A language primarily aimed at industrial use should be friendly to experts, of course, but a language cannot be *just* expert friendly. Most people who use a programming language are not – and do not want to become – expert in all aspects of that language. They want to get their job done sufficiently well with the least distraction from the language. A programming language is there to allow application ideas to be expressed, not to turn programmers into language lawyers. It follows that a language design should strive to make simple things simple. A language that is also for experts must additionally ensure that nothing essential is impossible or unreasonably expensive.

Another criterion commonly applied in discussions about potential C++ language extensions and standard-library components is "Is it easy to teach?" This question is now very common; it was pioneered by Francis Glassborow and me. The idea of "being easy to teach" was there from the earliest days of C++; you find it in *The Design and Evolution of C++* [Stroustrup 1994].

Naturally, proponents of something new inevitably deem their design simple, easy to use, reasonably safe, efficient, easy to teach, and useful for most programmers. Opponents tend to doubt some or all of those claims. However, it is important that this discussion takes place for every new feature proposed for C++: in person at meetings, in papers [WG21 1989–2020], and in email discussions. In these discussions, I often point out that I'm a novice most of the time. That is, whenever I learn a new feature, technique, or application domain, I am a novice and can use all the help I can get from the language and standard library. One result was that C++11 offered some facilities specifically aimed to simplify the use of C++ by learners, novices, and non-language-experts.

Every new feature makes something simpler for somebody. The "Simplifying use" theme focuses on features for which making it simpler to express a known idiom was a major part of its motivation. Here is a selection:

- §4.2.1: **auto** – avoiding redundant repetition of type names
- §4.2.2: Range-**for** – simple linear traversal of ranges
- §4.2.3: Move semantics and rvalue references – minimizing copying of data
- §4.2.4: Resource-management pointers – "smart" pointers managing the lifetime of the objects they point to (**unique_ptr** and **shared_ptr**)
- §4.2.5: Uniform initialization – an (almost) completely general syntax and semantics for initializing objects of all kinds and types
- §4.2.6: **nullptr** – a name for the null pointer
- §4.2.7: **constexpr** functions – compile-time evaluated functions
- §4.2.8: User-defined literals – literals for user-defined types

- §4.2.9: Raw string literals – literals where the escape character ('\') is not used to escape, mostly for regular expressions
- §4.2.10: Attributes – associating essentially arbitrary information with a name
- §4.2.11: An interface to an optional garbage collector
- §4.3.1: Lambdas – unnamed function objects

After C++11 was beginning to see serious use, I started taking small unscientific surveys as I traveled widely to talk with C++ users: Which C++11 features do you most like? The top three were invariably:

- §4.2.1: **auto**
- §4.2.2: Range-**for**
- §4.3.1: Lambdas

These three are among the simplest additions to C++11 and do not offer any new fundamental functionality. What they do could be done in C++98, but not as elegantly.

I interpret this to imply that programmers of all abilities really appreciate concise notation for simple common uses. They will happily abandon a more general notation for a simpler and more specialized one for the cases where it applies. A common rallying cry is "There should be only one way of saying something!" This "design principle" simply doesn't reflect real-world user preferences. I tend to rely on *the onion principle* instead [Stroustrup 1994]. You design so that doing simple tasks is simple and when you need to do something less simple, you need to be more detailed, using a more complicated technique or notation. That is, you peel one layer off the onion. The more layers you peel off, the more you cry.

Please note that here *simple* does not mean *low level*. It is superficially simple to learn low-level facilities such as **void***, macros, C-style strings, and casts but it is hard to use them to produce quality, maintainable software.

4.2.1 ***auto*** *and* ***decltype***. The oldest new feature in C++11 was the ability to specify that an object should have the type of its initializer. For example:

```
auto i = 7;          // i is an int
auto d = 7.2;        // d is a double
auto p = v.begin();  // p gets the type of v's iterator
                     // (begin() returns an iterator)
```

auto is a static facility allowing the deduction of the (static) type of an object from its initializer. If you need a dynamically typed variable, use **variant** or **any** (§8.3).

I implemented **auto** in the winter of 1982/83 but was forced to remove it to preserve C compatibility.

In the context of C++11, people proposed a **typeof** operator to replace **typeof** macros and compiler extensions that had become popular. Unfortunately, the different **typeof** macros were incompatible in their treatment of references, so none of those could be adopted without major code breakage. Introducing a new keyword is always hard because if it is short with a reasonably obvious meaning, it will already have been used many thousands of times. If the suggested keyword is ugly and long, people will dislike it.

Jaakko Järvi, one of the most prolific contributors of interesting libraries to Boost and at the time my colleague at Texas A&M, led the discussion of **typeof**. We realized that the problem with semantics could be summarized as "is **typeof** of a reference the reference itself or the referenced type?" Also, we felt that **typeof** was a bit verbose and error prone:

```
typeof(x+y) z = y+x;
```

Here, I thought I was repeating **x+y**, but I wasn't (with potentially bad effects) and anyway, why should I have to repeat anything? At this point, I realized that I had solved this problem back in 1982; we could eliminate the repetition and "highjack" the keyword **auto**:

```
auto z = y+x;   // z gets the type of y+x
```

In C, and then in C++, **auto** meant "allocate in automatic storage (i.e., on the stack)" and was never used. Looking through many millions of lines of C and C++, we verified that **auto** was only used in test suites and in error, so we could recycle it with my 1982 meaning "get the type of the initializer expression."

That left the problem of what to do with the use cases where we would like to deduce the type of a reference as a reference. This is not uncommon in template-based foundation libraries. We proposed an operator **decltype** with the reference-preserving semantics:

```
template<typename T> void f(T& r)
{
    auto v = r;           // v is a T
    decltype(r) r2 = r;   // r2 is a T&
    // ...
}
```

Why **decltype**? Unfortunately, I don't remember who suggested that name, but I know why:

- We couldn't use **typeof** because that would break a lot of code.
- We couldn't find a nice, short, and unused name.
- **decltype** is sufficiently mnemonic to remember ("declared type") and sufficiently odd not to have been used in existing code.
- **decltype** is reasonably short.

The paper proposing this was from 2003 [Järvi et al. 2003b]. The paper accepted by vote was from 2006 [Järvi et al. 2007]. Jaakko Järvi did most of the detailed work getting **decltype** through the committee. Doug Gregor, Gabriel Dos Reis, Jeremy Siek, and I also helped and appear as co-authors on various papers. It turned out that specifying the exact semantics of **decltype** is far harder than I have made it sound here. Spending years on the details of an apparently simple feature is unfortunately not uncommon. Part of the reason is inherent complexity of features; part is the number of people who have to agree that every detail is designed and specified to their satisfaction.

I consider **auto** to be a purely simplifying feature whereas the primary purpose of **decltype** is to enable sophisticated metaprogramming in foundation libraries. They are however very closely related when looked at from a language-technical point of use.

There were two obvious generalizations of **auto** that I had explored elsewhere [Stroustrup and Dos Reis 2003b]: as return types and argument types. This is obvious because argument passing and value return are defined as initialization in C++. In 2003, when I first presented those ideas to the committee, the members of the evolution working group reacted with undisguised horror. Consider:

```
auto f(auto arg)
{
    return arg;
}

auto x = f(1);                  // x is an int
auto s = f(string("Hello"));    // s is a string
```

Such examples were received more negatively than any other idea I have ever presented to the committee. "It was like the duchess who saw the mouse" was my description: "Eeeeek!" However, that was not the end of the story. C++17 offered **auto** for both arguments and return types for lambdas (§4.3.1), but for functions C++17 offered **auto** only for return types. C++20 added **auto** for function arguments as part of concepts (§6.4), thus finally completing support for my 2003 suggestion.

A weaker use of **auto** was added in C++11 to move the specification of the return type after the arguments. For example, in C++98, we'd write

```
template<typename T>
vector<T>::iterator vector<T>::begin() { /* ... */ }
```

The repetition of **vector<T>::** is annoying and there was no way a return type could depend on an argument type (as is useful in some generic programming). C++11 remedied that and improves readability:

```
template<typename T>
auto vector<T>::begin() -> iterator { /* ... */ }
```

So, after years of work, we had **auto**. It immediately became very popular because it saved programmers from spelling out long type names and from thinking about details of types in generic code. For example:

```
for (auto p = v.begin(); p!=v.end(); ++p) ...  // traditional STL loop
```

It allowed people to line up names:

```
class X {
public:
    auto f() -> int;
    auto gpr(int) -> void;
    // ...
};

void use(int x, char* p)
{
    auto x2 = x*2;    // x2 is an int
    auto ch = p[x];   // ch is a char
    auto p2 = p+2;    // p2 is a char*
    // ...
}
```

There were even calls to use **auto** everywhere or almost everywhere [Sutter 2013b]. This is classic: Every new useful feature is initially overused and misused. After a while, parts of the community find a balance. Articulating such balanced use as a best practice is one of the reasons I (and many others) work on programming guidelines (§10.6). For **auto**, I received many comments about lack of readability when people used it with initializers that did not have obvious types. Consequently, the *C++ Core Guidelines* [Stroustrup and Sutter 2014–2020] (§10.6) has this rule:

*ES.11: Use **auto** to avoid redundant repetition of type names*

My books have advice along the same line [Stroustrup 2013, 2014d]. Consider:

```
auto n = 1;    // OK n is an int
auto x = make_unique<Gadget>(arg);  // OK: x is a std::unique_ptr<Gadget>
auto y = flopscomps(x,3);           // Bad: what might flopscomps() return?
```

It's not 100% clear how to apply such a rule in every case, but it is far better than no rule and leads to more readable code than the absolute rules "never use **auto**!" and "always use **auto**!" Real-world programming tends to take more skill than simple examples illustrating language features.

If **flopscomps()** isn't part of a generic computation, it would be better to explicitly state the desired type. We had to wait for C++20 to use a concept to constrain the return type (§6.3.5):

```
Channel auto y = flopscomps(x,3);   // y can be used as a Channel
```

So, was the work on **auto** worthwhile? It is a small facility that for the simple cases can be implemented in a day, yet it took 4 years to get through the committee. It is not even novel: many languages have had facilities like this for the last 40 years. Even C with Classes had it 35 years ago!

I often despair about the time it takes to get even the smallest feature through the C++ standards committee, and the often-painful discussions it involves. On the other hand, once it is done well, millions of programmers benefit. When something is done really well, the most common comment is "Obvious! What took you so long?"

*4.2.2 Range-**for**.* A range-**for** is a statement that loops over all elements of a sequence from first to last. For example:

```
void use(vector<int>& v, list<string>& lst)
{
    for (int x : v) cout << x << '\n';
    int sum = 0;
    for (auto i : {1,2,3,5,8}) sum+=i; // an initializer list is a sequence
    for (string& s : lst) s += ".cpp"; // use a reference allow modification
}
```

It was first proposed by Thorsten Ottosen (University of Aalborg, Denmark) because "Just about any modern language has some form of 'for each' built into it." [Ottosen 2005]. I don't usually consider "everybody else has one" as a good argument, but in this case the real point is that a simple loop over a range simplifies one of the most common operations and offers some optimization opportunities. Thus, range-**for** fits my overall aims for C++ perfectly. It directly expresses what should be done, rather than spelling out in detail how it is done. It was also a nice clean syntax and the semantics is obvious.

Being simpler and more specific, the range-**for** notation eliminates the possibility for making a few "trivial" but common errors:

```
void use(vector<int>& v, list<string>& lst)
{
    for (int i=0; i<imax; ++i)
        for (int j=0; i<imax; ++j) ...  // bad nested loop

    for (int i=0; i<=max; ++i) ...      // off-by-one error?
}
```

Still, over the years there were changes. Doug Gregor suggested a change to use C++0x concepts which was nice and approved [Ottosen et al. 2007]. I remember him writing the proposal in my office in Texas, but unfortunately, we had to back out that change when we removed C++0x concepts (§6). In 2018, a small change was made to accommodate the infinite sequences supported by the Ranges TS (§9.3.5).

4.2.3 Move Semantics. Traditionally, in C and C++, to get a large amount of data out of a function as a result, you allocate it on the free store (heap, dynamic memory) and pass a pointer to it. Examples are factory functions and functions returning containers (e.g., **vector**s and **map**s. This seemed natural to the community and is suitably efficient. Unfortunately, it is one of the major sources of explicit use of pointers, leading to notational inconvenience, explicit memory management, and hard-to-find errors.

For years many experts had used "trickery" to eliminate this problem by using classes that were handles and were passed around as simple values (often called *value types*). For example:

```
Matrix operator+(const Matrix&, const Matrix&);

void use(const Matrix& m1, const Matrix& m2)
{
    Matrix m3 = m1+m2;
    // ...
}
```

Here, **operator+** offers the conventional mathematical notation and is an example of a factory function returning a large object.

Passing **Matrix**es into a function by **const**-reference was (and is) conventional and efficient. The problem is to return a **Matrix** by value without copying all the elements. As early as 1982, I had partly addressed this problem with an optimization that simply allocated the return value on the caller's stack frame. This worked very well, but it was only an optimization technique and didn't handle more complex return statements. What users needed to return "large objects" by value was a guarantee that large amounts of data were never copied.

To do provide that, it was observed that a "large object" typically was a handle to data on the free store. To avoid copying of large amounts of data we then simply had to ensure that that the constructor used to implement the return must copy the handle, rather than all the elements. The C++11 solution to this problem looks like this:

```
class Matrix {
    double* elements; // pointer to the elements
    // ...
public:
    Matrix(Matrix&& a)  // a move constructor
    {
        elements = a.elements;  // copy the handle
        a.elements = nullptr;   // now a's destructor has nothing to do
    }
    // ...
};
```

A *move* is preferred over a *copy* when the source of an initialization or an assignment is about to be destroyed anyway: a move operation simply "steals" the representation. The **&&** indicates that the constructor is a *move constructor* and **Matrix&&** is called an *rvalue reference*. The **&&** notation for an rvalue reference, called a *forwarding reference* when used for a template parameter, was suggested by John Spicer in a 2002 meeting with Dave Abrahams and Howard Hinnant.

The **Matrix** example is particularly interesting because if we return a pointer from a **Matrix** addition, the conventional mathematical notation (**a+b**) cannot be used.

The use of move semantics can have significant implications for efficiency: it eliminates expensive temporary variables. For example:

```
Matrix mx = m1+m2+m3;  // requires no temporary
string sx = s1+s2+s3;  // requires no temporary
```

I added the **string** example because move semantics was immediately added to all standard-library containers. This speeded up some C++98 programs without a source-code change.

Allowing the designer of a class to define move operations completes the control of object lifetimes and resource management that started in 1979 with the introduction of constructors and destructors. Move semantics is a cornerstone of C++'s model of resource management [Stroustrup et al. 2015]. It's the mechanism that enables simple and cheap movement of objects between scopes.

This important general point may have been obscured by an early emphasis on argument passing, perfect forwarding, and smart pointers. Howard Hinnant, Dave Abrahams, and Peter Dimov proposed the general version of move semantics in 2002 [Hinnant et al. 2004, 2002]:

> *"The rvalue reference can be used to easily add move semantics to an existing class. By this we mean that the copy constructor and assignment operator can be overloaded based on whether the argument is an lvalue or an rvalue. When the argument is an rvalue, the author of the class knows that he has a unique reference to the argument."*

A prominent example was factory functions yielding "smart pointers":

```
template <class T, class A1>
std::shared_ptr<T> factory(A1&& a1)
{
    return std::shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```

The (now) standard-library function **forward** tells the compiler to treat its argument as an rvalue reference, so that a **T**'s move constructor is used to steal that argument (rather than **T**'s copy constructor). It is basically a cast (explicit type conversion) to an rvalue reference.

In C++98, without rvalue references, such "smart pointers" were very difficult to implement. In C++11, the solution is simple [Hinnant et al. 2006]:

```
template <class T>
class clone_ptr
{
private:
    T* ptr;
public:
    // ...
    clone_ptr(clone_ptr&& p)              // move constructor
        : ptr(p.ptr)    // copy representation
    {
        p.ptr = 0;     // "zero out" the source's representation
    }

    clone_ptr& operator=(clone_ptr&& p)   // move assignment
    {
        std::swap(ptr, p.ptr);
        return *this;   // destroy old value of the target
    }
};
```

Soon the move semantics technique was applied to all the standard-library containers, such as **vector**, **string**, and **map**. The **shared_ptr** and **unique_ptr** are smart, but still pointers. I prefer to emphasize move constructors and move assignments that allow efficient movement of large objects (represented as handles) from scope to scope.

The rvalue reference had a rough passage through the committee. Some people thought it likely that the rvalue references and move semantics would not make it into C++11 because the notions were novel and we didn't have suitable terminology for them. Partly as a consequence of the terminology problems [Miller 2010], the use of the term *rvalue reference* in the core language and in the standard library diverged, thus making the draft standard inconsistent. At the Pittsburg meeting in March 2010, I got involved in discussions about that in the Core Working Group (CWG). As the CWG broke for lunch, it seemed to me that "we were headed for an impasse or a mess or both." Instead of going to lunch, I did an analysis of the problems and concluded that there were only two fundamental concepts involved: *has identity* and *can be moved from*. From these two primitives, I derived the conventional categories of *lvalue* and *rvalue* [Barron et al. 1963] as well as three new ones needed to resolve our definitional problems. When the CWG returned, I presented my solution. It was promptly accepted so that we could keep move semantics for C++11 [Stroustrup 2010a].

4.2.4 Resource-Management Pointers. C++11 offered "smart pointers" (§4.2.4):

- **shared_ptr** – representing shared ownership
- **unique_ptr** – representing unique ownership (replacing the C++98 **auto_ptr**)

The addition of these resource-management "smart" pointers" representing ownership had great impact on programming style. For many, their use meant the end of resource leaks and a meaningful decrease in dangling pointer problems. They were the most visible part of the effort to automate resource management and minimize the use of raw pointers for that (§4.2.3).

shared_ptr is a conventional counted pointer: all shared pointers to an object share a counter. When the last shared pointer to an object is destroyed, the object pointed to is destroyed. That's a simple, general, and effective form of garbage collection. It handles non-memory resources correctly (§2.2.1). To deal with circular data structures, there is also a **weak_ptr**. It is often suboptimal, though. People often used (and use) a **shared_pointer** simply to safely return data from a factory function:

```
shared_ptr<Blob> make_Blob(Args a)
{
    auto p = shared_ptr<Blob>(new Blob(a));
    // ... fill *p with lots of good stuff ...
    return p;
}
```

When moving an object out of a function, the use count just goes from 1 to 2 and back to 1. In a multi-threaded program, that typically is a slow operation involving synchronization. Also, when naively used and/or implemented the use-count adds allocation and deallocation overhead.

As expected, **shared_ptr** quickly became very popular and seriously overused in places. Consequently, **unique_ptr** that does not imply any overheads was provided. A **unique_ptr** has exclusive ownership of the object it refers to and simply **delete**s that object when it itself is destroyed.

```
unique_ptr<Blob> make_Blob(Args a)
{
    auto p = unique_ptr<Blob>(new Blob(a));
    // ... fill *p with lots of good stuff ...
    return p;
}
```

The **shared_ptr** and **weak_ptr** were the work of Peter Dimov [Dimov et al. 2003]. Howard Hinnant contributed **unique_ptr** as an improvement on the C++98 **auto_ptr** [Hinnant et al. 2002]. Given that **unique_ptr** is a drop-in replacement for **auto_ptr**, it offered a rare opportunity for (eventually) removing a flawed facility from the standard. The resource management pointers are closely related to the work on move semantics, perfect forwarding, and rvalue references (§4.2.3).

The resource management pointers were (and are) extensively used to hold onto objects so that exceptions (and the like) don't cause a leak (§2.2). For example:

```
void old_use(Args a)
{
    auto q = new Blob(a);
    // ...
    if (foo) throw Bad();  // can leak
    if (bar) return;       // can leak
    // ...
    delete q;    // easy to forget
}
```

That old style using explicit **new** and **delete** is error-prone and not recommended in modern C++ (e.g., the C++ Core Guidelines (§10.6)). Instead, we can write:

```
void newer_use(Args a)
{
    auto p = unique_ptr<Blob>(new Blob(a));
    // ...
    if (foo) throw Bad();  // doesn't leak
    if (bar) return;       // doesn't leak
    // ...
}
```

This is shorter, safer, and quickly became very popular. However, "smart pointers" are still overused: "they are smart but they are still pointers." Unless we actually need a pointer, simply using a local variable is better still:

```
void simplest_use(Args a)
{
    Blob b(a);
    // ...
    if (foo) throw Bad();  // doesn't leak
    if (bar) return;       // doesn't leak
    // ...
}
```

The primary use for smart pointers for representing ownership of resources is object-oriented programming where a pointer (or reference) is used to access objects for which the exact type is unknown at compile time.

4.2.5 Uniform Initialization. For historical reasons, C++ has a variety of notations for initialization and their semantics vary in surprising ways.

From C, C++ inherited three forms of initialization and added a fourth:

```
int x;              // default initialization (for static variables only)
int x = 7;          // value initialization
int a[] = {7,8};    // aggregate initialization
```

```
string s;           // initialization by default constructor
vector<int> v(10);  // initialization by constructor
```

The notion used for initialization can depend on both the type of the object being initialized and the context of the initialization. This is a mess and was recognized as such. For example, why could we initialize a built-in array with a list, but not a **vector**?

```
int a[] = {7,8};         // OK
vector<int> v = {7,8};   // should work (obviously, but it did not)
```

The last example bothered me a lot because it violates the fundamental C++ design aim of providing equally good support for built-in and user-defined types. In particular, by providing better support for array initialization than for **vector**, it encourages the use of the error-prone built-in arrays.

From 2002, when the work on C++0x started, there were many discussions and proposals to address parts of the problem from Daniel Gutson, Francis Glassborow, Alisdair Meredith, Bjarne Stroustrup, and Gabriel Dos Reis. In 2005, Gabriel Dos Reis and I proposed a *uniform initialization* notation that could be used for every type and have the same meaning everywhere in a program [Stroustrup and Dos Reis 2005b]. This notation held the promise of great simplification of user code and the elimination of many subtle errors. The notation was (and still is) based on the list notation using braces. For example:

```
int a = {5};             // built-in type
int a[] {7,8};           // array
vector<int> v = {7,8};   // user-defined type with constructor
```

Braces ({}) are optional for single values and = is optional before a brace initializer list. For uniformity the brace-style initialization is accepted in many places where C++98 didn't allow it or = initialization:

```
int f(vector<int>);
int i = f({1,2,3});  // function argument

struct X {
    vector<int> v;
    int a[];
    X() : v{1,2}, a{3,4} {}  // member initializers
    X(int);
    // ...
}

vector<int>* p = new vector<int>{1,2,3,4};   // new expression
X x {};   // default initialization

template<typename T> int foo(T);
int z = foo(X{1});   // explicit construction
```

Many of these cases, such as an initializer list for an object created by **new**, simply couldn't be done using earlier notations.

Unfortunately, this ideal was only incompletely approximated, so we have a scheme that is only almost uniform. Some people found the use of **{...}** "weird" unless the **...** was a homogeneous list, others clung to the conventional C distinction between aggregates and non-aggregates, and many

were worried that lists without an explicit type marker would lead to ambiguities and errors. For example, this was considered dangerous, but eventually accepted:

```
struct S { string s; int i; };

S foo(S s)
{
    // ...
    return {string{"foo"},13};
}

S x = foo({string{"alpha"},12.3});
```

In one case, the quest for uniform notation was defeated by a common practice. Consider:

```
vector<int> v1(10);          // 10 elements
vector<int> v2 {10};         // 10 elements or 1 element with the value 10?
vector<int> v3 {1,2,3,4,5}; // vector with 5 elements
```

There are many millions of lines of code using size initializers like **vector<int> v1(10)** and from first principles **vector<int> v2 {10}** really is ambiguous. In a new language, I would not use a plain integer to indicate a size, I'd introduce a specific type for that (e.g., **Size** or **Extent**); for example:

```
vector<int> v1 {Extent{10}};  // 10 elements with the default value 0
vector<int> v2 {10};          // 1 element with the value 10
```

However, C++ is not a new language, so we decided to give the initializer list interpretation priority when choosing among constructors. That makes **vector<int> v2 {10}** a **vector** with one element and the interpretation of **{...}** initializers consistent. However, that forces us to use the **(...)** notation when we want to avoid the initializer-list constructor.

One problem with initialization is exactly that it is everywhere so that essentially all problems with programs and the language rules manifest themselves in the context of initialization. Consider:

```
int x = 7.2;   // traditional initialization
int y {7.2};   // brace initialization
```

From the introduction of floating-point numbers into C (about 1974), the value of **x** has been **7**; that is, **7.2** is implicitly truncated resulting in a loss of information. This is a source of errors. The brace initialization does not allow narrowing conversions (here, the truncation). That's good, but makes it harder to upgrade old code:

```
double d = 7.2;
int x = d;   // OK: truncates
int y {d};   // error
```

This is an example of a common problem. People want a simple upgrade path, but unless some effort and change is needed, the result of a perfectly simple upgrade is that the old problems and errors are preserved. Improving a language in wide use is harder than we tend to think.

After many heated debates and many modifications (not all of which I consider improvements), uniform initialization was approved for C++0x in 2008 [Stroustrup 2008b].

As ever, notation was a contentious issue, but eventually we agreed to have a standard-library type **initializer_list** to be used as the argument type of initializer-list constructors. For example:

```
template<typename T> class vector {
public:
    vector(initializer_list<T>);   // initializer-list constructor
    // ...
};

vector<int> v3 {1,2,3,4,5}; // vector with 5 elements
```

Sadly, uniform initialization ({}-initialization) isn't as widely used as I had hoped for. People seem to prefer the familiar notations and familiar bugs. I seem to have fallen prey to the N+1 problem: You have N incompatible and incomplete solutions to a problem, so you add a new and better solution. Unfortunately, the original N solutions don't go away so now you have N+1 solutions. To be fair, there were subtle problems beyond the scope of this paper that were only slowly being remedied (in C++14, C++17, and C++20). My impression is that generic programming and a general push for more concise notation is slowly increasing the appeal of uniform initialization. All standard-library containers (e.g., **vector**) have initializer-list constructors.

4.2.6 ***nullptr.*** In C and C++, the literal **0** denotes the null pointer if assigned to a pointer or compared to a pointer. More confusingly, any integer constant expression that evaluates to zero denotes the null pointer if assigned to a pointer or compared to one. For example:

```
int* p = 99-55-44;      // the null pointer
int* q = 2;             // error: 2 is an int, not a pointer
```

This has annoyed and confused many, so there is a standard-library macro **NULL** (adopted from C), which in standard C++ is defined as **0**. Some compilers warn against **int* p = 0;** but you still can't overload a function for a pointer and an integer without getting ambiguity for **0**.

This could be easily fixed by giving a name to the null pointer, but somehow nobody had gotten around to making a proposal that people could agree to. Sometime in 2003, I was attending a meeting over the phone where people discussed how to name the null pointer. Suggestions, such as **NULL**, **null**, **nil**, **nullptr**, and **0p**, were among the alternatives. As usual, all short and "nice" names had been used thousands of times and thus could not be used without breaking millions of lines of code. Being a bit bored having heard variants of this discussion dozens of times, I was listening with only half an ear. People were saying variations of null pointer, null ptr, nullputter. I woke up and said: "You are all saying **nullptr**; I don't think I have seen that in code."

Herb Sutter and I wrote up that suggestion [Sutter and Stroustrup 2003] and it passed relatively easily in 2007 (after only four minor revisions), so now we can say:

```
int p0 = nullptr;
int* p1 = 99-55-44;      // OK for compatibility
int* p2 = NULL;          // OK for compatibility

int f(char*);
int f(int);

int x1 = f(nullptr);    // f(char*)
int x2 = f(0);          // f(int)
```

I pronounce **nullptr** as "null pointer."

I still think that defining the macro **NULL** to be **nullptr** would have eliminated a significant class of problems, but the committee deemed that change too radical.

4.2.7 ***constexpr*** *Functions.* In 2003, Gabriel Dos Reis and I proposed a radically different and significant better mechanism for doing constant expression evaluation in C++ [Dos Reis 2003]. People were using (typeless) macros and the impoverished C definition of constant expressions. Others were beginning to use template metaprogramming to compute values (§10.5.2). "This is tedious and error-prone" [Dos Reis and Stroustrup 2010]. Our aim was to

- Make compile-time computation type safe
- Generally, improve efficiency by moving computation to compile-time
- Support embedded systems programming (especially ROMs)
- Directly support metaprogramming (as opposed to *template* metaprogramming (§10.5.2))
- Make compile-time programming very similar to "ordinary programming"

The idea was simple: allow functions prefixed with **constexpr** to be used in constant expressions and also allow simple user-defined types, called literal types, to be used in constant expressions. A literal type is basically a type for which all operations are **constexpr**.

Consider an application where we for efficiency, ROMability, or reliability wanted to use a unit system [Dos Reis and Stroustrup 2010]:

```
struct LengthInKM {
    constexpr explicit LengthInKM(double d) : val(d) { }
    constexpr double getValue() { return val; }
private:
    double val;
};

struct LengthInMile {
    constexpr explicit LengthInMile(double d) : val(d) { }
    constexpr double getValue() { return val; }
    constexpr operator LengthInKM() { return LengthInKM(1.609344 * val); }
private:
    double val;
};
```

Given that, we can make a table of constants without fear of unit errors or conversion errors:

```
LengthInKM marks[] = { LengthInMile(2.3), LengthInMile(0.76) };
```

The traditional solution would either require more run time or have the programmer work out the values on a doodle pad. My interest in unit systems was stimulated by the loss of the 1999 Mars Climate Orbiter due to an undetected unit mismatch [Stephenson et al. 1999].

A **constexpr** function can be evaluated at compile time, so it cannot access non-local objects (they don't exist at compile-time) so C++ acquired a kind of pure functions.

Why did we require that programmers should use **constexpr** to mark functions that can be executed at compile time? In principle, the compiler can figure out what can be computed at compile time, but without an annotation, users would be at the mercy of variations in cleverness of compilers and a compiler would need to keep bodies of all functions around "forever" in case they were needed for constant expression evaluation. We picked the word **constexpr** because it was sufficiently mnemonic, yet "sufficiently odd" not to break existing code.

In a few places, C++ requires constant expressions (e.g., array bounds and case labels). In addition, we can require a variable to be initialized at compile time by declaring it **constexpr**:

```
constexpr LengthInKM marks[] = { LengthInMile(2.3), LengthInMile(0.76) };
```

```
void f(int x)
{
    int y1 = x;
    constexpr int y2 = x;   // error: x is not a constant
    constexpr int y3 = 77;  // OK
}
```

The early discussions focused on simple examples of performance and embedded systems. Only much later (from about 2015) did **constexpr** functions become a mainstay of metaprogramming (§10.5.2). C++14 allowed local variables, and therefore loops, to be used in **constexpr** functions; before that they had to be purely functional. C++20 (finally, about 10 years after it was first proposed) allows literal types as value template parameter types [Maurer 2012]. Thus, C++20 will be very close to the original (1979) goal of being able to use user-defined types wherever built-in types can be used (§2.1).

The **constexpr** functions quickly became wildly popular. They are all over the C++14, C++17, and C++20 standard libraries and there is a constant stream of suggestions for allowing more language constructs in **constexpr** functions, for applying **constexpr** to more functions in the standard library, and for more support for compile-time computation (§9.3.3).

However, **constexpr** functions were not easy to get into the standard. They were repeatedly deemed useless and unimplementable. Implementing **constexpr** functions obviously required work on older compilers, but soon the writers of all major compiler proved the "unimplementable" claims wrong. The discussions about **constexpr** were about the most heated and unpleasant ever. It took four years to get the initial version through the standards process [Dos Reis and Stroustrup 2007] and twelve more to complete the process.

4.2.8 User-Defined Literals. "User-defined literals" is very a small feature. However, it fits into the general thrust to make the support of user-defined types similar to what built-in types receive. Built-in types have literals, for example, **10** is an integer and **10.9** is a floating-point number. I tried to convince people that the equivalent for user-defined types was explicit use of constructors; for example, **complex<double>(1.2,3.4)** is the **complex** equivalent to a literal. However, many didn't consider that good enough: the notation is not conventional and there was no guarantee that the constructor would be evaluated at compile time (though it was from the earliest days). For **complex**, people wanted **1.2+3.4i**.

Compared to other problems, this didn't seem important, so nothing happened for decades. One day in 2006, David Vandevoorde (EDG), Mike Wong (IBM), and I were out for a good dinner in a Chinese restaurant in Berlin. We were talking shop, and a design emerged on a paper napkin. The discussion was prompted by the need for suffixes in an IBM proposal for decimal-floating-point that eventually became its own international standard [(editor) 2007]. After significant mutation, that design became *user-defined literals* (often referred to as *UDL*s) in 2008 [McIntosh et al. 2008]. The significant development that made UDLs interesting just then was the progress of the **constexpr** function proposal (§4.2.7). Given that, we could guarantee compile-time evaluation.

As is often the case, finding an acceptable notation was a problem. We decided that the cryptic **operator""** was acceptable as a notation for *literal operator*, after all **""** is a literal. Then, **""x** is the notation for a literal followed by the suffix **x**. Given that and an **Imaginary** type for use with **complex** numbers, we can define:

```
constexpr Imaginary operator""i(long double x) { return Imaginary(x); }
```

Now, **3.4i** is an **Imaginary** and **1.2+3.4i** is **complex<double>(1.2,3.4)**. Mission accomplished!

The language technical details are quite quirky, but I consider that reasonable for a relatively rarely-used feature. Even when UDLs are used heavily, there are few definitions of literal operators. What matters most is the elegance and ease of use of the suffixes. For many types, it is important that the conversion from the built-in types to user-defined types can be done at compile time.

Naturally, people used UDLs to define literals for many useful types, including several from the standard library (e.g., **s** for seconds and **s** for **std::string**). There was discussion about support for binary literals, and Peter Sommerlad (HSR) suggested what I consider a candidate for a "best abuse of the rules" award: define **operator""_01(long int)** appropriately and **101010_01** is a binary literal! When the astonishment and laughter had died down, the committee decided to define binary literals in the language itself and use **0b** as the prefix meaning "binary" (e.g., **0b101010**) in analogy to the use of **0x** to mean "hexadecimal" (e.g., **0xDEADBEEF**).

4.2.9 Raw literals. This is a rare simple feature with the single purpose of providing an alternative to an error-prone notation. Like C, C++ uses the backslash as an escape character. This means that to represent a backslash in a string literal, you need to use a double backslash (\\) and when you want a double quote in a string, you need to use \". However, the usual regular expression patterns use backslashes extensively and double quotes are common, so patterns quickly get messy and error prone. Consider a simple example (a US postal code):

```
regex pattern1 {"\\w{2}\\s*\\d{5}(-\\d{4})?"}; // ordinary literal

regex pattern2 {R"(\w{2}\s*\d{5}(-\d{4})?)"};  // raw literal
```

The two patterns are identical. The *raw literal* **R"(** ... **)"** bracketing can be elaborated to hold more complicated patterns, but when you use regular expressions (§4.6) the simplest version is sufficient and a major convenience. Providing raw literals is a minor detail, of course, but (similar to digit separators (§5.1) much loved by people who need many literals.

The raw literals were proposed by Beman Dawes in 2006 [Dawes 2006] based on experience with **boost::regex** [Maddock 2002].

4.2.10 Attributes. Attributes provide a way of associating essentially arbitrary information with an entity in a program. For example:

```
[[noreturn]] void forever()
{
    for(;;) {
        do_work();
        wait(10s);
    }
}
```

The **[[noreturn]]** informs a compiler or other tool that **forever()** is never supposed to return so that it can suppress warnings against the lack of a return. An attribute is bracketed by **[[** ... **]]**.

Attributes were first proposed in 2007 by Alisdair Meredith [Meredith 2007], the head of the Library Working Group, to eliminate incompatibilities among vendor attribute notations (e.g., **__declspec**s and **__attribute__**s) that complicated library implementation. In response to that, Jens Maurer and Michael Wong did an analysis of the problems and proposed the **[[** ... **]]** syntax based on an implementation that Michael had done for IBM's XL compiler [Maurer and Wong 2007]. In addition to standardizing a multitude of non-portable practices, this would also allow language extension to be done with fewer keywords and new keywords are always controversial.

The proposal mentioned possible uses: explicit syntax to override virtual functions, dynamic libraries, user-controlled garbage collection, thread local storage, controlling alignment, identifying classes that are "plain old data," defaulted and deleted functions, strong enums, strong typedefs, pure side-effect free functions, final overrides, sealed classes, fine grained control over concurrency, support for runtime reflection, and lightweight support for contract programming. Many more were mentioned in early discussions.

"Attributes" is certainly a feature that makes certain things simpler, but I am not sure that it encourages good design or that the "things" it makes simpler are always what yield the most benefits. I had visions of attributes opening the floodgates for a mass of unrelated, half understood, minor features. Instead of proposing a feature to WG21, anyone would be able to add a attribute to a compiler and lobby for its adoption everywhere. Many programmers just love such minor features. The elimination of the need to introduce keywords and to modify the grammar would lower the barrier to entry. So would the inevitable insufficient attention to feature interactions and overlapping, but incompatible, similar features in different compilers. This had already happened with proprietary extensions, but I considered those inevitable, localized, and often transient.

To attempt to limit potential damage, we decided that an attribute should imply no change of a program's semantics. That is, a compiler that ignored an attribute would do no harm. Over the years, this "rule" almost worked. Most standard attributes – though not all – have no semantic effects even though they help with optimizations and error detection.

In the end, most of the initially suggested uses of attributes were addressed by ordinary syntax and language rules.

C++11 added the standard attributes **[[noreturn]]** and **[[carries_dependency]]**.

C++17 added **[[fallthrough]]**, **[[nodiscard]]**, and **[[maybe_unused]]**.

C++20 added **[[likely]]**, **[[unlikely]]**, **[[deprecated(message)]]**, **[[no_unique_address]]**, and **[[using: ...]]**.

I still see attribute proliferation as a potential danger, but so far the floodgates haven't opened. The C++ standard-library uses attributes liberally; **[[nodiscard]]** is particularly popular, especially to protect against potential leaks from unused return values that are resource handles.

The attribute syntax was adopted for the (failed) C++20 contract design (§9.6.1).

4.2.11 Garbage Collection. From the earliest days of C++, *optional* garbage collection was considered (for a variety of definitions of "optional") [Stroustrup 1993, 2007]. After much debate, C++11 offered an interface to conservative garbage collectors designed by Mike Spertus and Hans-J Boehm [Boehm and Spertus 2005; Boehm et al. 2008]. However, few people noticed that and fewer used garbage collection (despite the availability of good collectors). The approach was to

> *support both garbage collected implementations and reachability-based leak detectors. This is done by giving undefined behavior to programs that "hide a pointer" by, for example, xor-ing it with another value, and then later turn it back into an ordinary pointer and dereference it.* [Boehm et al. 2008]

This work became a boon to the precise specification of the semantics of C++ and a few uses of garbage collection in C++ exist (e.g., by Macaulay2 [Eisenbud et al. 2001; Macaulay2 2005–2020]). However, garbage collectors don't address non-memory resources and the C++ community in general chose to use combinations of resource-management pointers (§4.2.4) and RAII (§2.2.1).

4.3 C++11: Improving Support for Generic Programming

Generic programming (and its offspring template metaprogramming (§10.5.2)) became a run-away success with C++98. Its use seriously strained the language and inadequate language support led to baroque programming techniques and horrendous error messages. It is a testimony of the utility

of generic programming and metaprogramming that so many sane programmers were willing to suffer to gain the benefits. Those benefits were

- Flexibility beyond what could be obtained in C-style or object-oriented style
- Cleaner code
- Finer granularity of static type checking
- Efficiency (mostly from inlining, having the compiler see code from several sources at once, and better type checking)

The main new features supporting generic programming in C++11 are:

- §4.3.1: Lambdas
- §4.3.2: Variadic templates
- §4.3.3: **template** aliases
- §4.3.4: **tuple**s
- §4.2.5: uniform initialization

Concepts should have been the centerpiece of improved support for generic programming in C++11, but that didn't happen (§6.2.6). We had to wait until C++20 (§6.4).

4.3.1 Lambda. BCPL allowed blocks of code as expressions, but to save space in the compiler, Dennis Ritchie didn't adopt this feature into C. I followed C in this, but added **inline** functions to (re)gain the ability to get code executed without the cost of a function call. However, this still didn't offer the ability to

- Write code exactly where it was needed (usually as a function argument).
- Access the context of the code from within the code.

During the work on C++98, there had been proposals for local functions to address that second point, but they were voted down as a likely bug source.

Instead of allowing function definitions inside functions, C++ relied on functions defined inside classes. That allowed the context of a function to be represented as class members and function objects became very popular. A *function object* is simply a class with an application operator (**operator()()**). It was a very efficient and effective technique and I (and others) expressed the opinion that named objects resulted in clearer code than unnamed operations. However, that is only the case when something can be given a reasonable name outside the context of its use and especially if it is used more than once.

In 2002, Jaakko Järvi and Gary Powell wrote the Boost lambda library [Järvi and Powell 2002] that allowed us to write something like this

```
find_if(v.begin(), v.end(), _1<i);  // find element with a value less than i
```

Here, **_1** is the name of a first argument to the code fragment **_1<i** and **i** is a variable in the enclosing scope. The **_1<i** expands into a function object, where **i** is bound to a reference and **_1** becomes the argument to an **operator()()**:

```
struct Less_than {
  int& i;
  Less_than(int& ii) :i(ii) {}  // bind to i
  bool operator()(int x) { return x<i; }  // compare to argument
}
```

The lambda library was a masterpiece of early template metaprogramming (§10.5.2) and very convenient and popular. Unfortunately, it wasn't particularly efficient. For years, I tracked its performance relative to hand-coded equivalents and found a fairly consistent 2.5 times overhead. I could not recommend something that was convenient, but slow. Doing so would damage C++'s

reputation as a language for efficient code. Obviously, the slowdown was partly a result of poor optimization, but for this and other reasons a group of us, led by Jaakko Järvi, decided to propose lambda expressions as a language feature [Willcock et al. 2006]. For example:

```
template<typename Oper>
void g(Oper op)
{
    int xx = op(7);
    // ...
}

void f()
{
    int y = 3;
    g(<>(int x) -> int {return x + y;});  // call g() with a lambda argument
}
```

Here, **xx** will become **3+7**.
The <> was the lambda introducer. We did not dare to propose a new keyword.
That proposal generated quite a lot of excitement and many lively discussions:

- Should the syntax be expressive or terse?
- What names from which scope can a lambda refer to? [Crowl 2009].
- Should the function object generated from a lambda be mutable? Not by default.
- Can a lambda be polymorphic? Not until C++14 (§5.4).
- What is the type of a lambda? A unique type unless it basically is a local function.
- Can a lambda have a name? No. If you need a name just assign it to a variable.
- Are names bound by value or by reference? You have a choice.
- Can variables be moved into a lambda (as opposed to copied)? Not until C++14 (§5).
- Would the syntax clash with various non-standard extensions? (Not seriously).

By the time lambdas were approved in 2009, the syntax had mutated and become more conventional [Vandevoorde 2009]:

```
void abssort(float *x, unsigned N)
{
    std::sort(x, x+N,
        [](float a, float b) { return std::abs(a) < std::abs(b); }
             );
}
```

The switch from <> to **[]** was suggested by Herb Sutter and implemented by Jonathan Caves. The change was in parts motivated by the need for a simple way of designating which names from the surrounding scopes could be used by the lambda. Herb Sutter recalls:

> *I was motivated by needing lambdas for my parallel algorithms project ... and seeing that the lambdas EWG had adopted were just butt-ugly to use and poorly designed from a syntax consistency/cleanness point of view (e.g., captures appeared in two separate places, syntax elements were used inconsistently, the ordering was wrong because all the "constructor" elements should come first followed by the "operator" elements that get invoked later, and various smaller issues).*

By default, a lambda cannot refer to names in the local environment, so they are just ordinary functions. However, we can specify that a lambda should "capture" some or all of the variables from

its environment. Callbacks are a common use case for lambdas because often the action is unique and need to use some information from the context of the code installing the callback. Consider:

```
void test()
{
    string s;
    // ... compute a suitable value for s ...
    w.foo_callback([&s](int i){ do_foo(i,s); });
    w.bar_callback([=s](double d){ return do_bar(d,s); });
}
```

The **[&s]** says that **do_foo(i,s)** can use **s** and that **s** is passed ("captured") by reference. The **[=s]** says that **do_bar(d,s)** can use **s** and that **s** is passed by value. If the callback is invoked on the same thread as **test**, **[&s]** capture is potentially efficient because **s** is not copied. If the callback is invoked on a different thread, **[&s]** capture could be a disaster because **s** could go out of scope before it was used; in that case, we want a copy. A **[=]** capture list means "copy all local variables into the lambda." A **[&]** capture list means "the lambda can refer to all local variables by reference" and implies that the lambda can be implemented as simply a local function. The flexibility of the capture mechanism has proven very valuable. The capture mechanism allows control of what names can be referred to from a lambda, and how. This is an answer to the 1990s worries about local functions being error prone.

The implementation of a lambda is basically that the compiler builds a suitable function object and passes it. The captured local variables become members initialized by a constructor and the lambda's code becomes the function object's application operator. For example, the **bar_callback** becomes:

```
struct __XYZ {
    string s;
    __XYZ(const string& ss ) : s{ss} {}
    int operator()(double d) { return do_bar(d,s); }
};
```

The return type of a lambda can be deduced from its return statement. If there is no return statement, the lambda doesn't return anything.

I classify lambdas as support for generic programming because one of the most common uses – and a major motivation – was for the use as arguments to STL algorithms:

```
// sort in descending order:
sort(v.begin(),v.end(),[](int x, int y) { return x>y; });
```

As such, lambdas added significantly to the attraction of generic programming.

After C++11, C++14 added generic lambdas (§5.4) and move capture (§5).

4.3.2 Variadic Templates. In 2004, Douglas Gregor, Jaakko Järvi, and Gary Powell (all then at Indiana University) proposed a feature called *variadic templates* [Gregor et al. 2004] to

> "*directly addresses two problems:*
> - *The inability to instantiate class and function templates with an arbitrarily-long list of template parameters.*
> - *The inability to pass an arbitrary number of arguments to a function in a type-safe manner.*"

These are important goals, but I initially found the solution overly complex, the notation too cryptic, and the programming style too recursive for my taste. However, after a brilliant presentation by

Doug Gregor in 2004, I changed my mind and backed the proposal to the hilt so that variadic templates had a relatively smooth passage through the committee. Part of what convinced me was measurements of compile times comparing variadic templates with (then) current workarounds. Variadic templates was a major (sometimes 20 times) improvement over what was an increasingly serious problem as template metaprogramming was beginning to see major use (§10.5.2). Unfortunately, variadic templates became so popular and an essential part of the C++ standard library, so the compile-time problem reappeared. However, that penalty for success was (then) still in the far future.

The basic idea is to build up a *parameter pack* recursively and then use it by another recursive pass. The recursive technique was necessary because each element of a parameter pack has its own type (and size).

Consider an implementation of **printf** that can handle every type that can be output using the standard-library iostream output operator << [Gregor 2006]:

> To build our type-safe **printf()**, we use the following strategy: write out the string up until the first format specifier is reached, print its corresponding value, then call **printf()** recursively to print the rest of the string and remaining values.

```
template<typename T, typename... Args>
void printf(const char* s, const T& value, const Args&... args)
{
    while (*s) {
        if (*s == '%' && *++s != '%') { // ignore the char that follows
                                        // the '%': we already know the type!
            std::cout << value;
            return printf(++s, args...);
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
 }
```

The **<typename T, typename... Args>** specifies a traditional list with a head (**T**) and a tail (**Args**). Each call handles the head and then calls itself with the tail. Ordinary characters are simply printed and the format character **%** indicates that an argument is to be printed. The test case offered by Doug (then a resident of Indiana) was:

```
const char* msg = "The value of %s is about %g (unless you live in %s).\n";

printf(msg, std::string("pi"), 3.14159, "Indiana");
```

That prints

```
The value of pi is about 3.14159 (unless you live in Indiana).
```

One nice thing about this implementation is that, in contrast to the standard **printf**, user-defined types are handled as well as built-in ones. By using << it also avoids errors from mismatch between the type indicator and the argument type, e.g., **printf("%g %c","Hello",7.2)**.

The technique illustrated by this **printf** is a basis for the C++20 **format** (§9.3.7).

The weakness of variadic templates is that they can easily lead to code bloat as N parameters imply N instantiations of the template.

4.3.3 Aliases. The C mechanism for defining an alias for a type is a **typedef**. For example:

```
typedef double (*pf)(int);  // pf is a pointer to a function
                            // taking an int and returning a double
```

This is a bit convoluted, but type aliases are very useful and ubiquitous in C and C++ code. From the first days of C++ templates, people were considering whether we could have *typedef templates* and if so, what they would be. In 2002, Herb Sutter proposed a solution [Sutter 2002]:

```
template<typename A, typename B> class X { /* ... */ };
template<typename T> typedef X<T,int> Xi;  // the alias
Xi<double> Ddi;                            // equivalent to X<double,int>
```

Building on that and after lengthy email reflector (archived email group) discussions, Gabriel Dos Reis (then at INRIA in France) and Matt Marcus (Adobe) resolved some nasty problems related to specialization and introduced a simplified syntax for what David Vandevoorde named *alias templates* [Dos Reis and Marcus 2003]. For example:

```
template<typename T, typename A> class MyVector { /* ... */};
template <typename T> using Vec = MyVector<T, MyAlloc<T> >;
```

The **using** syntax, where the name being introduce always comes first, was my suggestion.

Finally, together with Gabriel Dos Reis, I generalized this to an (almost) complete aliasing mechanism, that was accepted [Stroustrup and Dos Reis 2003c]. This gave people a choice of notation even where templates are not involved:

```
typedef double (*analysis_fp)(const vector<Student_info>&);

using analysis_fp = double (*)(const vector<Student_info>&);
```

Type and template aliases are key to some of the most effective techniques for zero-overhead abstraction and modularization. An alias allows a user to refer to a set of standard names while various implementations use their own (differing) implementation techniques and names. That way we can have true zero-overhead abstractions while maintaining convenient user interfaces. Consider a real-world example from a communications library (using the concepts TS [Sutton 2017] and C++20 notational simplifications):

```
template<InputTransport Transport, MessageDecoder MessageAdapter>
class InputChannel {
public:
   using InputMessage = MessageAdapter::InputMessage<Transport::InputBuffer>;
   using MessageCallback = function<void(InputMessage&&)>;
   using ErrorCallback = function<void(const error_code&)>;
   // ...
};
```

We have found concepts and aliases invaluable for managing such composition at scale.

The user-interface for **InputChannel** primarily consists of the three aliases **InputMessage**, **MessageCallback**, and **ErrorCallback**, that are initialized from its template arguments.

The **InputChannel** needs to initialize its transport layer, represented by a **Transport** object. However, the **InputChannel** should not know about the details of the transport implementation, so it cannot directly initialize its **Transport** member. Variadic templates (§4.3.2) come to the rescue:

```
template<InputTransport Transport, MessageDecoder MessageAdapter>
class InputChannel {
public:
    template<typename... TransportArgs>
        InputChannel(TransportArgs&&... transportArgs)
            : _transport {forward<TransportArgs>(transportArgs)... }
        {}
    // ...
    Transport _transport;
};
```

Without variadic templates, we would need to either define a common interface for initialization of transports or expose the transport to users.

I consider this an excellent example of how the C++11 features (plus concepts) combine to provide an elegant zero-overhead solution to a hard problem.

4.3.4 ***tuples.*** C++98 had a **pair<T,U>** template; it was mainly used for returning pairs of values, such as two iterators or a pointer and a success indicator. In 2002, with reference to Haskell, ML, Python, and Eiffel, Jaakko Järvi proposed to generalize this idea to a **tuple** [Järvi 2002]:

> "*Tuples are fixed-size heterogeneous containers. They are a general-purpose utility, adding to the expressiveness of the language. Some examples of common uses for tuple types are:*
> - *Return types for functions that need to have more than one return type.*
> - *Grouping related types or objects (such as entries in parameter lists) into single entities.*
> - *Simultaneous assignment of multiple values.*"

For a specific purpose, a class with appropriate names for members and semantic relations among members is usually best. Alisdair Meredith forcefully presented that view in the committee, discouraging us from overusing unnamed types in interfaces. However, when writing generic code, bundling values together in a tuple and manipulating that tuple as an entity often simplify implementation. Tuples are most useful for intermediate groupings that are not worthy of a name and not worth designing a class for.

For example, consider a matrix decomposition that simply returns three values:

```
auto SVD(const Matrix& A) -> tuple<Matrix, Vector, Matrix>
{
    Matrix U, V;
    Vector S;
    // ...
    return make_tuple(U,S,V);
};

void use()
{
    Matrix A, U, V;
    Vector S;
    // ...
    tie(U,S,V) = SVD(A); // using the tuple form
}
```

Here, **make_tuple()** is a standard-library function that makes a **tuple** with element types deduced from its function arguments and **tie()** is a standard-library function that assigns a **tuple**'s members to named variables.

In C++17 using structured bindings (§8.2), this example reduces to:

```
auto SVD(const Matrix& A) -> tuple<Matrix, Vector, Matrix>
{
    Matrix U, V;
    Vector S;
    // ...
    return {U,S,V};
};

void use()
{
    Matrix A;
    // ...
    auto [U,S,V] = SVD(A); // using the tuple form and structured bindings
}
```

A further notational simplification was proposed for C++20 [Spertus 2018], but didn't make it on time:

```
tuple SVD(const Matrix& A) // deduce the tuple template arguments
                           // from the return statement
{
    Matrix U, V;
    Vector S;
    // ...
    return {U,S,V};
};
```

Why is **tuple** not a language feature? I don't remember that question being asked at the time, though someone must have thought of it. It has long (since 1979) been the policy not to add a feature to C++ if we could reasonably add it as a library; if not, improve the abstraction mechanisms to make it possible. There are obvious advantages to this policy:

- It is usually far easier to experiment with a library than with a language feature, so we get better feedback sooner.
- A library can see serious use long before all compilers can be upgraded to support the new feature.
- Improvements to the abstraction mechanisms (classes, templates, etc.) help beyond the immediate problem.

For **tuple** we had **boost::tuple** to build on and people were proud of the clever implementations. There appears not to be run-time efficiency reasons to prefer a language implementation over a library implementation in this case. That's somewhat impressive.

Parameter packs is an example of tuples provided with a compiler-supported interface (§4.3.2). Tuples are extensively used in libraries interfacing C++ with other languages (e.g., Python).

4.4 C++11: Increase Static Type Safety

There are two reasons to rely on static type safety

- Make it clear what is intended
  - to help the programmer directly express ideas
  - to help the compiler catch more errors
- Help the compiler generate better code.

The second point is a consequence of the first. Inspired by Simula, my aim for C++ was to provide a flexible and extensible static type system for C++. The aim is not just type safety, but the ability to directly express fine-grained distinctions, such as physical unit checking (§4.2.7). A program written exclusively using built-in types, such as integers and floating-point types, can be type-safe without delivering significant benefits. Such code doesn't directly express the concepts of the application. In particular, an **int** or a **string** can represent just about anything so that passing one doesn't give a clue about the semantics of the value passed.

The C++11 improvements relating directly to type safety are:

- Type-safe interface to threading and locking – avoid POSIX and Windows reliance on **void**** and macros in concurrent code (§4.1.2)
- Range-**for** – avoid erroneous specification of ranges (§4.2.2)
- Move semantics – addresses the problem of overuse of pointers (§4.2.3)
- Resource management pointers (**unique_ptr** and **shared_ptr** (§4.2.4))
- uniform initialization – make initialization more general, more uniform, and safer (§4.2.5)
- **constexpr** – eliminate many uses of (untyped and unscoped) macros (§4.2.7)
- User-defined literals – make user-defined types more like built-in types (§4.2.8)
- **enum class**es – eliminate some weakly-typed practices involving integer constants
- **std::array** – avoid unsafe "decay" of built-in arrays to pointers

It has repeatedly been suggested that the committee should improve type safety by banning unsafe features (e.g., abandoning "C-isms," such as built-in arrays and casts). However, attempts to remove features (to "deprecate" them) have repeatedly failed as users ignored the warnings about removal and insisted that their implementation providers continue to supply them. A more viable approach seems to be to provide guidelines for use and the means to enforce them while leaving the standard itself compatible with earlier standards (§10.6).

4.5 C++11: Support for Library Building

Foundational libraries for C++ are often designed to compete in performance and ease-of-use with built-in facilities in C++ and other languages. This is where subtleties of lookup-rules, overload resolution, access control, template instantiation rules, and more combine to yield great expressive power but also to expose fearsome complexity.

4.5.1 Implementation Techniques. Some implementation techniques are essentially "dark arts" to which non-experts should not be exposed. Most programmers can happily write good C++ for years without knowing the sophisticated trickery and esoteric techniques. Unfortunately, beginners flock to examine the most horrendously specialized code and take great pride in explaining it (often erroneously) to others. Bloggers and speakers enhance their reputations by showing off hair-raising examples. This is a major source of C++'s reputation for complexity. In other languages, such optimization opportunities are either not offered or the trickery is hidden inside optimizers.

I cannot go into details here, so I will mention just one technique that emerged as key during the development of C++11 and has become almost universal in template-based libraries (including the C++ standard library) known by its odd acronym: *SFINAE* (Substitution Failure Is Not An Error).

How do you provide an operation if and only if some predicate is true? Concepts provide that for C++20 (and have been available in GCC since 2015), but in the early 2000s, people had to rely on obscure language rules. For example:

```
template<typename T, typename U>
struct pair {
    T first;
    U second;
```

```
    // ...
    enable_if<is_copy_assignable<T>::value
              && is_copy_assignable<U>::value,pair&>::type
        operator=(const pair&);
    // ...
};
```

Now, **pair** has a copy assignment if and only if both of its members have one. This is extraordinarily ugly, but – in the absence of concepts – also extraordinarily useful for defining and implementing foundational libraries.

The idea is that **enable_if<...,pair&>::type** will become plain **pair&** if the members have copy assignments and fail to instantiate otherwise (because **enable_if** didn't provide a return type for the assignment). This is where SFINAE comes in: failure to instantiate is not an error; the result of failure is as if the whole declaration wasn't there.

The **is_copy_assignable** is a **type trait**. C++11 provided dozens of such traits to allow programmer to inquire about the properties of types at compile time.

The **enable_if** metafunction was pioneered by Boost and is part of C++11. A plausible implementation is

```
template<bool B, typename T = void>
struct enable_if {}; // the false case: no mention of 'type'

template<typename T>
struct enable_if { typedef T type; };  // type T
```

The precise rules for SFINAE are very subtle and hard to craft, but under steady pressure from users, they became simpler and more general during the development of C++11. As a side effect, this significantly improved the internals of compilers as they had to be able to back out of failed template instantiations without side effects. This seriously discouraged their use of non-local state.

4.5.2 Metaprogramming Support. The first decade of the 2000s were a bit like the Wild West for metaprogramming in C++ with new techniques and applications being tried with essentially no support beyond the basic template mechanisms. Those mechanisms were sorely stressed. Error messages were atrocious, compile times often extraordinarily long, and it was easy to exhaust the compiler's resources (e.g., memory, recursion depth, and identifier length). Also, individuals repeatedly re-discovered problems and re-invented basic techniques. Clearly, better support was needed. Attempts to help took two complementary (at least in theory) tracks:

- *Language*: concepts (§6), compile-time functions (§4.2.7), lambdas (§4.3.1), template aliases (§4.3.3), and more precise specification of template instantiation (§4.5.1).
- *Standard library*: **tuple**s (§4.3.4), type traits (§4.5.1), and **enable_if** (§4.5.1).

Unfortunately, concepts failed for C++11 (§6.2) leaving the field open for (often horribly complex and error-prone) workarounds, typically involving traits and **enable_if** (§4.5.1).

*4.5.3 **noexcept** Specifications.* The original exception design did not have any way to indicate that an exception might be thrown from a function. I still consider that the correct design. To get exceptions accepted for C++98, we had to add exception specifications, a way of listing which exceptions a function could throw [Stroustrup 1993]. Their use was optional and they were checked at run time. As I feared, that led to maintenance problems, run-time overhead as an exception was repeatedly checked along the unwinding path, and bloated source code. For C++11, exception specifications were deprecated [Gregor 2010] and for C++17, we finally removed them (unanimously).

There was always a group of people who wanted compile-time checks of which exceptions a function could throw. That, of course, works well in type theory, with small programs, with fast compilers, and with full control of the source code. The committee repeatedly rejected that idea on the grounds that it doesn't scale to million-line programs developed and maintained by dozens (or more) organizations [Stroustrup 1994]. See also (§7.4).

Without exception specifications, library implementers faced a performance problem: In many important cases, a library implementer needs to know if a copy operation can throw. If it can, taking a copy is necessary to avoid leaving an invalid object behind (violating the exception guarantees [Stroustrup 1993]). If not, we can write straight into a target. The performance difference can be very significant and the simplest exception specification **throw()**, throw nothing, was helpful here. So, as exception specifications were pushed into disuse and eventually removed from the standard, we introduce the notion of **noexcept** based on proposals from David Abrahams and Doug Gregor[Abrahams et al. 2010; Gregor 2010; Gregor and Abrahams 2009].

A **noexcept** function is still dynamically checked. For example:

```
void do_something(int n) noexcept
{
    vector<int> v(n);
    // ...
}
```

If **do_something()** throws, the program is terminated. Doing so happens to have very close to zero overhead because it simply short-circuits the usual exception propagation mechanism. See also (§7.3).

There is also a conditional version of **noexcept** so that people can write templates where the implementation depends on whether a parameter may throw. That's the original use case that motivated **noexcept**. For example, here is an implementation of a move constructor for **pair** that is defined if and only if both elements of the **pair** have move constructors:

```
template<typename First, typename Second>
class pair {
    // ...
    template <typename First2, typename Second2>
    pair(pair<First2,Second2>&& rhs)
        noexcept(is_nothrow_constructible<First,First2&&>::value
              && is_nothrow_constructible<Second,Second2&&>::value)
    : first(move(rhs.first)),
      second(move(rhs.second))
    {}
    // ...
};
```

The **is_nothrow_constructible**<> is one of the C++11 standard-library type traits (§4.5.1).

Writing optimal code at this relatively low and very general level is non-trivial. At the foundational level, knowing whether to bitwise copy, move, or memberwise copy can make large factors of difference.

4.6 C++11: Standard-Library Components

C++ has always had a tiny standard library compared to other modern languages. Furthermore, most standard-library components are foundational rather than addressing application-level tasks. However, C++11 added a few key library components supporting specific tasks:

- **thread**s – thread-and-lock-based concurrency
- **regex** – regular expressions
- **chrono** – time
- **random** – random number generators and distributions

Compared to the massive corporate support libraries this is obviously pitiful, but the components are of high quality and massive compared to what had previously been offered as standard for C++.

These components all provide significant help to programmers for the tasks they were designed for. Unfortunately, the backgrounds of these library components show up as differences in interface styles; there was no coherent overall design philosophy beyond aiming for flexibility and high performance. There were no articulated criteria for incorporating a component into the C++11 standard library (C++98 had some [Stroustrup 1994]). Rather, components were adopted from what was available and had proven successful in the community. Many came from Boost (§2.3).

If you need to use regular expressions, the addition of **regex** to the standard library was a massive improvement. Similar, adding unordered containers (hash tables), such as **unordered_map**, saved many programmers a lot of tedious work and yielded better programs. However, these library components didn't seriously affect the way people organized their code, so I will not discuss such library components in detail.

The **regex** library was primarily the work of John Maddock [Maddock 2002].

Hash tables just barely missed the cutoff for C++98 and were one of the very first proposals for C++0x [Austern 2001]. They are called unordered (e.g., **unordered_map**) to distinguish them from the older, ordered, standard containers (e.g., **map**) and because the obvious names (e.g., **hash_map**) had been used extensively in other libraries before C++11. Also, **unordered_map** is arguably a better name because it refers to what the type offers, rather than to how it is implemented.

The **random** library provides distributions and random number generators of a sophistication that has earned in the description "what every random-number library wants to be when it grows up," but it is not easy to use for beginners or casual users (who often need random numbers). It was first proposed by Jens Maurer in 2002 [Maurer 2002] and finally accepted in 2006 after a revision by a group of people from the Fermi National Lab [Brown et al. 2006].

In contrast, Howard Hinnant's **chrono** library [Hinnant et al. 2008] for dealing with time points and durations is sophisticated, but also very easy to use. For example:

```
using namespace std::chrono;  // in sub-namespace std::chrono
auto t0 = system_clock::now();
do_work();
auto t1 = system_clock::now();
cout << duration_cast<milliseconds>(t1-t0).count() << "msec\n";
```

The **duration_cast** converts the clock-dependent "ticks" into the time unit of the programmer's choice.

Using such simple code, you can get even first-year students to appreciate the different costs of different algorithms and data structures. **chrono** provides the notion of time in the **thread** library (§4.1.2).

For C++20, **chrono** was enhanced with facilities for dealing with dates and time zones (§9.3.6). C++20 also allows us to simplify that example:

```
cout << t1-t0 << '\n';
```

This will output the time elapsed between **t0** and **t1** with an appropriate unit.

5 C++14: COMPLETING C++11

According to the plan of alternating major and minor releases, C++14 [du Toit 2014] was aimed at "completing C++11" (§3.2); that is, to include ideas accepted as good after the 2009 feature freeze and to remedy problems discovered during initial large-scale use of the C++11 standard. In this limited aim, it succeeded.

Importantly, it demonstrated that WG21 could deliver a standard on time. This, in turn, allowed implementers to deliver on time. Before the end of 2014, three major C++ implementers (Clang, GCC, and Microsoft) delivered what most people could consider feature complete. The conformance wasn't perfect, but people could experiment with essentially all features and combinations of features. The ability to compile libraries using "all advanced features" lagged a bit (until 2018 for Microsoft), but for most users the conformance was good enough for real use. The standards effort and the implementation efforts had become closely tied. That made a big difference to the community.

The C++14 feature set can be summarized as:

- Binary literals – e.g., **0b1001000011110011**
- §5.1: Digit separators – for readability, e.g., **0b1001'0000'1111'0011**
- §5.2: Variable templates – parameterized constants and variables
- §5.3: Function return type deduction
- §5.4: Generic lambdas
- §5.5: Local variables in **constexpr** functions
- Move capture – e.g., **[p = move(ptr)] { /* ... */ }**; move a value into a lambda
- Accessing a tuple by type, e.g., **x = get<int>(t);**
- User-defined literals in the standard library – e.g., **10i**, **"Hello!"s**, **10s**, **3ms**, **55us**, **17ns**

Most of these features were met with a combination of "Good, what took you so long?" and "Huh? who needs that?" My impression is that every addition was well motivated by some significant need – even if that need was not universal – and that the addition of local variables in **constexpr** functions and generic lambdas caused significant improvements to many people's code.

It was important that upgrading from C++11 to C++14 was relatively painless, with no ABI breakage. People who had done the major – and often difficult – upgrade from C++98 to C++11 were in for a pleasant surprise: they could upgrade sooner than expected and with less effort.

5.1 Digit Separators

Curiously enough, the digit separators led to the most heated debates. Lawrence Crowl repeatedly presented analyses of the alternatives [Crowl 2013]. Many, including me, argued for using the underscore as a separator (as in several other languages). For example:

```
auto a = 1_234_567;     // 1234567
```

Unfortunately, people were using the underscore as part of user-defined literal suffixes:

```
auto a = 1_234_567_s;  // 1234567 seconds
```

This could cause ambiguities. For example, is that last underscore a redundant separator or the start of a suffix? To my surprise this potential ambiguity made the underscore unacceptable to many. One reason was that to protect programmers from surprises, the library group had reserved suffixes *not* starting with an underscore for the standard library. After many long discussions, including debates in full committee (about 100 people), we agreed to use the single quote:

```
auto a = 1'234'567;     // 1234567 (an int)
auto b = 1'234'567s;    // 1234567 seconds
```

Despite dire warnings against this use of single quotes would break uncountable numbers of tools, this seems to work well. The single quote was suggested by David Vandevoorde [Crowl et al. 2013]. He pointed out that a single quote used as a separator in some countries, notably in Swiss financial notation.

My other suggestion, whitespace, never gained traction:

```
int a = 1 234 567;      // 1234567
int b = 1 234 567s;     // 1234567 seconds
```

Many people thought it was a joke relating to an old April fool's article [Stroustrup 1998]. In fact, it mirrors the old rule that adjacent string literals are concatenated, so that **"abc" "def"** means **"abcdef"**.

5.2 Variable Templates

In 2012, Gabriel Dos Reis proposed to extend the template mechanism to offer template variables in addition to template classes, functions, and aliases [Dos Reis 2012]. For example:

```
template<typename T>
constexpr T pi = T(3.1415926535897932385);

template<typename T>
T circular_area(T r)
{
    return pi<T> * r * r;
}
```

At first, this struck me as an obvious language-technical generalization of no particular importance. However, the history of problems with workarounds for specifying constants of various precisions is long and littered with uncomfortable workarounds and compromises. After this simple language generalization, significant amounts of code were simplified. In particular, variable templates emerged as the main way of defining concepts (§6.3.6). For example:

```
// expression:
template<typename T>
concept SignedIntegral = Signed<T> && Integral<T>;
```

The C++20 standard-library offers a set of mathematical constants defined as variable templates with the most common cases defined as a plain **constexpr** [Minkovsky and McFarlane 2019]. For example:

```
template<typename T> constexpr T pi_v = unspecified;
constexpr double pi = pi_v<double>;
```

5.3 Function Return Type Deduction

C++11 had introduced the ability to deduce the return type of a lambda from its return statement. C++14 extended that to functions

```
template<typename T>
auto size(const T& a) { return a.size(); }
```

This notational convenience can be significant for small functions in generic code. Users have to be careful, though, because such a function does not provide a stable interface because its type now depends on its implementation and that implementation must now be visible when a use of the function is compiled.

5.4 Generic Lambdas

Lambdas are function objects (§4.3.1) and as such they could obviously be templates. The problems related to generic (polymorphic) lambdas had been extensively discussed for C++11, but were deemed to be not quite ready then (§4.3.1).

In 2012, Faisal Vali, Herb Sutter, and Dave Abrahams proposed generic lambdas [Vali et al. 2012]. The proposed notation simply omitted the type from the syntax:

```
auto get_size = [](& m){ return m.size(); };
```

This was strongly opposed by many in the committee (including me) who pointed out that this syntax was unique and didn't generalize to constrained generic lambdas, so the notation was changed to use **auto** as a token indicating that a type was to be deduced:

```
auto get_size = [](auto& m){ return m.size(); };
```

This brought generic lambdas into line with concepts proposals and suggestions for generic functions stretching back as far as 2002 [Stroustrup 2003; Stroustrup and Dos Reis 2003a,b].

This direction of integrating the lambda syntax with the syntax used for the rest of the language was counter to the efforts of some who wanted a unique (ultra-terse) syntax for generic lambdas similar to what was found in other languages [Vali et al. 2012]:

```
C# 3.0 (2007):      x => x * x;
Java 1.8 (~2013):   x -> x * x;
D 2.0 (~2009):      (x) { return x * x; };
```

I think that the decision to use **auto** and in general not to introduce special notation for lambdas that is not shared with functions was correct. Further, I think that it was a mistake to introduce generic lambdas in C++14 without also introducing concepts so that the rules and notations for constrained and unconstrained lambda arguments and function arguments were not considered together. The language-technical irregularities stemming from this are (finally) remedied in C++20 (§6.4). However, we now have a generation of programmers accustomed to use unconstrained generic lambdas and proud of that. It will take significant time to overcome that.

From the brief discussion here, it might look as if notation/syntax are given an outsized importance in the committee process. That may be so, but syntax is not unimportant. The syntax is the programmer's user interface and debates about syntax often reflect differences over semantics or over the desired use of a feature. A notation should reflect the underlying semantics and a syntax typically favors one kind of use over another. For example, a fully general and verbose notation favors experts wanting to express subtle distinctions whereas a notation optimized for expressing simple cases favors novices and casual users. I am typically on the side of the latter and often in favor of supplying both (§4.2).

5.5 Local Variables in constexpr Functions

By 2012, people had stopped being scared of **constexpr** functions and started to demand relaxations of the restrictions on their implementations. There were people who essentially wanted to be able to do anything in **constexpr** functions. However, neither the users nor the compiler implementers were ready for that.

After some discussions, Richard Smith (Google) proposed a relatively modest set of relaxations [Smith 2013]. In particular, local variables and **for**-loops were allowed. For example:

```
constexpr int min(std::initializer_list<int> xs)
{
  int low = std::numeric_limits<int>::max();
  for (int x : xs)
    if (x < low)
      low = x;
  return low;
}

constexpr int m = min({1,3,2,4});
```

Given a constant expression as argument, this **min()** can be evaluated at compile time. The local variables (here, **low** and **x**) simply "live" in the compiler. The evaluation cannot have side effects on the environment of a caller. The original (academic) **constexpr** paper by Gabriel Dos Reis and Bjarne Stroustrup pointed to this possibility [Dos Reis and Stroustrup 2010].

This relaxation simplified many **constexpr** functions and pleased the many C++ programmers who had not been happy to find that only purely functional expressions of algorithms could be evaluated at compile time. In particular, they wanted loops to avoid recursion. In the longer term, this unleashed demands for further relaxations in C++17 and C++20 (§9.3.3). To illustrate the potential power of compile-time evaluation, I have pointed out that **constexpr thread**s are possible, though I am not in a hurry to propose that.

6 CONCEPTS

Generic programming and metaprogramming with templates have been runaway successes for C++. However, the proper specification of interfaces to generic components was slow reaching a satisfactory state. For example, in C++98, the standard-library algorithm was specified roughly like this:

```
template<typename Forward_iterator, typename Value>
ForwardIterator find(Forward_iterator first, Forward_iterator last,
                             const Value& val)
{
    while (first!=last && *first==val)
        ++first;
    return first
}
```

The standard says that

- the first template argument must be a forward iterator.
- the second template argument type must be comparable to the value type of that iterator using ==.
- the first two function arguments must denote a sequence.

These requirements are implicit in the code: all the compiler has to go by is the use of the template parameters in the function body. The result is great flexibility, splendid generated code for correct calls, and spectacularly bad error messages for incorrect calls. The obvious solution is to specify the first two requirements as part of the template's interface:

```
template<forward_iterator Iter, typename Value>
    requires equality_comparable<Value,Iter::value_type>
forward_iterator find(Iter first, Iter last, const Value& val);
```

This is roughly what C++20 offers. Note the **equality_comparable** concept. It captures the required relationship between the two template arguments. Such multi-argument concepts are very common.

To express the third requirement (that [first:last) is a sequence) requires a library extension. C++20 offers that in the Ranges standard-library component (§9.3.5):

```
template<range R, typename Value>
    requires equality_comparable<Value,Range::value_type>
forward_iterator find(R r, const Value& val)
{
    auto first = begin(r);
    auto last = end(r);
    while (first!=last && *first==val)
        ++first;
    return first
}
```

This section describes the attempts to provide good support for the specification of a template's requirements on its template arguments:

- §6.1: The prehistory of concepts
- §6.2: C++0x concepts
- §6.3: The concepts TS
- §6.4: C++20 concepts

6.1 The Prehistory of Concepts

In 1980, I conjectured that generic programming could be effectively supported through C-style macros [Stroustrup 1982]. I was flat wrong; a few useful simple generic abstractions could be expressed that way and the 1980s pre-standard C++ supported generic programming with a set of macros in **<generic.h>**, but macros weren't manageable in larger projects or in wide use. I did identify a problem that needed solving to meet my aims for "C with Classes," though, even though generic programming didn't have a place in "object-oriented thinking" as was then becoming fashionable.

In about 1987, I tried to design templates with proper interfaces [Stroustrup 1994]. I failed. I wanted three fundamental properties to support generic programming:

- *Full generality/expressiveness* – I explicitly didn't want facilities that could express only what I could imagine.
- *Zero overhead compared to hand coding* – for example, I wanted to build a vector that could compete with C arrays for time and space performance.
- *Well-specified interfaces* – I wanted facilities for type checking and overloading comparable to what we had for non-generic code.

Then, nobody could figure out how to get all three, so C++ got:

- Turing completeness [Veldhuizen 2003]
- Better than hand-coding performance
- Lousy interfaces (basically compile-time duck typing), but still statically type safe

The first two properties made templates a run-away success.

The lack of well-specified interfaces led to the spectacularly bad error messages we saw over the years and still in C++17. The lack of well-specified interfaces bothered me and many others over the years. It bothered me a lot because templates failed to meet the fundamental design criteria of C++ [Stroustrup 1994]. We (obviously) needed a simple way of specifying the requirements of a template on its template arguments that didn't imply run-time overheads.

For years, several people (including me) believed that requirements for template arguments could adequately be specified in the C++ itself. In 1994, I documented the basic idea in [Stroustrup 1994] and published examples on my website [Stroustrup 2004–2020]. Since 2006, Boost provided a variant of that idea, the Boost concept check library [Siek and Lumsdaine 2000–2007], based on the work of Jeremy Siek. Somehow, this never caught on as widely as I had hoped. I suspect the reason is that it wasn't sufficiently general, sufficiently elegant (Boost felt obliged to hide details in macros), and not supported in the standard. Many saw it as a hack.

Concepts, as defined for C++, go back to Alex Stepanov's work on generic programming starting in the late 1970s and first documented under the name "Algebraic structures" [Kapur et al. 1981]. Note that's almost a decade before the design of Haskell's type classes [Wadler and Blott 1989] and about 5 years before I tried to address the problem for C++. Alex Stepanov used the name "concept" for such requirements in lectures in the late 1990s and documented that in [Dehnert and Stepanov 2000]. I mention this because many have conjectured that concepts were derived from Haskell type classes and misnamed. Alex used the name "concept" because concepts were meant to represent fundamental concepts from an application domain, such as algebra.

The current use of concepts as type predicates relying of use patterns to describe operations has its origins in the work of Bjarne Stroustrup and Gabriel Dos Reis in the early 2000s and documented in [Dos Reis and Stroustrup 2005b, 2006; Stroustrup and Dos Reis 2003b, 2005a]. The approach is even mentioned in D&E from 1994 [Stroustrup 1994], but I don't remember when I first experimented with it. The reason for basing concepts on use patterns was primarily to handle implicit conversions and overloading in a simple and general manner. We knew of Haskell type classes, but they were not significant influences on the current C++ design because we considered them too inflexible.

Precise specification of a template's requirements on its arguments and the checking of those were to have been the centerpiece of C++0x and the crucial support for generic programming. It didn't even make C++17.

The 2003 papers by Bjarne Stroustrup and Gabriel Dos Reis [Stroustrup 2003; Stroustrup and Dos Reis 2003a,b] make it clear that concepts were part of an ambitious program to simplify generic programming. For example, a **concept** can be defined as a set of constraints specified as *use patterns*; that is, as language constructs that should be valid for a type [Stroustrup and Dos Reis 2003b]:

```
concept Value_type {
    constraints(Value_type a)
    {
        Value_type b = a;        // copy initialization
        a = b;                   // copy assignment
        Value_type v[] = {a};    // not a reference
    }
};

template<Value_type V>
    void swap(V& a, V& b);     // the arguments to swap() must be value types
```

The syntax and semantics were still very immature, though. We were primarily trying to establish design criteria [Stroustrup and Dos Reis 2003a]. From a modern (2018) perspective, [Stroustrup 2003; Stroustrup and Dos Reis 2003a,b] have many flaws. However, they offered design constraints for concepts together with suggestions for

- Concepts – compile-time predicates for specifying requirements on template arguments.
- Use patterns for specifying primitive constraints – to handle overloading and implicit type conversions.
- Multi-argument concepts – e.g., **Mergeable<In1,In2,Out>**.
- Both type and value concepts – that is, concepts can take values as well as types as arguments, e.g., **Buffer<unsigned char,128>**.
- A shorthand notation for "type of type" uses of templates – e.g., **template<Iterator Iter> ...**.
- A "simplified notation for template definitions" – e.g., **void f(Comparable&);** to bring generic programming closer to "ordinary programming."
- **auto** as the least constrained type in function arguments and return values.
- Uniform function call (§8.8.3) – to alleviate problems with style differences between generic programming and object-oriented programming (e.g., **x.f(y)**, **f(x,y)**, and **x+y**).

Curiously enough, we didn't suggest general **requires**-clauses (§6.2.2). Those have been part of all later variations of concepts.

6.2 C++0x Concepts

In 2006, essentially everyone expected that the version of concepts described in [Gregor et al. 2006; Stroustrup 2007] and voted into the draft standard (working paper) would be part of C++09. However, C++0x became C++11 and in 2009 the committee agreed by a massive voting majority to abandon the concept design [Becker 2009] as mired in complexity and usability problems [Stroustrup 2009a,b]. The reasons for this failure are varied and may have lessons beyond the C++ standards effort.

In 2004 there were two independent efforts to add concepts to C++. There were often referred to as "Indiana" and "Texas," respectively because the major proponents came from Indiana University and Texas A&M University:

- *Indiana*: An approach related to Haskell type classes and primarily relying on tables of operations to define concepts. It was deemed important for a programmer to explicitly state that a type "modeled" a concept; that is, that the type offered a set of operations specified by the concept [Gregor et al. 2006]. Key people were Andrew Lumsdaine (professor) and Douglas Gregor (postdoc and compiler writer).
- *Texas*: An approach based on compile-time type predicates and predicate logic. It was deemed important for usability that a programmer should *not* have to explicitly specify which types matched which concepts (those matches could be computed by the compiler). It was considered essential for C++ to elegantly and efficiently handle implicit conversions, overloading, and mixed-type expressions [Dos Reis and Stroustrup 2006; Stroustrup and Dos Reis 2003b]. Key people were Bjarne Stroustrup (professor) and Gabriel Dos Reis (postdoc, later professor).

Given these descriptions, it may seem obvious that the approaches were irreconcilable, but that was not obvious to the people involved at the time. In fact, I argued that the approaches were theoretically equivalent [Stroustrup and Dos Reis 2003b]. That argument may indeed be true, but the practical implications for detailed language design and use in the context of C++ were not equivalent. Separately, the WG21 consensus process as interpreted by the committee members strongly encourages collaboration and joint proposals as opposed to working for years on competing proposals and ending up with a grand shoot-out between them (§3.2). I consider

the latter approach a recipe for dialect creation because a losing side is unlikely to just go away abandoning their implementation and users. Note that all the people mentioned above together with Jeremy Siek (grad student in Indiana and my summer intern in AT&T Labs) and Jaakko Järvi (postdoc in Indiana and later professor in Texas A&M) were co-authors of the OOPSLA paper presenting the first version of the compromise design. The Indiana and Texas groups were never completely disjoint and we tried hard for genuine consensus. In addition, I had known Andrew Lumsdaine for years before this work. We really wanted the compromise design to work.

The Indiana design was far ahead of the Texas design in terms of implementation and had more people involved, so we proceeded primarily based on that. The Indiana design was also more conventional, based on function signatures and had obvious similarities to Haskell type classes. Given the number of academics involved, it was important that the Indiana design was seen as more conventional and academically respectable. It seemed that we "just" had to

- make the compiler acceptably fast
- generate efficient code
- handle overloading and implicit conversions.

That decision cost us three years of hard work and much controversy.

The C++0x concept design is described in [Gregor et al. 2006; Stroustrup 2007]. The former paper contains a standard academic "related work" section comparing the design to facilities offered by Java, C#, Scala, Cecil, ML, Haskell, and G. Here I summarize, using examples from [Gregor et al. 2006].

6.2.1 Concept Definitions. A concept was defined as a set of operations and associated types:

```
concept EqualityComparable<typename T> {
    bool operator==(const T& x, const T& y);
    bool operator!=(const T& x, const T& y) { return !(x==y); }
}

concept InputIterator<typename Iter> {
    // Iter must have a member value_type:
    typename value_type = Iter::value_type;
    // ...
}
```

The similarity between a concept and a class was by some (Indiana) considered an advantage.

Functions specified in concepts were not exactly like functions defined in classes, though. For example, an operator declared within a **concept** doesn't have the implicit argument ("**this**") that an operator defined within a **class** has.

There was a serious problem lurking in the approach of defining a concept as a set of operations. Consider the ways an argument can be passed in C++:

```
void f(X);
void f(X&);
void f(const X&);
void f(X&&);
```

Ignore for a moment **volatile** because that's rarely seen for arguments in generic code, but we still have four alternatives. In a **concept**, do we

- represent **f** as one function and have the users pick the right kind of argument for their calls?
- overload **f** with all alternatives?

- represent **f** as one function and let the users define a **concept_map** (§6.2.3) to map to **f**'s desired argument type?
- have the language implicitly map user's argument types to the template's parameter type?

For two arguments, we would have 16 alternatives. Three argument generic functions are rare, but we would have 4*4*4 alternatives. What about variadic templates? (§4.3.2); for those, we could have 4^N alternatives.

The semantics of the different ways of passing an argument are not equivalent, so we naturally drifted towards accepting the argument type as specified, pushing the burden of matching onto designers of types and writers of **concept_map**s (§6.2.3).

Similarly, we have the problem whether to specify a **concept** for **x.f(y)** (object-oriented style) or **f(x,y)** (functional style), or both. This problem occurs immediately when we try to specify a binary operator, such as **+**.

In retrospect, we were far too optimistic about solving these problems within the framework of concepts defined in terms of operations with specific types or specific "pseudo signatures," where a "pseudo signature" somehow represented a solution to the problems outlined here.

Relationships among concepts were defined by explicit *refinement*:

```
concept BidirectionalIterator<typename Iter>   // A BidirectionalIterator is
    : ForwardIterator<Iter> {                  // a kind of ForwardIterator
        // ...
}
```

Refinement was almost, but not quite, like class derivation. The idea was for programmers to explicitly build up hierarchies of concepts. Unfortunately, that introduces a serious inflexibility into the system. Concepts (in the conventional English meaning) are often not strictly hierarchical.

6.2.2 Concept Use. A concept could be used either as a predicate in a **where**-clause or in a shorthand notation:

```
template<typename T>
    where LessThanComparable<T>       // explicit predicate
const T& min(const T& x, const T& y)
{
    return x<y ? x : y;
}

template<GreaterThanComparable T>   // shorthand notation
const T& max(const T& x, const T& y)
{
    return x>y ? x : y;
}
```

For simple "type of type" concepts, the shorthand notation (first suggested in [Stroustrup 2003]) quickly became very popular. However, we soon found that **where** was far too popular as an identifier in existing code and renamed it to **requires**.

6.2.3 Concept Maps. Relationships between concepts and types were defined by specializations of **concept_map**s:

```
concept_map EqualityComparable<int> {};    // int is EqualityComparable

// student_record is EqualityComparable:
```

```
concept_map EqualityComparable<student_record> {
    bool operator==(const student_record& a, const student_record& b)
    {
        return a.id_equal(b);
    }
};
```

For **int**, we can simply say that the type **int** has the properties required by **EqualityComparable** (that is, it has == and **!**=). However, **student_record** doesn't have a ==, but we can add one in the **concept_map**. Thus, a **concept_map** is a very powerful mechanism for non-intrusively adding properties to a type in specific circumstances.

Why do we have to tell the compiler that **int**s can be compared? The compiler already knows.

This was a constant point of contention. The "Indiana group" generally felt that being explicit was good (always) and the "Texas group" tended to consider a concept map worse than useless unless it added functionality. Would explicit statements save use from significant errors from "accidental" syntactic matches that were semantically meaningless? Alternatively, would such errors be rare and the explicit modeling statements be mostly an annoyance to write and opportunities for making mistakes? The compromise solution was to allow the definer of a **concept** to declare the use of a **concept_map** optional by adding **auto**:

```
auto concept EqualityComparable<typename T> {
    bool operator==(const T& x, const T& y);
    bool operator!=(const T& x, const T& y) { return !(x==y); }
}
```

Now, the **concept_map** for **EqualityComparable** is automatically used when **EqualityComparable** required for a type even when a user didn't supply a specialization for that type.

6.2.4 Definition Checking. The code in a template definition was checked against the concepts of its template parameters:

```
template<InputIterator Iter, typename Val>
    requires EqualityComparable<Iter::value_type,Val>
Iter find(Iter first, Iter last, Val v)
{
    while (first<last && !(*first==v))   // error: no < in EqualityComparable
        ++first;
    return first;
}
```

Here, we use < to compare iterators, but **EqualityComparable** guarantees only == so this definition won't compile. Catching such used of non-guaranteed operations was seen as an important benefit, but is turned out to have serious negative implications: (§6.2.5) and (§6.3.1).

6.2.5 Lessons Learned. The years after the initial proposal and relatively fast approval was filled with plugging holes in this initial design and addressing comments about generality, implementability, quality of specification, and usability.

Doug Gregor, as the primary implementer, performed heroics to generate quality code, but at the end, the concepts compiler was still more than 10 times slower than a compiler implementing just unconstrained templates. I suspect that the implementation problems had their ultimate roots in the adoption of a class-like structure for representing concepts in the compiler. This enabled quick early results, but left the concepts represented in a way carefully crafted for classes, and concepts are not classes. Having a concept represented as a set of functions (like virtual member

functions), led to problems handling implicit conversions and mixed-type operations. The very flexible combinations of code from different contexts that is the "secret" of some of the powerful generic programming and metaprogramming code-generation techniques became impracticable to specify with C++0x concepts. It is essential for matching (unconstrained) template performance that the functions used to specify a concept do not appear in the generated code as function calls (or worse still, as indirect function calls).

I was unpleasantly reminded of the problems that many early C++ compiler writers had had from adopting the structure and code base of a C compiler. The handling of C++ scopes and overloading didn't fit well into a C compiler framework. Based on the idea that design concepts should be represented directly in code, Cfront (§2.1) used specific scope classes to avoid such problems. However, most compiler writers with a C background thought they could take a shortcut using familiar C techniques, but ended up having to rewrite their C++ front-end from scratch anyway. Language design and implementation techniques can strongly influence each other.

It soon became obvious that we needed language support for the transition from unconstrained templates to templates using concepts. In the C++0x design, they were very different:

- A constrained template cannot call an unconstrained one because it is not known what operations an unconstrained template uses, so the definition check of the constrained template cannot be done.
- An unconstrained template can call a constrained one, but checking must be postponed until instantiation time because not until then do we know what types the unconstrained template uses in its calls.

The solution to the first problem was to allow the programmer to say "don't check these calls from a constrained template" using a **late_check** block [Gregor et al. 2008]:

```
template<Semigroup T>
T add(T x, T y) {
    T r = x + y;      // uses Semigroup<T>::operator+
    late_check {
        r = x + y;  // uses operator+ found at instantiation time
                    // (not considering Semigroup<T>::operator+)
    }
    return r;
}
```

This "solution" was at best a patch and had the extraordinary problem that a **concept_map** for **Semigroup** wouldn't be known in the unconstrained called template. This had the "interesting effect" that an object could be used in exactly the same way in two places in a program with different semantics. Thus, the type system had been violated in a really hard-to-trace way.

As we used concepts more, the role of semantics in the design of concepts (and indeed of types and libraries) became increasingly clear and many in the committee started pushing for a mechanism for expressing semantics rules. This was not a surprise. Alex Stepanov was fond of saying "concepts are all about semantics." However, most people were approaching concepts like any other language facility and were more concerned with syntax and name lookup rules.

In 2009, a notation, called **axiom**s was proposed by Gabriel Dos Reis (strongly supported by me) and approved [Dos Reis et al. 2009]:

```
concept TotalOrdering<typename Op, typename T> {
    bool operator()(Op, T, T);
    axiom Antisymmetry(Op op, T x, T y) {
        if (op(x, y) && op(y, x))
```

```
            x <=> y;
        }
        axiom Transitivity(Op op, T x, T y, T z) {
            if (op(x, y) && op(y, z))
                op(x, z);
        }
        axiom Totality(Op op, T x, T y) {
            op(x, y) || op(y, x);
        }
    }
```

Curiously enough, it was hard to get the notion of an axiom accepted. The main objection seemed to be that the proposers explicitly rejected the idea of compilers testing axioms against the types for which they were used "to catch errors." Apparently, the notion that **axiom**s were axioms in the mathematical sense (that is, assumptions that you are allowed to make because you in general can't check them) was alien to some committee members. Others weren't convinced that specifying axioms could help tools other than the compilers. However, **axiom**s were adopted as part of **concept** specifications.

There were obvious problems with our definition and implementation of concepts, but we had a pretty complete facility and plowed along trying to solve the problems and to gain experience by using concepts in the definition of the standard library [Gregor and Lumsdaine 2008] and other libraries.

6.2.6 What Went Wrong? In 2009, I had reluctantly come to the conclusion that the concepts effort was in deep trouble. The problems that I had expected us to solve were still festering and new ones had been discovered:

- We still didn't have an agreement whether implicit or explicit modeling (implicit or explicit use of **concept_map**s) was the right approach in most cases.
- We still didn't have an agreement whether to rely on implicit or explicit statement of relations among concepts (should we explicitly build hierarchies of "refinement" relations in a way very similar to object-oriented inheritance?).
- We were still seeing examples where code generated from concept-constrained code was inferior to code generated from unconstrained templates. The late composition opportunities from templates kept showing surprising strengths.
- It was still difficult to write concepts to capture all the conversions and overload cases that we were used to from generic and non-generic C++.
- We were seeing increasing numbers of examples where the combination of non-trivial **concept_map**s and **late_check**s led to inconsistent views of types (aka surprising and almost invisible violations of the type system).
- The complexity of specification in the draft standard had ballooned beyond all expectation (91 pages, excluding any uses of concepts in the library) and some of us considered it essentially unreadable.
- The set of concepts used to specify the standard library was getting large (about 125 concepts, 103 for the STL alone).
- The compiler was getting better at code generation (because of heroic efforts from Doug Gregor) but not faster. Some major compiler vendors were telling me in private and confidence that if a concept-enabled compiler was more than 20% slower than older compilers they'd have to oppose concepts however nice they were. At the time, the concept-enabled compiler was more than 10 times slower.

In the spring of 2009, a wide-ranging discussion occurred on the standard reflectors (archived email groups). It started when Howard Hinnant asked an eminently practical question about the use of concepts: A utility that he was designing could be done in two ways: One would require quite a lot of users – not necessarily expert users – to write concept maps. The other – far less elegant – design would avoid concept maps (and concepts) so as not to require users to understand anything significant about concepts. Should "average users" understand concepts? Just enough to use them? Enough to define them?

This became known as the "Are concepts required of Joe Coder?" thread. Who is "Joe Coder?" asked Peter Gottschling. Great question, I answered:

> *"I think most C++ programmers are "Joe Coder" (I again register my opposition to that term). I'm Joe Coder most of the time and with most libraries. I expect to remain so as long as I keep learning new techniques and libraries. Yet, I want to use concepts (and, when I must, concept maps). I want the "doctrine of use" radically simpler than the subtle expert-only use of facilities we have now."*

In other words, should we design concepts as a delicate instrument for fine control by a small group of language experts or as a robust tool for most programmers? This is a question that comes up again and again in the design of language features and standard-library components. I heard it for years about classes; for some, defining a class was obviously something that most programmers should be discouraged from doing. The "average programmer" (sometimes derisively referred to as "Joe Coder") was to some obviously not smart enough or educated enough to use sophisticated features and techniques. I was (and am) strongly of the opinion that most programmers can learn to use features like classes and concepts well. Once they do, their programming become easier and their code better. It may take years for the community at large to absorb a lesson, but if that can't be done, we – as language and library designers – have failed.

In response to that thread and reflecting my growing concern about the direction of the work on C++0x concepts, I wrote a paper *Simplifying the use of concepts* [Stroustrup 2009c] outlining what I saw as the minimum improvements necessary for concepts to be acceptable in C++0x:

- Make **concept_map**s rare.
- Make all **concept_map**s implicit/automatic.
- Make a concept that requires **begin(x)** accept **x.begin()** and vice versa (uniform function call; (§6.1), (§8.8.3)
- Make all standard-library concepts implicit/automatic.

The paper is quite detailed with many examples and suggestions that had emerged over the years.

One reason that I insisted on making *all* concepts implicit/automatic was the observation that given a choice, the least flexible and least trusting programmers could force everyone to live with their choice of explicit concepts. Library writers were showing a strong tendency to push resolution of even the most obvious choices onto their users by using explicit (non-automatic) concepts.

I observed that the then recent *Elements of Programming Style* [Stepanov and McJones 2009] by Alex Stepanov, the father of C++ generic programming, didn't use a single concept map to describe a superset of the facilities from the STL and a superset of the then common generic programming techniques.

The committee responded with a discussion almost exclusively focused on whether a consensus was likely for a timely standard and came to the obvious conclusions that it was not likely. We could not agree to "fix" concepts to make them usable by most programmers and also ship the standard (more or less) on time. Thus, "concepts" – the result of years of work by many competent

people – was removed from the draft standard. My summary of the "remove concepts" decision [Stroustrup 2009a,b] is more readable than the technical papers and discussions.

As the committee voted to remove concepts by a large majority (I too voted for removal), everyone who spoke up reconfirmed that they wanted concepts. The vote just reflected that the concept design wasn't ready for standardization. I think that the problem was far worse: The committee wanted concepts, but the members didn't agree about what kind of concepts they wanted. The committee did not have a shared set of design aims. This is still a problem, and not just for concepts. There are deep "philosophical" differences among members. In particular:

- *Explicit vs. implicit*: Should programmers – in the interest of safety and the avoidance of surprises – be explicit about how choices among potential alternatives are resolved? This discussion turns up in discussion about overload resolution, scope resolution, matching of types to concepts, relationships among concepts, and more.
- *Experts vs. average*: Should key language and standard-library facilities be designed for the use of experts? If so, should "average programmers" be encouraged to exclusively use a limited subset of the language and should separate libraries be designed for "average programmers"? This discussion turns up in the contexts of the design and use of classes, class hierarchies, exceptions, templates, and more.

In both cases, a "yes" will bias the design of facilities towards complicated features that require great expertise and frequent use of notation to get right. I tend systematically to be on the other side of such arguments, trusting "average programmers" more and relying on regular language rules and checking by compilers and other tools to avoid nasty surprises. Programmers are at least as likely to get explicit resolution of tricky issues wrong as are (implicit) language rules.

Different people have drawn different conclusions from the C++0x concepts failure. I drew three main ones:

- We put too much weight on early implementation. We should have spent more effort on working out the requirements, the constraints, the desired use patterns, and a relatively simple implementation model. After that, we could have grown an implementation relying on feedback from use.
- Some disagreements are fundamental (philosophical) and cannot be resolved through compromises. We must identify and articulate such issues early on.
- No set of facilities can serve all the diverse wishes from a large committee of experts without becoming so large that the bloat becomes a problem for implementers and an impediment to users. We must identify core needs and serve them well with a simple notation; more complex uses and rarer use cases can be served with facilities and notation that puts greater requirements on the expertise of their users (§4.2).

These conclusions do not have anything in particular to do with concepts. They are general observations about design aims and decision making within a large group.

6.3 The Concepts TS

In 2009, almost immediately after the removal of concepts from C++0x, Gabriel Dos Reis, Andrew Sutton, and I started to re-design concepts based on our original ideas, the experience gained from the C++0x language design, the experience from use of C++0x concepts, and the feedback from the standards committee. We concluded

- concepts must have semantic meaning
- there should be relatively few concepts
- concepts should be fundamental, rather than minimal

We deemed most of the concepts included in the C++ standard library meaningless in isolation [Sutton and Stroustrup 2011]. "The STL does not have 103 concepts for any reasonable definition of 'concepts'!" I cried during a discussion with Andrew Sutton, "even basic algebra doesn't have more than about a dozen!" Language design discussions can become quite animated.

In 2011, at the urging of Andrew Lumsdaine, Alex Stepanov called a week-long meeting in Palo Alto. A largish group, including most of the people closely associated with the C++0x concepts effort, Sean Parent, and Alex Stepanov, met to attack the problem from the user's perspective: What would a properly constrained set of STL algorithms ideally look like? Then, we went home to document our user-oriented design and invent language mechanisms to approximate that ideal design [Stroustrup and Sutton 2012]. This effort re-booted the standards effort based on a new, fundamentally different and better, approach than the C++0x effort. The 2016 ISO TS (Technical Specification) for concepts [Sutton 2017] and the C++20 concepts (§6.4) are direct results of that meeting. Andrew Sutton's implementation has been used for experimentation since 2012 and shipping as part of GCC 6.0 and higher.

In the concepts TS [Sutton 2017]

- Concepts are based on compile-time predicates (including multi-argument predicates and value arguments).
- Primitive requirements specified in terms of use patterns [Dos Reis and Stroustrup 2006] (**requires**-expressions).
- A concept can be used in general **requires**-clauses, as an alternative to a **typename** in template parameter definitions, and as an alternative to a type name in function parameter definitions.
- Matching of types to concepts is implicit (no **concept_map**s).
- Relations among concepts for overloading are implicit (computed; there are no explicit refinement of concepts).
- There is no definition checking (for now at least, so no **late_check**).
- There are no **axiom**s, but that is just because we didn't want to complicate and delay the design with a potentially controversial feature. The C++0x **axiom**s would make a good start.

Compared to the C++0x concepts there is a strong emphasis on simplifying the use of concepts. A major part of that has been to eliminate requirements for programmers to be explicit and rely on the compiler resolving issues based on well-specified and simple algorithms.

People who favor explicit resolution by users tend to deem that an emphasis of semantics over syntax and warn against "accidental matches" and "surprises." The most common example is that a **Forward_iterator** differs from an **Input_iterator** only in its semantics: a **Forward_iterator** allows multiple passes over its sequence. Nobody disagrees that such examples exist, but endless debates have raged (and still rage) over how important such examples are and how to resolve them. I consider it a bad mistake to let a few rare complicated examples dominate a design.

The Concept TS design is based on the opinion (backed by much experience) that such examples are very rare (especially in well-designed concepts [Stroustrup 2017a]), are usually well understood by concept writers, and can often be resolved by adding an operation to the most constrained concept to reflect the sematic difference. For example, a simple solution to the **Forward_iterator**/**Input_iterator** problem is to require a **Forward_iterator** to offer a **can_multipass()** operation. It doesn't even have to do anything; it would just be there so that the concept resolution mechanism can check for its existence. Thus, no new language features need to be added specifically to resolve possible accidental ambiguities.

Because the point is often missed, I must emphasize that concepts are predicates. They are not classes or class hierarchies. Fundamentally, we are just asking a type simple questions, such as "are

you an Iterator?", and asking sets of types questions about their interoperation, such as "can you compare against each other using ==?" (§6.3.2). Using concepts, we only ask questions that can be answered at compile time; no run-time evaluation is involved. Potential ambiguities are detected by comparing the predicates involved for a type (or set of types); not by having the programmer write resolution rules (§6.3.2).

Sensitive to the problems with C++0x concepts (§6.2.6), we were careful to design concepts so that their use would not imply significant compile-time overheads. Even early versions of Andrew Sutton's compiler was able to compile templates using concepts *faster* than programs using workarounds (e.g., **enable_if** (§4.5.1)).

6.3.1 Definition Checking. Sometime during the months after that Palo Alto meeting, Andrew Sutton, Gabriel Dos Reis and I decided to attack the design and implementation of the concept language features in stages. That way, we could learn from the implementation experience and gain early feedback from use before "freezing the design". In particular, we decided to postpone the implementation of *definition checking* (§6.2.4); that is, the checking that a template doesn't use facilities not specified for its arguments. Consider a simplified version of **std::advance()** that moves an iterator **n** positions forward in a sequence:

```
template<Forward_iterator Iter>
void advance(Iter p, int n)
{
    p+=n; // advance p n positions
}
```

A **Forward_iterator** doesn't offer +=, only ++, so definition checking will catch this as an error. Without checking the body of **advance()** in isolation (before use) we will only get poor instantiation-time error message from the (mis)use of +=. Note that the code generated from template instantiation is always type checked, so not doing definition checking will not lead to run-time errors.

We decided that on the order of 90% of the benefits from concepts would accrue from point-of-use checking and that the relatively expert implementers of constrained templates could do without definition checking for a while. This 90% was obviously an ad hoc estimate based on limited information, but with the benefit of a decade's work with concepts I consider it a good guess. It was more important to us as designers of language features and libraries to gain experience with use, starting with the STL algorithm examples from the Palo Alto Technical Memorandum [Stroustrup and Sutton 2012]. We valued feedback over theoretical completeness. This was radical. Looking back over the documents about concepts (in C++ and for other languages), definition checking was always emphasized as a major reason for offering concepts as a language feature [Gregor et al. 2006; Stroustrup and Dos Reis 2003b].

For a while, this new design was referred to as *Concepts Lite* and many deemed it incomplete, or even useless. However, we quickly discovered that *not* doing definition checking offered real benefits [Sutton and Stroustrup 2011].

- With definition checking we can't use partial concept checking during development. It is very common not to know the full set of requirements during the initial construction of a larger program. Partial checking allows many errors to be caught early and encourages gradual improvement of a design based on feedback from early use.
- Definition checking makes it hard to have stable interfaces. In particular, it is not possible to add debug statements, statistics gathering, tracing, or "telemetry" to a class or a function

without changing its interface to include the facilities for that. Such facilities are rarely fundamental to a class or a function and tend to change over time.
- When we don't use definition checking, existing templates can be gradually converted to use concepts. However, if we have definition checking, a constrained template cannot use an unconstrained one because we cannot in general know which facilities the unconstrained template uses. Also, an unconstrained template using a constrained one implies late (instantiation-time) checking in either case.

Ville Voutilainen, the chair of the EWG from 2014 onwards, said it stronger:

I cannot support any proposal for concepts that includes definition checking.

We might eventually get a form of definition checking, but only if we can design a mechanism for escaping it to address transition and data gathering needs. That will have to be carefully thought out and experimentation will be needed. The C++0x **late_check** is not sufficient.

The problems with definition checking are problems of use, not implementation. Gabriel Dos Reis designed and implemented an experimental language, called Liz, testing out the facilities of the Concepts TS design [Dos Reis 2012], including definition checking. If we find an acceptable form of definition checking, we can implement it.

6.3.2 Concept Use. Simple examples look much like they did for C++0x and before:

```
template<Sequence Seq, Number Num>
Num sum(Seq s, Num v)
{
    for (const auto& x : s)
        v+=x;
    return v;
}
```

Here **Sequence** and **Number** are concepts. Using a concept instead of **typename** to introduce the name of a type means that the type used must satisfy the requirements of the concept. Note that because the concepts TS doesn't offer definition checking, the use of += will not be checked by the concepts, but only late, at instantiation time. This is likely during initial development. Later we would most likely be more explicit:

```
template<typename T>
using Value_type = typename T::value_type;  // simplified alias

template<Sequence Seq, typename Num>
    requires Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n)
{
    for (const auto& x : s)
       v+=x;
    return v;
}
```

That is, we must have arithmetic operations, including +=, for combinations of the **Sequence**'s value type and the type we use as the accumulator. We no longer have to specify **Num** to be a **Number**; **Arithmetic** checks that **Num** has the needed properties. Here, **Arithmetic** is used explicitly as a predicate in a (C++0x-style) **requires**-clause.

Overloading is handled by picking the function that has the strictest requirements. Consider a simple version of the classical **advance** example from the standard library:

```
template<Forward_iterator Iter>
void advance(Iter p, int n)   // move p n elements forward
{
   while (--n)
        ++p;    // a forward iterator has ++, but not + or +=
}

template<Random_access_iterator Iter>
void advance(Iter p, int n)   // move p n elements forward
{
     p+=n;    // a random-access iterator has +=
}
```

That is, we should use the second version for sequences that offer random access and the first for sequences that offer only forward iteration:

```
void user(vector<int>::iterator vip, list<string>::iterator lsp)
{
    advance(vip,10);  // use the fast advance()
    advance(lsp,10);  // use the slow advance()
}
```

The compiler breaks the concepts for the two functions into primitive ("atomic") requirements and since the requirements for forward iteration are a strict subset of those for random access iteration, this example can be resolved.

Overlapping requirements that are not strict subsets of each other give ambiguities (compile-time errors) when an argument type match both. For example:

```
template<typename T>
    requires Copyable<T> && Integral<T>
T fct(T x);

template<typename T>
    requires Copyable<T> && Swappable<T>
T fct(T x);

int x = fct(2);    // ambiguous: int is Copyable, Integral, and Swappable
auto y = fct(complex<double>{1,2});  // OK: complex is not integral
```

The only control mechanism offered to programmers is the ability to add operations when defining a concept. That seems to be sufficient for real-world examples, though. Of course, you can define concepts that differ only semantically so that there would be no way of distinguishing them based on our syntax-only concepts. However, it's not difficult to avoid doing that.

6.3.3 Concept Definition. Primitive requirements are specified by use patterns in **requires**-expressions:

```
template<typename T, typename U =T>
concept Equality_comparable =
    requires (T a, U b) {
        { a == b } -> bool; // compare Ts and Us with == yielding a bool
        { a != b } -> bool; // compare Ts and Us with != yielding a bool
    };
```

The **requires**-expressions were invented by Andrew Sutton as part of his implementation of the Concepts TS. They turned out to be so useful that the users insisted they become part of the standard.

The =**T** gives a default for the second type argument so that **Equality_comparable** can be used for a single type.

The notation for the use patterns were invented by Bjarne Stroustrup in real time at the Palo Alto meeting based on ideas from 2003 [Stroustrup and Dos Reis 2003b]. This notation and the ideas that it is based upon do not involve function signatures or function-table implementation.

There is no specific mechanism for saying that a type matches a concept, but if someone wants to, the general C++11 **static_assert** can be used:

```
static_assert(Equality_comparable<int>);       // succeeds
static_assert<Equality_comparable<int,long>);  // succeeds

struct S { int a; };
static_assert(Equality_comparable<S>);     // fails because structs don't
                                           // automatically get == and !=
```

The associated type notion from C++0x (and earlier [Stroustrup 2003]) is also supported:

```
template<typename S>
concept Sequence = requires(S a) {
  typename Value_type<S>;              // S must have a value type.
  typename Iterator_type<S>;           // S must have an iterator type.

  { begin(a) } -> Iterator_type<S>;    // begin(a) must return an iterator
  { end(a) } -> Iterator_type<S>;      // end(a) must return an iterator
  { a.begin() } -> Iterator_type<S>;   // a.begin() must return an iterator
  { a.end() } -> Iterator_type<S>;     // a.end() must return an iterator

  requires Same_type<Value_type<S>,Value_type<Iterator_type<S>>>;
  requires Input_iterator<Iterator_type<S>>;
};
```

Note the repetition to be able to accept both **a.begin()** and **begin(a)**. The lack of uniform function call hurts (§6.1), (§8.8.3).

6.3.4 Concept Name Introducers. One thing we learned from use was that primitive uses of concepts were repetitive. We used too many **requires**-expressions directly in **requires**-clauses and we used too many "small" concepts. Our requirements looked a lot like code written by novice programmers: too few functions, too little abstraction, and too few symbolic names.

Consider the standard **merge** family of functions. These functions all take three sequences and need to specify the relationships among those sequences. That's three requirements on the type of sequences plus three requirements on the relationships among the elements of those sequences. First try:

```
template<Input_iterator In1, Input_iterator In2, Output_iterator Out>
    requires Comparable<Value_type<In1>,Value_type<In2>>
    && Assignable<Value_type<In1>, Value_type<Out>
    && Assignable<Value_type<In2>, Value_type<Out>
Out merge(In1, In1, In2, In2, Out);
```

That's tedious and such patterns of type name introduction are very common; for example, there are at least four **merge** functions in the STL. Tedious and repetitive code is error-prone and hard to maintain. We soon learned to use more multi-argument concepts to define common patterns of requirements among types:

```
template<Input_iterator In1, Input_iterator In2, Output_iterator Out>
    requires Mergeable<In1,In2,Out>
Out merge(In1, In1, In2, In2, Out);
```

This then became too messy for Andrew Sutton, who in 2012 probably had written more code using concepts than anyone else. He suggested a mechanism for saying "introduce a set of type names for types that must satisfy a concept." That reduced the **merge** example to its logical minimum:

```
Mergeable{In1,In2,Out} // concept name introducer
Out merge(In1, In1, In2, In2, Out);
```

It is amazing what you can learn just by trying! It is also amazing how much resistance novel notations and solutions can encounter from people who have yet to experience the problem.

6.3.5 Concepts and Types. Many still saw (and see) concepts as a variant of the type-of-type idea. Yes, a concept taking a single type argument can be seen as a type of a type, but only the simplest uses fit that pattern.

Most generic functions (algorithms) take more than one template argument and for such a function to make any sense, those argument types must somehow relate to each other. Then, we must use multi-argument concepts. For example:

```
template<Forward_iterator Iter, typename Val)
    requires Equality_comparable<Value_type<Iter>,Val>
Forward_iterator find(Iter first, Iter last, Val v)
{
    while (first!=last && *first!=v)
        ++first;
    return first;
}
```

Crucially, the multi-argument concepts directly address the need to handle implicit conversions and mixed-type operations. Together with Gabriel Dos Reis, I had considered the possibility of specifying all constraints on each argument in isolation from other arguments as early as 2003 [Stroustrup 2003; Stroustrup and Dos Reis 2003b]. This would involve

- parameterization (e.g., an **Iterator** parameterized with its value type)
- some form of inheritance (e.g., a **Random_access_iterator** is a **Forward_iterator**)
- the ability to apply more than one concept to a template argument type (e.g., the element of a **Container** must be **Value_type** and a **Comparable**)
- combinations of those three techniques.

The result was very complex template argument type constraints. We deemed that complexity unnecessarily and unmanageable. Consider **x+y** and **y+x** where **x** and **y** are of different template argument types, **X** and **Y**. Dealing with each template argument on its own, we would have to parameterize **X** with **Y** and **Y** with **X**. In a pure object-oriented language, that might seem natural; after all, there would be two methods to cope with **+**, one in **X**'s hierarchy and one in **Y**'s hierarchy. However, I rejected that solution for C++ back in 1982. To complete the picture, we must add

implicit conversions (e.g., to handle **x+2** and **2+x**) . The multi-argument concepts exactly match the way C++ resolves such examples and avoid most of the complexity.

This decision was revisited repeatedly over the years and confirmed. In the context of the C++0x concepts effort, people tried applying the standard academic systems as found in Haskell typeclasses and Java constraints. However, that failed to deliver the simplicity of implementation and use needed for large scale use.

Where a generic use fits the type-of-type pattern, concepts supports it quite elegantly.

- *A type* specifies the set of operations that can be applied to an object (implicitly and explicitly), relies on function declarations and language rules, and specifies how an object is laid out in memory.
- *A concept* specifies the set of operations that can be applied to an object (implicitly and explicitly), relies on use patterns reflecting function declarations and language rules, and says nothing about the layout of the object. Thus, a concept is a kind of interface.

My ideal is to be able use concepts wherever we use a type and in the same way. Except for defining layout, they are very similar. Concepts can even be used to constrain the type of variables with their type determined by their initializer (constrained **auto** variables (§4.2.1)). For example:

```
template<typename T>
concept Integer = Same<T,short> || Same<T,int> || Same<T,long>;

Integer x1 = 7;
int x2 = 9;

Integer y1 = x1+x2;
int y2 = x2+x1;

void f(int&);          // a function
void f(Integer&);      // a function template

void ff()
{
    f(x1);
    f(x2);
}
```

C++20 comes close to fulfilling this ideal. To get that example to work in C++20, we have to add a logically redundant **auto** after each use of the concept **Integer** (§6.4). On the other hand, in C++20, we can use the standard-library concept **integral** instead of the obviously incomplete **Integer**.

6.3.6 Improvements. At the start of the Concepts TS effort, a **concept** was a **constexpr** function (§4.2.7) returning a **bool**. That made sense since we saw concepts as compile-time predicates. Then Gabriel Dos Reis got variable templates accepted into C++14 (§5.2). Now, we had a choice:

```
// function style:
template<typename T>
concept bool Sequence() { return Has_begin<T>() && Has_end<T>(); }

// expression style:
template<typename T>
concept bool Sequence = Has_begin<T> && Has_end<T>;
```

We could live happily with either style, but having both, users of a concept would have to know which style was used in its definition to use parentheses correctly. That quickly became impractical.

The functional style allows for overloading of concepts, but we had only few examples of concept overloading and decided we could do without those, so we simplified to use variable templates exclusively for concepts. Andrew Sutton pioneered the consistent use of the expression form of concepts.

We (Andrew Sutton, Gabriel Dos Reis, and I) always knew that explicitly mentioning that a **concept** returned a **bool** was redundant. After all, a concept is by definition a predicate. However, we decided not to mess with the grammar and concentrate on semantically significant topics. Later, people were latching on to the redundant mention of **bool** as an argument against the concept design. So we fixed it and no longer have to mention **bool**.

The removal of **bool** was part of a set of suggested improvements from Richard Smith that included a more precise specification of what constituted an atomic predicate and simplifications of the matching rules [Smith and Sutton 2017]. We now use the expression style:

```
// expression style:
template<typename T>
concept Sequence = Has_begin<T> && Has_end<T>;
```

6.3.7 Syntax Equivalences. The concepts TS supports three notations for using concepts in function declarations:

- explicit **requires**-clauses for full generality
- the shorthand notation for type-of-type uses
- the natural notation (also known as the terse notation, the conventional notation, and more)

The fundamental idea was to let the programmer use a notation that closely matched the needs of a particular declaration without drowning that definition with notation needed by more complex declarations. To allow the programmer a free choice and in particular to allow the notation to be adjusted as functions change during initial development or maintenance, these notation styles were defined as equivalent:

```
void sort(Sortable&); // natural notation
```

is equivalent to

```
template<Sortable S> void sort(S&); // shorthand notation
```

is equivalent to

```
template<typename S> requires Sortable<S> void sort(Sortable&);
```

Users were rather happy with this and tended to prefer the natural and shorthand notations for most declarations. However, some committee members reacted in horror to the natural notation ("I can't see that it's a template!") and rather liked the most explicit notation using **requires** because it can express even the most complex examples ("why would you want anything more than that?"). My interpretation is that we have a clash of two views of what's simple:

- I can write my code in the simplest and shortest way
- I need only to learn one notation

I am in favor of the former view, considering it a good example of the Onion Principle (§4.2).

The natural notation became a focus of strong opposition to concepts. I – and others – insisted on the elegance of

```
void sort(Sortable&); // natural notation
```

We saw (and see) it being a useful and elegant step towards making generic programming "just ordinary programming" rather than a dark art with different syntax, different source code organization ("header only"), and different coding styles (e.g., template metaprogramming (§10.5.2). Modules addresses the source code organization issues (§9.3.1). In addition, that "natural" syntax addresses frequent loud complaints about the template syntax being verbose and clumsy. I agree with those complaints. The prefix **template<...>** syntax was not my first choice when I designed templates. It was forced upon me by people worried that templates would be misused by less competent programmers, leading to confusion and errors. The heavy syntax for exception handling, **try { ... } catch(...) { ... }**, was a similar story [Stroustrup 2007]. It seems that for every new feature many people demand a *LOUD* syntax to protect against real and imagined potential problems. After a while, they then complain about verbosity.

However, a large minority of committee members insisted that the natural syntax would lead to confusion and misuse because people, especially less experienced programmers, would not realize that functions defined in this way were templates and therefore different from other functions. Having used and taught concepts for years without observing these problems, I wasn't particularly concerned about such hypothetical problems, but the opposition turned out to be solid. People simply *knew* that such code was dangerous. The prime example was

```
void f(C&&);  // Danger: Is C a concept or a type?
```

The meaning of **C&&** is somewhat different dependent on whether **f** is a function template or an "ordinary" function. To my mind, this difference in the semantics of **C&&** is a most unfortunate design mistake in C++11 that we should try to correct, rather than let it affect the definition of concepts. The possibility of a misunderstanding is undeniably real and would certainly happen to someone once the facility became used by millions. However, I haven't seen it in real life and doubt that the relatively experienced programmers who would write code where the difference matters would have real trouble with it. In other words, I considered it an example of "the tail wagging the dog;" that is, an obscure example blocking a feature that could benefit large numbers of users.

I'm also pretty sure that my aim of making generic programming as similar to "ordinary" programming as possible wasn't universally shared. There are still people who think that generic programming is beyond that vast mass of programmers. I still see no evidence of that.

6.3.8 Why No Concepts in C++17? I had hoped and expected to see concepts in C++17. Of the extensions that I considered feasible in the 2017 time frame (§9.2), I saw concepts as the one that would yield the most significant improvement to C++ programmers' basic vocabulary. It would eliminate the need for much ugly and error-prone template metaprogramming (§10.5.2), would simplify the precise specification of libraries, and would significantly improve the design of libraries. I fear that was part of the problem: concepts would directly affect essentially all of the voting members. Some were more comfortable with their old ways, many had no experience with concepts, and some considered them an untried ("academic"/"theoretical") idea.

It was such worries – fueled by the fiasco of C++0x concepts (§6.2) – that led to us having a TS [Sutton 2017] in the first place. We didn't have experience with Technical Specifications for language features, but it seemed worth trying and Andrew Sutton's concepts implementation in GCC was still new so caution seemed warranted. At the Bristol standards meeting (2013), Herb Sutter strongly argued for the TS route whereas J-Daniel Garcia and I warned against likely delays. I also pointed to dangers of considering concepts separately from generic lambdas (§4.3.1), but "caution" and "we need more experience" are strong arguments in a standards committee. In the end, I voted in favor of concepts TS. I now consider that a mistake.

In 2013, we had an implementation and a pretty good specification (primarily thanks to Andrew Sutton), but completing the concepts TS still took three years. I am unable to discern a difference in rigor in the treatment of the TS and inclusion into the ISO standard proper. However, at the 2016 Jacksonville meeting, when concepts as described in the TS came up for a vote for inclusion into the standard, all the earlier arguments against were repeated. It seemed that the opponents had just ignored concepts for three years. I even heard arguments against concepts that were valid for C++0x concepts but had never been relevant to the concept TS design. People again argued for "caution" and "we need more experience." As far as I could tell, there were more people present in Jacksonville who had not tried concepts than there had been in Bristol – partly an effect of the increased size of the committee. In addition to all the objections I had heard over the previous decade, novel objections were raised and untried designs were suggested in full committee, yet taken seriously.

At the meeting in February 2016 Jacksonville, Ville Voutilainen (the EWG chair) proposed to move the concepts as specified in the Concepts TS [Voutilainen 2016c]:

> ... *"programmers are aching to get the language feature into their hands, and it's high time we ship it to them. Conceptifying the standard library will take its time, and we will not find major design issues for Concepts while doing it. We shouldn't keep programmers waiting for the language feature out of concern for hypothetical design issues that we have no proof of, have some amounts of counter-proof of, and very likely don't exist. For the benefit of C++ users everywhere, let's ship Concepts the language feature in C++17."*

He was strongly supported by many, notably Gabriel Dos Reis, Alasdair Meredith (formerly, the LWG chair) and me, but (despite a positive EWG vote earlier in the week), the vote went against us: 25 in favor, 31 against, 8 abstained. My interpretation is that the users voted for and the language technicians voted against, but that may be considered sour grapes.

At the same meeting, uniform call syntax (§8.8.3) was voted down and coroutines (§9.3.2) sent to a TS, basically ensuring that C++17 would be a minor release of the standard (§8).

6.4 C++20 Concepts

In 2017, as one of the very first features for C++20, WG21 voted what they considered the foundational and uncontroversial parts of the Concepts TS [Sutton 2017] into the Working paper (§6.3.2):

- explicit **requires**-clauses for full generality; e.g., **requires Sortable<S>**
- the shorthand notation for type-of-type uses; e.g., **template<Sortable S>**

The natural notation (e.g., **void sort(Sortable&);** (§6.3.7) was left out as controversial. The reasons given for wanting it left out were:

- It is not obvious that **void sort(Sortable&);** is a template.
- The meaning of **void f(C&&);** depends on whether **C** is a concept or a type.
- In **Iterator foo(Iterator,Iterator);** must **Iterator** be the same type in all three cases or can the types be separately constrained?
- The natural syntax is confusing and hard to teach.
- How do we constrain the parameter in **template<auto N> void f();**?

These objections were not new, but they were accompanied with a lot of proposals for novel syntax [Honermann 2017; Keane et al. 2017; Köppe 2017a; Riedle 2017; Sutter 2018a]. These proposals were all different and all incompatible with the Concepts TS. They were presented with significant amount of emotion in meetings. None were supported by actual experience. To contrast, my position was based on about four years of teaching, much experimental use, some industrial use,

and several uses in proposed standard-library components (e.g., iterator facade [Dawes et al. 2016], tuple implementation [Voutilainen 2016b], ranges [Niebler et al. 2014]).

At the Jacksonville meeting (2018), Tom Honerman proposed to remove the natural syntax and offered an alternative [Honermann 2017]. I defended my position and the concept TS design [Stroustrup 2017a,b]. My basic defense was

- The natural syntax has not caused problems in real-world teaching and use for more than five years.
- Users like it.
- There are no technical ambiguities.
- It simplifies common cases.
- It is part of the drive to make generic programming more like ordinary programming.

That failed to convince anyone who was opposed in the first place, so the natural syntax was not moved to the working paper for C++20.

The last objection came from a new C++17 minor feature, **auto** for value arguments [Touton and Spertus 2015], and became a focal point for objections:

```
template<auto N> void f();
```

People wanted to syntactically distinguish value and type template arguments. Typically, that would imply that the shorthand syntax that had been the stable of proposal and uses since 2002 would no longer be valid.

```
template<typename T> void f(T&);  // proposed banned
```

In mid-2018, I suggested a minimal compromise [Stroustrup 2018b] that:

- Preserved the meaning of **template<Concept T> void f(T&);**
- Used a **template** prefix to identify template using the natural notation, e.g., **template void f(Concept&);**

That proposal succeeded, but so did a very different proposal from Herb Sutter [Sutter 2018a]. We were now in the extraordinary position of having two very different and incompatible proposals each supported by a large majority of the EWG. This deadlock opened the door for Ville Voutilainen (the EWG chair) to suggest a variant that was accepted with massive support in November 2018 [Voutilainen et al. 2018]:

- Preserved the meaning of **template<Concept T> void f(T&);**
- Used an **auto** to identify template arguments using the natural notation, e.g., **void f(Concept auto&);**

For example

```
// almost natural notation:
void sort(Sortable auto& x);     // x must be Sortable
Integral auto ch = f(val);       // the result of f(val) must be Integral
Integral auto add(Integral auto x, Integral auto x);   // can use a wider type
                                                       // to prevent overflow
```

The “natural notation” was renamed the “abbreviated syntax” even though it is not simply an abbreviation.

I supported the compromise even though I consider that use of **auto** redundant, distracting, and compromising my aim of making generic programming “just ordinary programming.” Maybe sometime in the future, people will (as Herb Sutter suggested at the time) agree and make **auto** after a concept name redundant. I am not holding my breath, though; there is a large group of people

who consider syntactic markers for implementation techniques important. Maybe auto-completion features of IDEs will save users from writing the redundant **auto**s.

Sadly, there was no consensus for re-introducing concept name introducers (§6.3.4). The lack of a sufficiently conventional syntax was a major stumbling block. Also, quite a few people still seem not to believe in their utility.

The years of delay introducing concepts caused long-lasting harm. Ad-hoc designs based on traits and **enable_if** proliferated. A generation of programmers was brought up on low-level, typeless metaprogramming.

6.5 Naming of Concepts

One of the final discussions about concepts before shipping C++20 was about naming conventions. Naming is always a tricky topic. In my early work with concepts, I named concepts the way I usually name types that are not in the standard: capitalize the first letter as for proper nouns and separate words with an underscore for readability (e.g., **Sortable** and **Forward_iterator**). Others (notably the Indiana team) used CamelCase (e.g., **Sortable** and **ForwardIterator**). Unfortunately, this naming convention crept into the standard text [Carter 2018] and caused some confusion by being different from all other names in the standard library. There, underscores and no capital letters are used (except for MACROS and three obscure, hard to find, examples). Some people then thought that the different naming convention aimed to distinguish the "novel and difficult" concepts from "ordinary constructs", such as functions and types.

When I noticed that rationalization, I strongly disliked it. In C++, we generally don't encode types in the names of entities, but I thought it too late to change the naming style. In 2019, Herb Sutter responded to my grumbles by proposing to rename all standard-library concepts to follow the usual standard-library naming convention [Sutter et al. 2019]. All the major concept designers and the range-library (§9.3.5) designers signed on as co-authors. Another reason for the change was that we were beginning to see clashes between the CamelCase names of standard-library concepts and CamelCase names in other libraries. One reason for using CamelCase (or using my convention of capitalizing types) is exactly to avoid clashes with the standard library. So, we now have **sortable**, **forward_iterator**, etc.

The C++20 standard-library has some 70 concepts covering the needs of invocation of operations, use of basic types, ranges, and the standard algorithms, including **constructible_from**, **convertible_to**, **derived_from**, **equality_comparable**, **invocable**, **mergeable**, **range**, **regular**, **same_as**, **signed_integral**, **semiregular**, **sortable**, **swappable**, and **totally_ordered**. They will guide much C++ library design. Note that many of those 70 concepts are not fundamental but simply there for notational convenience or as building blocks.

7 ERROR HANDLING

Error handling is – and I suspect will remain – a hotly debated topic. Many people have strong opinions – some based on solid experience in a variety of application areas – and many different techniques have been developed over the last 50 years or so. This is an area where the demands of performance, generality, and reliability easily get into conflict.

As is often the case with C++, the problem is not that we don't have a solution, but that we have many. It is fundamentally hard to address the diverse needs of the C++ community with a single mechanism, but looking at a subset of the problem people often think they have *the* solution [Stroustrup 2019a].

7.1 Background

From C, C++ inherited a variety of techniques based on error return codes, errors represented as special values, global state, local state, and callbacks. For example:

```
double sqrt(double d);  // set errno==33 if d is negative
int getchar();          // return -1 if end of file
char* malloc(int);      // return 0 if allocation failed
```

Early users of C++ (1980s) found those techniques confusing and insufficient. Returning a (value,error-code) pair became popular, adding to the variety and confusion. For example:

```
Result r = make_window(arguments);  // Result is a (value,error) pair
If (r.error) {
    // ... handle error ...
}
Shape* p = r.value;
```

That led to code becoming cluttered with tedious repeated error checks. When using error codes, it can be hard to distinguish the main logic of a program from error handling. The main line of a program ("the business logic") gets deeply intertwined with the handling of a multitude of rare and obscure errors. This can be a serious problem in the not unusual systems where the error-handling is the majority of code and the most complex part of the code.

The use of a class containing a (value,error-code) pair came with a significant cost. In addition to the cost of the test of the error-code, many ABI's did not pass even small structures in registers, so not only was more information passed (often twice the amount), but it was passed in a way that could be an order of magnitude slower. Sadly, this problem persists to this day (2020) in many ABIs, especially ABIs for embedded systems (designed for C code).

Furthermore, there was no really good way of using error codes to handle failure in a constructor (there is no return value in which to pass an error-code) and the then-fashionable large complicated class hierarchies led to complex and error-prone handling of the variety of potential errors from sub-object creation.

In addition, all traditional error handling techniques suffer from people forgetting to check for errors. This was – and still is in 2020 – a major source of errors. Minimizing errors from incomplete or complex error handling was a major goal for C++ exceptions.

Exceptions for C++ were designed in 1988-89 to address the mess of complicated and error-prone error-handling techniques then prevalent. They were documented in the ARM [Ellis and Stroustrup 1990] and adopted into ANSI C++ as part of the base document for standardization [Stroustrup 1993].

Compared to some exception designs, the one for C++ was complicated by the need to use C++ code in combination with code in other languages, especially C. Consider a C++ function, **f()** calling a C function **g()** that in turn calls a C++ function **h()**. Now **h()** throws an exception for **f()** to catch. In general, a C++ function does not know the implementation language of a called function. This scenario precludes adoption of implementation schemes that modify all function signatures to add an "exception propagation argument" or implicitly add a return code to the return type.

Exceptions together with RAII (§2.2) did solve many of the most vexing error-handling problems (such as how to handle an error in a constructor and how to handle errors detected far from code that could handle them) with a very minor run-time cost compared to use of other techniques (usually less than 3% by the mid-1990s and sometimes even cheaper than alternatives). Exceptions were never uncontroversial, though, and I underestimated their potential for controversy.

7.2 Real-World Problems

There have always been applications for which the use of exceptions was unsuitable. Examples include

- Systems where the memory is so limited that the run-time support needed for exception handling crowds out needed application functionality.
- Hard-real time systems where the tool chains cannot guarantee prompt response after a throw (e.g., [Lockheed Martin Corporation 2005]).
- Systems relying on multiple unreliable computers so that immediate crash-and-restart is a reasonable (and almost necessary) way of dealing with errors that cannot be handled locally.

Consequently, most C++ implementations always had a no-exception switch. On the other hand, there are problems for which error codes provide no good solution:

- *Constructor failures* – there is no return value (except the constructed object itself). Pure RAII must be replaced by explicit checks on object states.
- *Operators* – there is no place to return an error indicator from **++**, *****, or **->**. You will have to use non-local error indicators or live with an impoverished notation, e.g., **multiply(add(a,b),c)** rather than **(a+b)*c**.
- *Callbacks* – where the function using the callback should be able to invoke functions with a wide variety of possible errors (often, callbacks are lambdas (§4.3.1)).
- *Non-C++ code* – there is no way to propagate an error through a function that is not C++ and hasn't been written specifically to deal with error codes.
- *Addition of a new kind of error deep in a call chain* – every function on the call chain must be prepared to handle or propagate a new kind of error (e.g., a network error in a program not specifically designed to access data across a network).
- *People forgetting to test a return code* – there are clever schemes to try to ensure that error-codes are checked consistently, but they are either incomplete or rely on exceptions or termination once a failure to check is detected (e.g., [Botet and Bastien 2018]).

In addition, there were practical problems related to the use of exceptions:

- Some people could not introduce the use of exceptions because their code already was a large irreparable mess of unprincipled pointer use. Quite often, such people directed their critique toward exceptions rather than their old code.
- Some people (many) simply didn't understand or even didn't know of RAII (§2.2) and used exceptions as merely as an alternative return mechanism mirroring the use of error-return codes. Typically, code using try-catch as a form of if-then is uglier, larger, and slower than proper use of error codes or RAII.
- Many implementations of exceptions were slow because implementers generalized to handle other kinds of exceptions (e.g., Microsoft's "structured exceptions"), prioritized debugging (e.g., GCC walks the stack twice after a **throw** to preserve backtraces), used a single mechanism to serve a variety of languages (equally badly), or simply didn't expend much development effort on optimization.
- Exception handling have become *relatively* slower over the years because significant efforts have been spent optimizing non-exception cases. I suspect there are significant optimization opportunities left. For example, Gor Nishanov reported up to 1000 times speed improvements for some simple optimizations related to coroutine implementations on Windows and Linux [Nishanov 2019a]. Significant space improvements will likely be harder to achieve, though. Some recent experiments are promising [Renwick et al. 2019].

- To get exceptions accepted, we had to add exception specifications [Stroustrup 2007]. They never provided the improved maintainability that their proponents claimed, but they did deliver the verbosity and overhead that their opponents (including me) insisted they would. Once exception specifications were in the language, many people felt encouraged to use them and blamed exceptions in general for the resulting problems. Ironically, the people who insisted most loudly on getting exception specifications went on to help design Java. Exception specifications were deprecated in 2010 and finally removed in 2017 (§4.5.3). Partly as an alternative, C++11 introduced **noexcept** as a simpler and more efficient mechanism for controlling exceptions (§4.5.3).
- Catching an exception is done by specifying the type of exception to be caught. As a result, the implementations of **throw** and **catch** got entangled with the mechanism for runtime type information (RTTI [Stroustrup 2007]). This caused inefficiencies and complexity. In particular, it caused memory to be consumed (by the data needed for RTTI) even if an application never relied on RTTI for distinguishing exceptions and precluded optimization for simple cases. Furthermore, relying on RTTI made it hard to optimize the type matching where dynamic linking was used. Basically, exception handling implementations were tuned for the rare most complicated case. This was made worse when an exception class hierarchy was added to the standard library and people were encouraged to use that for even the simplest cases. For class hierarchies that can be statically analyzed (as in many embedded systems) fast constant-time type matching is possible [Gibbs and Stroustrup 2006]. The fact that exceptions are part of the platform ABIs makes it very hard to change early overdesigned implementations.
- Some people insist that only a single method of error handling be used and usually concludes that since exceptions are not suitable for every case, that method must be error-codes. The problems with error codes are then deemed "just inconveniences."
- Some people believed the persistent rumors of inefficiencies based on worst-case scenarios and/or unrealistic comparisons, such as leaving error-code handling in place after adding exceptions, comparing incomplete error-handling to exception-based handling, or using exceptions to handle ordinary choices rather than for just handling errors that cannot be handled locally. There has been far too little serious investigation into the cost of exception and its alternatives. I suspect that the myths about exceptions have had more influence than any fact.

The net effect was a bifurcation of the C++ community into exception and non-exception camps. De facto, "use no exceptions" is a dialect and dialects is one of the things standards are meant to avoid (§3.1). A dialect can be an advantage for an individual organization or community, but harms the C++ community as a whole by complicating sharing of code and competences.

It has been claimed that the problem with exceptions comes from it violating the zero-overhead principle (e.g., [Sutter 2018b]). For an error-handling scheme that responds to every error by terminating, any error-handling mechanism is obviously overhead, so the zero-overhead principle is violated (unless you take into account the cost of handling termination, e.g., in another processor). At the time of the design of exceptions, we considered that and thought it acceptable based on the argument that such cases were rare, that there was no run-time cost unless an exception was thrown, and that the tables used to implement exceptions could be kept in virtual memory [Koenig and Stroustrup 1989]. Where virtual memory isn't available and memory is scarce, the use of tables to implement exceptions can be a serious problem. The main concern at the time was systems where some form of error propagation and error handling were needed. In such cases, the zero-overhead was interpreted "as no overhead for exceptions compared with the use of error codes for the same degree of rigor of error handling."

Today, the confusion about error handling is worse than ever and the number of alternative techniques for handling errors is higher than ever, causing much confusion and harm. Given N ways of handling errors, someone comes up with a solution just to find that the old solutions don't go away so now we have to cope with N+1 ways ("the N+1 problem"). If an organization has M separately developed programs using N libraries, we may even have an N*M problem. The introduction of exceptions could be seen as having increased the number of popular ways of handling errors from 7 to 8. In 2015, Lawrence Crowl wrote an analysis of the problems [Crowl 2015a].

The problem of multiple error-reporting schemes is felt most acutely by writers of foundational libraries. They cannot know what their users prefer and their users may very well have many different preferences. The authors of the C++17 filesystem library (§8.6) chose to duplicate their interfaces: for each operation, they offer two functions, one that throws in case of error and another that sets a standard-library **error_code** passed to it as an argument:

```
bool create_directory(const filesystem::path& p); // throws in case of error
bool create_directory(const filesystem::path& p, error_code& ec) noexcept;
```

This, of course, is a bit verbose and pleases only people who like either exceptions or **error_code**s. Note how the **bool** return value is supplied so that people don't have to use **try** or directly test the **error_code** all the time. The fact that the file systems (quite properly, IMO) uses exception for rare errors doesn't please people who consider exceptions fundamentally flawed. In particular, it requires that exception support be present.

7.3 noexcept Specifications

Using **noexcept** (§4.5.3), people can suppress all exception throws from a function and allow callers to ignore the possibility of exceptions being thrown.

The use of **noexcept** can reassure people who worry about performance problems (real and imaginary). It can also improve optimization by decreasing the number of control paths, but only if the programmers don't add those paths back by testing return codes. There are lots of low-level functions that are naturally **noexcept**, such as most C functions.

The use of **noexcept** can simplify error handling (if a function can't throw, we don't need to catch any exceptions) or complicate it (if a function cannot throw, yet may fail, we have to use another error-reporting mechanism). In particular, a **noexcept** on a path between a **throw** and its handler can turn a **throw** into termination. Thus, making a function **noexcept** during maintenance can cause failure of a previously correct program.

Note that one of the significant reasons that exceptions were added to C++ was to support applications where an error must never lead to an unconditional abort. An exception simply signals that a failure has happened and any code on the path from **main()** to the **throw**-point can take control. In particular, this supports the important use case of doing some local cleanup (e.g., flushing some output buffers or adding an error-report to a log file) before terminating.

7.4 Type System Support

The traditional approach to logical and performance problems in C++ is to move computation from run time to compile time. Obviously, the possibility of integrating exceptions with the static type system was seriously considered in the 1980s and repeatedly reconsidered later. If exceptions were part of every function type, programs would be better type checked, functions more self-documenting, and exception handling easier to optimize.

A major reason not to make exceptions part of the type system was the observation that if exceptions were part of a function's type, a change to the set of exceptions it could possibly throw

would require all functions calling the function to be recompiled. In a world where most major programs are composed from many separately developed libraries, this would lead to disastrous brittleness and unmanageable dependencies [Stroustrup 1994].

There are also obvious problems related to pointers to functions. There was – and still is – a lot of C-style code in most major C++ programs. The main parameterization mechanism for C-style generic code (e.g., **qsort** parameterized by its comparison criteria) and callbacks (e.g., in GUIs) is pointers to functions.

If I need a pointer to function and exceptions are part of the type system, I either have to decide to always require exceptions from the function pointed to, never accept exceptions, or somehow handle both alternatives. Handling both is hard unless support for type inquiry or overloading based on exceptions are added to the language. Having decided what kind of pointer-to-function argument(s) to accept, I now have to adjust the error checking in the calling function to match. Even if this could be handled in C++, interaction with C would be impeded: how would a pointer to a C++ function be passed to C? For example, what would callbacks from C to a C++ program relying on exceptions look like? Obviously, the original C++ exceptions would not go away, so we would have four alternatives: error codes, compile-time checked exceptions (e.g., [Sutter 2018b]), current exceptions, and **noexcept**. Only the current exceptions and non-local error-codes do not affect the type system or the calling conventions (ABIs). Fortunately, few functions require two pointers to functions or we would risk having to deal with 16 alternatives. If different exception types were accepted (as for the current exceptions), the chaos would be complete.

In modern C++, this kind of problem would persist in different guises for other callback mechanism, such as objects with member functions meant to be called, function objects, and lambdas.

My conclusion (as endorsed by WG21) was and is that adding exceptions to C++'s static type system would lead to brittleness, significant increases in complexity of code, serious incompatibilities, and problems with interaction with C code. This was appreciated in 1989.

7.5 Back to Basics

Fundamentally, I think C++ needs two error-handling mechanisms:

- *Exceptions* – for errors that are rare or cannot be handled by an immediate caller.
- *Error codes* for errors that can be handled by an immediate caller (often "disguised" in easy-to-use test operations or returned from a function as a (value,error-code) pair).

Consider:

```
void user()
{
    vector<string> v {"hello!"};
    for (string s; cin>>s; )
        v.push_back(s);
    auto ps = make_unique<Shape>(read_shape(cin));
    Smiley_face face{Point{0,0},20};
    // ...
}
```

This example is artificial, but stylistically not atypical. The **user()** function offers many opportunities for unlikely errors: memory exhaustion, read errors, construction failures (e.g., deep in the hierarchy of **Smiley_face**). In addition, the use of a **unique_ptr<Shape>** protects against a memory leak. If we used explicit error-codes instead of exceptions, we would need at least five error-checks in this function, doubling the amount of source code, plus a few more checks inside the various constructors. Without RAII (and its integration with exceptions) the code would bloat

further still. On average, more code implies more errors. This is especially so when the added code complicates control flows. This point is often underappreciated by people who argue from small examples. For small examples, "just one test" doesn't matter much and is relatively hard to forget.

On the other hand, some errors should be expected and for those checks of some form of error code is preferred:

```
ifstream f {"Myfile"};
if (!f) {
    // ... deal with error ...
}
// ... use f ...
```

Here, the error-code is hidden inside the stream state for ease of use.

So, ideally, there would be just two ways of handling an error: However, I do not know how to get to such an ideal state. There are on the order of a dozen variants of the (value,error-code) pair idea in wide use (e.g., **std::map::insert()**) and several new ones were being discussed in WG21 in 2018 (e.g., [Botet and Bastien 2018; Sutter 2018b]). Even if the committee could agree on one, there would still be at least a dozen widely used error-handling schemes "out there" each supported by large groups of devoted followers, many with millions of lines of hard-to-change code.

There has been little serious research on the topics of performance of exceptions and reliability of the resulting code in C++ ([Renwick et al. 2019] is an exception). There are, however, many small unscientific studies and lots of loudly expressed opinions – often claiming exceptions to be inherently slower than various forms of checking of error-codes. That is not my experience. To the best of my knowledge no half-way serious study has failed to observe that there are realistic examples where error codes win big and realistic examples where exceptions win big. In this context, "big" means integer factors, rather than a few percent.

Run a simple performance test: go N levels deep into a call sequence and then report an error. If the error is rare, say 1:1000 or 1:10000 and the call nesting is deep, say 100 or 1000, exception handling is much faster than explicit tests. If the call depth is 1 and the error happens 50% of the time, explicit tests win big. The call depth and error probability determine the difference between the such examples. My naive, but potentially useful question is "how rare must an error be to be considered exceptional?" Unfortunately, the answer is "that depends." It depends on the complexity of the code, the hardware, the optimizer, the implementation of exception handling, and more. C++ exceptions were designed assuming an answer at least in the 1:100 region. In other words, that propagation of error indicators is far more common than explicit handling.

The space problem is likely to be harder to solve than the run-time problem. For systems relying on immediate termination for all errors that cannot be handled locally, I could imagine an implementation simply terminating on a **throw** but if errors are to be propagated and handled, the tradeoff difficulties won't go away.

Any solution of the error-handling mess is liable to hit the N+1 problem (§4.2.5) [Stroustrup 2018a].

Curiously enough, one of the worries at the time where exceptions were included in C++ was that they were not sufficiently general. A significant number of people considered resumption semantics essential [Stroustrup 1993]. My guess at the time was that allowing resumption would have slowed down exception handling by at least a factor of two.

8 C++17: LOST AT SEA

After the minor standards revision, C++14, C++17 [Smith 2017] was to have been a major revision of the standard. C++17 has quite a few new features, but none that I would deem major. The key

question about C++17 is "Why did all of that hard work not lead to more significant improvements?" The processes that led to the C++11 and C++14 successes were in place, the standards community was more experienced, larger, and more enthusiastic.

C++17 had about 21 new language features (depending how you count), including:

- Constructor template argument deduction – simplify object definitions (§8.1)
- Deduction guides – an explicit notation for resolving constructor template argument deduction ambiguities (§8.1)
- Structured bindings – simplify notation and eliminate a source of uninitialized variables (§8.2)
- **inline** variables – simplify the use of statically allocated variables in header-only libraries [Finkel and Smith 2016]
- Fold expressions – simplify some uses of variadic templates [Sutton and Smith 2014]
- Explicit test in conditions – a bit like conditions in **for**-statements (§8.7)
- Guaranteed copy elision – eliminate many redundant copy operations [Smith 2015]
- Stricter expression evaluation order – prevents some subtle order-of-evaluation mistakes [Dos Reis et al. 2016b]
- **auto** as a template argument type – type deduction for value template arguments [Touton and Spertus 2016]
- Standard attributes to catch common mistakes – **[[maybe_unused]]**, **[[nodiscard]]**, and **[[fallthrough]]** [Tomazos 2015]
- Hexadecimal floating-point literals [Köppe 2016a]
- Constant expression **if** – simplify compile-time evaluated code [Voutilainen and Vandevoorde 2016]

Unfortunately, this is not quite the full list of extensions. Quite a few are so small that they are not easy to briefly describe.

The C++17 standard-library added about 13 new features plus many minor modifications:

- **optional**, **variant**, and **any** – standard-library types for expressing alternatives (§8.3)
- **shared_mutex** and **shared_lock** (reader-writer locks) and **scoped_lock** (§8.4)
- parallel STL – multi-threaded and/or vectorized versions of standard-library algorithms (§8.5)
- file system – the ability to portably manipulate file-system paths and directories (§8.6)
- **string_view** – a non-owning reference to an immutable sequence of characters [Yasskin 2014]
- Mathematical special functions – including Laguerre and Legendre polynomials, beta functions, Riemann zeta function [Reverdy 2012]

I have not been able to identify any unifying themes for the C++17 features. These features look to be merely a set of "bright ideas" thrown into the language and standard library as voting majorities could be found. That bothers me a lot. It bothers me a lot even when I like an individual feature. There was no overall plan. That boded ill for the future – something had to be done [Stroustrup 2018d]. The creation of the Direction Group was part of WG21's response (§3.2) (§9.1).

C++17 undeniably offers something that can help most programmers in minor ways, but nothing that I can deem major. In this context, I define "major" as "makes a difference to the way we think about programming and organize our code." Here, I describe the facilities that I suspect will have the largest positive impact.

I also examine a few examples that, despite serious consideration, did not make it into C++17:

- §6.3.8: Concepts (C++20)
- §8.8.1: Networking library
- §8.8.2: Operator dot (**operator.()**)

- §8.8.3: Uniform Function Call
- §8.8.4: Default comparison operators ==, !=,<, <=, >, and >= for sufficiently simple types
- §9.3.2: Coroutines (C++20)

I suspect that had they been adopted, each of these would have been among the most significant features of C++17. They fit a coherent view of what C++ should become (§9.2) and even a couple of those would have drastically changed the ways C++17 could have been used.

Looking at C++11, I see a dense web of mutually supportive features leading to support of better ways of writing code. I don't see that for C++17. However, C++20 completes such a web to bring C++ another major step forward (§9). It could be argued that C++17 was just a stepping stone on the way to C++20, but the committee discussions gave little hint on that; the focus was firmly on individual features. I have even heard claims that the train model (§3.2) precludes long-term planning; it does not.

8.1 Constructor Template Argument Deduction

For decades, people have wondered why template arguments could be deduced from function arguments, but not from constructor arguments. For example, in C++98, C++11, and C++14:

```
pair<string,int> p0 (string("Hi!"),129);  // no deduction
auto p1 = make_pair("Hi!"s,129);          // p1 is a pair<string,int>
pair p2 ("Hi!"s,129);    // error: template arguments missing for pair
```

Naturally, I had considered the possibility of deducing template arguments from constructor arguments when I first designed templates, but I was deterred by fear of ambiguities. There were also technical obstacles to a solution, but Michael Spertus and Richard Smith overcame them [Spertus and Smith 2015] so in C++17 we can write that last example (**p2**) without an error, removing the need for the **make_pair()** workaround.

This simplifies the use of types, such as **pair** and **tuple**, and also writing of concurrent programs using locks and mutexes (§8.4).

```
shared_lock lck {m};   // no explicit lock type
```

This is a rare example of mutually supportive features in C++17 leading to significantly simpler code. Unfortunately, these simplifications were accepted (or not) on a case by case basis, rather than in general, so the efforts to "fill holes" in the type deduction rules continues [Spertus et al. 2018].

In addition to what is described here, the facility provides a notation for resolving ambiguities. For an example, see (§8.3).

8.2 Structured Bindings

Structured bindings started out as a simple proposal by Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis [Sutter et al. 2015] to simplify notation and eliminate one of the few remaining sources of uninitialized variables. For example:

```
template<typename T, typename U>
void print(vector<pair<T,U>>& v)
{
    for (auto [x,y] : v)
        cout << '{' << x << ' ' << y << "}\n";
}
```

The names **x** and **y** are bound to the first and second elements of the **pair**. That can be a significant notational convenience.

C++14 had given us convenient ways of returning multiple values. For example:

```
tuple<T1,T2,T3> f(/*...*/)  // nice declaration syntax
{
    // ...
    return {a,b,c}; // nice return syntax
}
```

I consider **tuple**s somewhat overused in current C++ and prefer specifically defined classes when the multiple values are not independent, but notationally that makes no difference. However, C++14 did not offer a convenient way of unpacking such multiple return values to match the convenient ways of creating them. This led to verbose workarounds, uninitialized variables, or run-time overhead. For example:

```
Tuple<T1,T2,T3> res = f();
T1& alpha = get<0>(res);          // indirect access through alpha
T2& val = get<1>(res);
T3 err_code = get<2>(res);       // copy
```

Many experts preferred to use the standard-library function **tie()** for unpacking **tuple**s:

```
T1 x;
T2 y;
T3 z;
// ...
tie(x,y,z) = f(); // nice call syntax, with existing variables
```

Assigning to a **tie()** assigns to the variables used as arguments to the **tie()**. However, to use **tie**, you have to separately declare the variables and name their types to match those of the members of the object returned by **f()** (here, **T1**, **T2**, and **T3**). Unfortunately, that leaves the local variables open to use-before-set errors and initialization-followed-by-assignment overhead. Also, most programmers don't know that **tie()** exists or consider it too odd to use in real code.

Herb Sutter suggested a solution that basically mirrors the return syntax:

```
auto {x,y,z} = f(); // nice call syntax, introducing aliases
```

This would work for any **struct** holding three members, not just for **tuple**s. The elimination of the second-to-last source of uninitialized variables in the Core Guidelines (§10.6) was the major motivation for me. Yes, I liked the notation, but the important issue was the improved approximation to the ideals of C++.

Not everybody liked the idea and we almost didn't get to discuss it in time for C++17. The paper proposing structured bindings [Sutter et al. 2015] was late and Ville Voutilainen was about to close the EWG at the end of the November 2015 meeting in Kona when I noticed that we had about 45 minutes left before lunch and I thought the group would like to see this proposal. Kona 2015 was when we froze the feature set to be worked on for C++17, so those 45 minutes were critical. We didn't even have time to fetch Herb from another group, so I presented. The EWG liked the proposal; the minutes say *Encouragement by acclamation; EWG wants something like this.*

Now, the real work started.

In this and later meetings, several people – notably Chandler Carruth – pointed out that to meet the ideals for C++, we needed to extend the ability of break an object into N values to cope with types that are not **tuple**s or plain **struct**s. For example:

```
complex<double> z = 2+3i;
auto {re,im} = sqrt(z);      // sqrt() returns a complex value
```

The standard-library **complex** does not expose its representation.

For C++17 we solved this problem by allowing the user to define a set of **get** functions, e.g., **get<0>** and **get<1>** effectively pretending the result being a **tuple**. This works but requires the user to supply some inelegant boilerplate code. The discussions about potential improvements continues but no significant simplifications made it into C++20.

There were demands for having this work with functions returning arrays and functions returning **struct**s with bitfields. Support for that was added, so the resulting design was at least twice as complicated as the proposed one.

There was a longish debate (spread over several meetings) over whether it should be possible (or required) to explicitly specify the type of the local variables introduced. For example:

```
auto {int x, const double* y, string& z} = f();   // not C++
```

The arguments for that, most eloquently expressed by Ville Voutilainen, was that without explicit types, the notation decreased readability, could damage maintainability, and could lead to errors. These are very similar to the arguments against **auto** in general and having explicit types brings their own problems. What if the types don't match what is being returned? Some said it should be an error. Some said that conversions to the specified type would be very useful (e.g., **char[20]** returned into a **string**). I pointed out that structured binding was supposed to introduce zero-overhead aliases and any conversion that implied a change in representation would cause significant overhead. Also, one purpose of structured binding was to improve notation and requiring explicit types would lead to code that was more verbose than existing alternatives.

The original proposal used braces ({}) to group the names being introduced:

```
auto {x,y,z} = f(); // nice call syntax, introducing aliases
```

However, some members, notably Chandler Carruth and David Vandevoorde, feared syntactic ambiguities and insisted that would be confusing "because **{}** means scope". So we got the **[]** syntax:

```
auto [x,y,z] = f(); // call syntax, introducing aliases
```

This is a minor change, but I think a mistake. It was a last-minute change and led to minor grammar complications with the attribute notation (e.g., **[[fallthrough]]** (§4.2.10). I don't buy the aesthetic or scope arguments, and in 2014 I had presented ideas of adding functional-programming-style pattern matching to C++ using **{ ... }** for patterns to break out values (§8.3). The structured bindings had been designed to fit into that general scheme.

These were not the only proposals for late changes. Every proposal added or would have added complexity.

It is dangerous to consider one addition to a language at a time. Last-minute changes are dangerous unless they fit into a larger scheme. They also easily lead to "bloat" through demands for "completeness." In this case of structured bindings, I am not convinced that agreeing to allow structured bindings to refer to bitfields offer sufficient utility to warrant the increased complexity.

8.3 variant, optional, and any

Alternative types can be represented without run-time overhead by a **union**. For example:

```
union U {
    int i;
    char* p;
};

U u;
// ...
int x = u.i;    // OK: iff u holds an integer
char* p = u.p;  // OK: iff u holds a pointer
```

This has been used and misused since the earliest days of C as the fundamental way of different types "timesharing" a memory location. There are no compile-time or run-time checks to ensure that a location is used only as the type it actually holds. Ensuring consistent use of **union** members is the programmers' job, and programmers get it wrong with sickening regularity.

Experienced programmers avoid the problem by encapsulating a union in a class that guarantees proper use. In particular Boost offered three such types

- **optional<T>** – holds a **T** or nothing
- **variant<T,U>** – holds a **T** or a **U**
- **any** – holds any type

The great utility of these types has been demonstrated in C++ and many other languages.

The committee decided to standardize those three types. Unfortunately, the design of those three types was discussed separately as if their use cases were disjoint. The possibility of direct language, as opposed to standard-library, support appears not to have been seriously considered. The result was three standard-library types that were (like their Boost ancestors) dramatically different. So, despite the undoubted utility of these types, they are a classic example of design by committee. Consider:

```
optional<int> var1 = 7;
variant<int,string> var2 = 7;
any var3 = 7;

auto x1 = *var1;                // dereference the optional
auto x2 = get<int>(var2);       // access the variant as a tuple
auto x3 = any_cast<int>(var3);  // convert the any
```

To extract the value stored, I have to use one of three incompatible notations. That's a burden on programmers. Yes, experienced programmers will get used to it, but there shouldn't be an irregularity for people to get used to.

To simplify the use of **variant**s, a visitor mechanism is provided. First we need a helper template to define overloat an set:

```
// boilerplate for simple visitation:
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;
```

That **overloaded** template really ought to be standard. It is simple only to people comfortable with variadic templates (§4.3.2) and template argument deduction guides (§8.1). However, given **overloaded**, I can construct a switch on the type of a variant:

```
using var_t = std::variant<int, long, double, std::string>;  // a variant type

// boilerplate for simple visitation:
template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

void use()
{
    std::vector<var_t> vec = {10, 20L, 30.40, "hello"};

    for (auto& var: vec) {
        std::visit(overloaded {
            [](auto arg) { cout << arg << '\n'; },   // handle the integers
            [](double arg) { cout << "double: " << arg << '\n'; },
            [](const std::string& arg) { cout << "\"" <<arg << '\"\n'; },
        }, var);
    }
}
```

Indisputably, **variant** and friends address an important problem, but inelegantly. Maybe future work can reduce the confusing variations in the interfaces to what genuinely needs to differ. In the meantime, the problem is to get the wider C++ community to use these new types well so as to eliminate most of the decades' old problems with **union**s.

I see these three variants of the idea of a discriminated **union** as a stop-gap measure. Functional-programming-style pattern matching is a far more elegant, general, and potentially more efficient solution to the problems with **union**s. At the November 2014 meeting in the University of Illinois at Urbana-Champain, I gave a presentation on design issues related to pattern matching [Solodkyy et al. 2014] partly based on research done with Yuriy Solodkyy and Gabriel Dos Reis in Texas A&M University [Solodkyy et al. 2013]. We had a library implementation that performed comparably to functional programming languages despite lack of integration into the compiler. That library could cope with both closed sets of alternatives (algebraic types) and open sets (class hierarchies). One aim was to eliminate use of the visitor pattern [Gamma et al. 1994]. However, we did not have an acceptable syntax. My purpose for the presentation was to raise interest and set longer-term goals. There was significant interest and after the completion of C++17 work started [Murzin et al. 2019, 2020]. Maybe pattern matching can make it into C++23 (§11.5).

8.4 Concurrency

In C++17, the use of locking became significantly easier by the addition of

- **scoped_lock** – to acquire an arbitrary number of locks without the possibility of deadlock
- **shared_mutex** and **shared_lock** – to implement reader-writer locks

For example, we can acquire multiple locks without fear of deadlock:

```
void f()
{
    scoped_lock lck {mutex1,mutex2,mutex3}; // acquire all three locks
    // ... manipulate shared data ...
} // implicitly release all mutexes
```

C++11 and C++14 failed to give us reader-writer locks. That was obviously a serious omission, caused by the pressure of proposals and the length of time needed to process proposals. C++17 remedied that by adding **shared_mutex**:

```
shared_mutex mx;    // a mutex that can be shared

void reader()
{
    shared_lock lck {mx};   // willing to share access with other readers
    // ... read ...
}

void writer()
{
    unique_lock lck {mx};   // a writer needs exclusive (unique) access
    // ... write ...
}
```

Many readers can "share" the lock (i.e., enter the critical section at the same time) whereas the writer requires exclusive access.

I consider these good examples of the "make simple things simple" philosophy. Sometimes, I wonder – with many C++ programmers – "what took them so long?"

Note how the notation was simplified using template argument deduction from constructor arguments (§8.1).

8.5 Parallel STL

In the longer term, the use of parallel algorithms will be of prime importance, because from a user's point of view, there is nothing simpler than just saying "please do this algorithm." From an implementer's perspective having a specific interface and no serial constraint on the algorithm is an opportunity. C++17 offers only a small start, but that's far better than no start because it sets a direction. Unsurprisingly, there was some opposition in the committee, mostly from people wanting more complex interfaces for expert users. Some people expressed serious doubt that such a simple solution would be viable and argued for delays.

The basic idea is to provide an extra argument to each standard-library algorithm, allowing the user to request vectorization and/or multithreading. For example:

```
sort(par_unseq,begin(v),end(v));  // consider parallelizing and vectorizing
```

This was done only for the STL algorithms, so the important **find_any** and **find_all** algorithms are missing. In the future we will see algorithms specifically designed for parallel use. This is already happening for C++20.

Another weakness is that there was still no standard way of cancelling a thread. For example, having found an item in a search, a thread cannot stop other parallel searches. This is a result of the POSIX intervention against all forms of cancellation (§4.1.2). C++20 offers cooperative cancellation (§9.4).

The parallel algorithms the C++17 support vectorization. This is important because improved support for SIMD is one of the few areas where hardware still (post-2017) delivers dramatic growth in single-thread performance.

In C++20, we can also use the ranges library (§6.3) to (finally) avoid the explicit mention of a container's sequence of elements and just write:

```
sort(v);
```

Unfortunately, the range versions of the parallel algorithms didn't get finished in time for C++20, so we'll have to wait for C++23 before writing:

```
sort(par_unseq,v);  // sort v using parallelism and vectorization
```

Alternatively, we can write our own adapter:

```
template<typename T>
concept execution policy = std::is_execution_policy<T>::value;

void sort(execution_policy auto&& ex, std::random_access_range auto& r)
{
  sort(ex,begin(r),end(r));    // sort with the execution policy ex
}
```

After all, the standard library is extensible.

8.6 File System

In 2002, Beman Dawes wrote the Boost file system library that became one of the most popular Boost libraries [Boost 1998–2020]. In 2014, the Boost file system library [Dawes 2002–2014] was (with modifications) made into a TS [Dawes 2014, 2015] and after further modifications moved into the standard for C++17. Dealing with file names and file systems is tricky because it involves concurrency, many natural languages, and differences among operating systems. It is nice finally to be able to manipulate directories (folders) in a standard manner (as has been done using Boost for 15 years). The key type offered is that of a **path** that abstracts away notions of character sets and filesystem notations. For example:

```
void do_something(const string& name)
{
    path p {name};  // name could be in Russian or Arabic
                    // name could use Linux or Windows file notation
    try {
        if (exists(p)) {
            if (is_regular_file(p))
                cout << p << "regular file, size: " << file_size(p) << '\n';

            else if (is_directory(p)) {
                cout << p << "directory, containing:\n";

                for (auto& x : directory_iterator(p))
                    cout << "    " << x.path() << '\n';
            }
            else
               cout << p << " exists\n";
        }
        else
            cout << p << " does not exist\n";
    }
    catch (const filesystem_error& ex) {
        cerr << ex.what() << '\n';
        throw;
    }
    // ... use p ...
}
```

Catching the exception protects against rare errors, such as someone removing a file after the **exists(p)** check and before the more detailed inquiries. The file system's interfaces offer support for handling both rare (exceptional) and common (expected) errors (§7.2).

8.7 Explicit Tests in Conditions

I consider "many small proposals" a danger, even if each can help someone. Consider the ability to add an explicit test to a condition [Köppe 2016b]:

```
if (auto p = f(y); p->m>0) {
   // ...
}
```

The **p->m>0** is an explicit test, so the meaning is

```
{
    auto p = f(y);
    if (p->m>0) {
       // ...
    }
}
```

It is a generalization of the C++98 combined declaration and test (§2.2.1):

```
if (auto pd = dynamic_cast<Derived*>(pb)) {  // true if pb points to a Derived
  // ...
}
```

The question is whether that generalization is sufficiently obvious and useful to be worthwhile. My answer was "no." However, this is a (not all that uncommon) example of me being voted down.

My opinion is that the explicit test is best expressed within the **if**-statement. There, it is harder to overlook and being conventional has its benefits, especially for people who don't spend all of their time writing C++. On the other hand, the explicit test seems to be popular with people who design their code so that the result of every function needs to be tested for errors - a style I strongly dislike (§7.5).

There are people who aggressively rewrite code to use novel features. I heard of several cases of people who saw this:

```
if (auto p = f(y)) {
   if (p->m>2) {
      // ...
   }
   // ...
}
```

and immediately re-wrote it to this:

```
if (auto p = f(y); p->m>2) {
   // ...
}
```

Claiming elegance and shorter code. Naturally, it crashed when **p==nullptr**, which the original code didn't. Thus, whatever benefits we might get, we introduced a new opportunity for errors and confusion.

For generality, the explicit test was also allowed in **switch** and **while** conditions. For C++20, this facility was extended to include initializations in range-**for** statements [Köppe 2017c].

8.8 Proposals That Didn't Make C++17

In addition to concepts (§6.3.8), a few proposals that I consider important didn't make C++17. The story of C++ would not be complete without a mention of them.

- §6.3.8: concepts (C++20)
- §8.8.1: networking
- §8.8.2: operator dot
- §8.8.3: uniform call syntax
- §8.8.4: default comparison
- §9.3.2: coroutines (C++20)

Static reflection was processed in a study group (§3) and wasn't scheduled for C++17, but it was important work started in this period.

8.8.1 Networking. In 2003, Christopher M. Kohlhoff started developing a library, called *asio*, to support networking [Kohlhoff 2018]:

> *"Asio is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach"*

In 2005, it became part of Boost [Kohlhoff 2005] and in 2006 it was proposed for the standard [Kohlhoff 2006]. In 2018 it became a TS [Wakely 2018]. Despite 13 years of heavy production use, it didn't make it into C++17. Worse, work to get the networking library into C++20 also stalled. This implies that, after 15 years of production use of asio we still have to wait for at least until 2023 for it to become standard. The reason for the delay is that there are still serious discussions about how best to generalize the way concurrency is handled in asio and elsewhere. A proposal for that, called "executors", has broad support and some hoped it would make it into C++20 [Hoberock et al. 2019, 2018]. I consider the lack of executors and networking in C++20 an example of "the best is the enemy of the good."

8.8.2 Operator Dot. The very first proposal for an extension to C++ at the start of the standards process was by Jim Adcock in 1990 to allow the overloading of operator dot (**.**) [Adcock 1990]. We have been able to overload operator arrow (**->**) since 1984 and it is heavily used to implement "smart pointers" (e.g., **shared_ptr**). People wanted (and still want) operator dot to implement smart references (proxies). Basically, people asked for a way of making **x.f()** mean **x.operator.().f()** so that **operator.()** could control access to members. However, discussions tended to deadlock over the issue of whether an overloaded operator should be applied to an implicit use of dot. For example: **++x** for a user-defined type is interpreted as **x.operator++()**. Now, if the user-define type has **operator.()** defined, should **++x** mean **x.operator.().operator++()**? Andrew Koenig and Bjarne Stroustrup tried to resolve that in 1991 [Koenig and Stroustrup 1991a] just to be vehemently opposed by the original proposer, Jim Adcock. Gary Powell, Doug Gregor, and Jaakko Järvi tried again in 2004 for C++0x [Powell et al. 2004], but again the committee deadlocked. Finally, in 2014 Bjarne Stroustrup and Gabriel Dos Reis tried again for C++17 with what I consider a more complete and better reasoned approach [Stroustrup and Dos Reis 2014]. For example:

```
template<class X>
class Ref {  // smart reference (with ownership)
public:
        explicit Ref(int a) :p{new X{a}} {}
        X& operator.() { /* maybe some code here */ return *p; }
        ~Ref() { delete p; }
        void rebind(X* pp) { delete p; p=pp; }
        // ...
```

```
private:
        X* p;
};

Ref<X> x {99};
x.f();                  // means (x.operator.()).f() means  (*x.p).f()
x = X{9};               // means x.operator.() = X{9} means (*x.p)=X{9}
x.rebind(new X{77});    // means that x holds and own that new X
```

The basic idea is that operations defined on "the handle" (here **Ref**) are applied to the handle (e.g., the constructor, destructor, **operator.()**, and **rebind()**) whereas operations that are not defined on "the handle" are applied to "the value," that is, to the result of **operator.()**.

After much work [Stroustrup and Dos Reis 2016], that also failed. The reasons for the failure of the 2014 proposal are interesting. There was of course the usual wording problems and obscure "dark corners" of the design, but I think the proposal would have succeeded had the committee not become so excited by the idea of smart references that mission creep set in and alternative proposals were made by Mathias Gaunard and Dietmar Kühl [Gaunard and Kühl 2015] and Hubert Tong and Faisal Vali [Tong and Vali 2016], respectively. The former required a heavy dose of template metaprogramming from all who wanted to define an **operator.()** while the latter was basically object-oriented, introducing a new form of inheritance and implicit conversions.

Should the action of an **operator.()** depend on the member to be accessed or should **operator.()** be a unary operator depending only on the object it was applied to (just like **operator->()**)? The former was central to Gaunard and Kühl's proposal. Bjarne Stroustrup and Gabriel Dos Reis had also considered making **operator.()** binary but concluded that it was far too complex and that matching operator arrow (->) was important.

In the end, the initial proposal wasn't really rejected (it was approved by EWG, but never brought to a full committee vote), but further progress stalled for lack of new input to gain consensus among the competing proposals. Also, the original proposers (Bjarne Stroustrup and Gabriel Dos Reis) got distracted by even more important proposals, such as concepts (§6) and modules (§9.3.1), and their "day jobs." I consider operator dot a prime example of the members of the committee lacking a shared view of what C++ is and what it should become (§9.1). Thirty years, six proposals, many discussions, much design and implementation work, and we have nothing.

8.8.3 Uniform Call Syntax. The very first discussion of concepts in 2003, mentioned the need for a uniform syntax for function calls [Stroustrup and Dos Reis 2003b]. That is, **x.f(y)** and **f(x,y)** should ideally mean the same. The point is that when writing a generic library, you have to decide whether to call operations on arguments using the object-oriented or the functional notation (**x.f(y)** or **f(x,y)**). As a user, you have to adjust to the choice made by the library designer. Different libraries and different organizations differ on this. For operators, such as **+** and *****, uniform resolution has always been the rule; that is, a use (e.g., **x+y**) would find both member functions and free-standing functions. In the standard library, we have a plague of pairs of functions to deal with this dilemma (e.g., to make both **begin(x)** and **x.begin()** work).

I should have settled that issue in 1985 or so before the committee could tie itself into knots over details and potential problems, but I failed to generalize from the operator case.

In 2014, Herb Sutter and I independently proposed a "uniform function call syntax" [Stroustrup 2014a; Sutter 2014]. The proposals were incompatible, of course, but we immediately solved that by merging them into a joint proposal [Stroustrup and Sutter 2015].

Herb was partly motivated by a wish to support autocompletion in IDEs and was leaning toward the "object-oriented" notation (e.g., **x.f(y)**) whereas I was primarily motivated by generic programming concerns and leaned toward the traditional mathematical notation (e.g., **f(x,y)**).

The first serious objection was, as ever, compatibility; that is, that we might break existing code. The original proposal could indeed break some code by preferring a better match or making a call ambiguous, but we argued that it would be worthwhile and often beneficial. We lost that argument and a revised version worked by the principle that **x.f(y)** would look into the class of **x** first and only if no **f** was found there would it consider **f(x,y)**. Similarly, **f(x,y)** would look into the class of **x** only if no free-standing function was called. That would not make **f(x,y)** and **x.f(y)** completely equivalent but obviously it would break no existing code.

This looked most promising but was met with a howl of outrage: it would mean the end of stable interfaces! The argument, primarily presented by people from Google, was that no interface that depended on overload resolution could be stable because adding a function could change the meaning of existing code. That is of course true. Consider:

```
void print(int);
void print(double);

print('a'); // prints the integer value of 'a'

void print(char); // add a print() to change the overload set

print('a');  // print the character 'a'
```

My answer to that argument was that just about any program can have its meaning changed by quite a few different kinds of added declarations. Also, a common use of overloading is to add functions that provide semantically better resolutions (often, to fix bugs). We have always strongly recommended against adding overloads that (like **print(char)** above) changes the semantics of calls of an overload set in the middle of a program. In other words, this definition of "stable" is unrealistic. I (and others) pointed out that the problem already existed for class members. The answer was basically that the set of class members is closed so that version of the problem is manageable. I observed that by using namespaces, the set of free-standing functions associated with a class can be identified much as members are [Stroustrup 2015b].

By then, much controversy and confusion had erupted and novel proposals started to appear to compete with the ones under discussion. The UK delegation suggested C# style extension methods [Coe and Orr 2015] and several people, notably John Spicer insisted that if we needed a unified function call notation, it should be a new notation separate from the two we already have. I still fail to see how adding a third notation (e.g., **.f(x,y)** as suggested) would unify anything. It would become yet another example of the N+1 problem (§4.2.5).

After the defeat of the proposal, I was asked to reexamine the issue once we had modules (§9.3.1). Then, the reach of the name lookup for a free-standing function could be limited to the module of the class of its first argument. That might resurrect the unified function call proposal, but I don't see how that would address the (IMO vastly overstated) concerns about stability of interfaces.

Again, the lack of a shared view of C++'s role and future blocked progress (§9.1).

In retrospect, I don't think that the object-oriented notation (e.g., **x.f(y)**) should ever have been introduced. The traditional mathematical notation **f(x,y)** is sufficient. As a side benefit, the mathematical notation would naturally have given us multi-methods, thereby saving us from the visitor pattern workaround [Solodkyy et al. 2012].

8.8.4 Default Comparisons. Like C, C++ doesn't offer default comparisons for data structures. For example:

```
struct S {
    char a;
    int b;
};

S s1 = {'a',1};
S s2 = {'a',1};

void text()
{
    S s3 = s1;      // OK, initialization
    s2 = s1;        // OK, assignment

    if (s1==s2) { /* ... */}  // error: == undefined for Ss
}
```

The reason is that, given the usual memory layout of **S**, there will be "unused bits," in the memory holding an **S** so that a naive implementation of **s1==s2** comparing the bits of the words holding **s1** and **s2** might yield **false**. Had it not been for those "unused bits," C would have had at least default equality. I discussed this with Dennis Ritchie back in the early 1980s, but we were both too busy to do anything about it at the time. This problem doesn't occur for copying (e.g., **s1=s2**), where the naive and traditional solution simply copies all the bits.

Allowing assignment but not comparison because of the efficiency of simple implementations was appropriate for the 1970s, but odd for the 2010s. Our optimizers could easily handle that, and I – like many others – were tired of explaining why such comparisons weren't provided. In particular, many STL algorithms require == or < and therefore don't work for simple data structures without the user explicitly defining **operator==()** and/or **operator<()** for them.

In 2014, Oleg Smolsky [Smolsky 2014] proposed a simpler way of defining comparison operators

```
struct Thing {
    int a, b, c;
    std::string d;

    bool operator==(const Thing &) const = default;
    bool operator<(const Thing &) const = default;

    bool operator!=(const Thing &) const = default;

    bool operator>=(const Thing &) const = default;
    bool operator>(const Thing &) const = default;
    bool operator<=(const Thing &) const = default;
};
```

This addresses the right problem, but it is verbose (six long lines to say "I want the default operators") and definitely second best to getting comparison operators by default. There were other technical problems (e.g., "but that solution is intrusive: if I can't modify a class, I can't add comparisons") but the race was now on to better support operators for C++17.

I wrote a paper discussing the problem [Stroustrup 2014c] and proposed to supply the comparisons by default for simple classes [Stroustrup 2014b]. It turned out that defining what it meant for a

class to be simple in this context was hard and Jens Maurer discovered some nasty scope problems with user-defined comparison operators in the presence of default operators (e.g., "what does it mean if we use a default == and then later define an **operator==()** in a different scope?).

Many more papers were written by Oleg, me, and others but the proposals stalled. People started to heap more requirements on the proposal. For example, the performance of default comparisons should equal three-way comparisons for naive uses. Lawrence Crowl wrote an analysis of comparisons in general [Crowl 2015b] touching upon issues such as handling total, weak, and partial orders. The general opinion of the EWG was that Lawrence's analysis was great, but he'd need a time machine to get the mechanisms in place in C++.

Finally, in 2017, Herb Sutter made a proposal (partially based on Lawrence Crowl's work) based on a three-way comparison operator <=> (as found in a variety of languages) from which the usual operators could be generated [Sutter 2017a]. It didn't give us the default operators, but at least it offered a one-line formula for defining it:

```
struct S {
    char a;
    int b;
    friend std::strong_order operator<=>(S,S) = default;
};

S s1 = {'a',1};
S s2 = {'a',1};

bool b0 = s1==s2;     // true
int b1 = s1<=>s2;     // 0
bool b2 = s1<s2;      // false
```

The solution above is recommended by Herb Sutter as causing the fewest problems (e.g., with overloading and scope), but it is intrusive. I can't use it for a class that I cannot modify. In that case, a non-member <=> can be defined:

```
struct S {
    char a;
    int b;
};

std::strong_order operator<=>(S,S) = default;
```

The <=> proposal contained an option for implicitly defining <=> for simple classes, but unsurprisingly people who consider it safer to be explicit about everything voted that down.

So, instead of a facility that made trivial examples work as expected for novices, we got an elaborate facility that allows experts to carefully craft subtle comparisons.

The <=> had an easier pass through the committee than any other recent proposal I can think of. This was the case even though there were no usable implementations and the proposal had strong implications for the standard library. Predictably, this led to many surprises (§9.3.4) including the kind of lookup problems that had help scuttle the earlier proposals for ==. My guess is that the comparison operator discussions had convinced many that something was worth doing and the <=> proposal addressed a variety of issues and fitted with what was familiar from other languages.

Sometime in the future, I will most likely again propose that == and <=> be defined by default for simple classes. The novice and casual users of C++ deserve that simplicity.

Being proposed in 2017, <=> missed C++17, but after much further work, it is in C++20 (§9.3.4).

9 C++20: A STRUGGLE FOR DIRECTION

Design by a 350+ member committee is unlikely to produce a coherent result. People from vastly different backgrounds (including different education) and differing pressures from their "day jobs" naturally differ on direction, priorities, and committee procedures. At a first approximation, for every proposal there are a dozen or so members who strongly object to something. Given that WG21 wants 80% or 90% in favor to declare consensus, it's a amazing that C++ has succeeded so far.

9.1 Design Principles

What does C++ want to be when it grows up? In other words, does WG21 have a clear view of what it is trying to do? I don't think so. Individual members have ideas, but none that are generally accepted and sufficiently concrete to guide individual discussions and decisions.

The ISO C++ standards committee doesn't have a generally accepted set of design criteria or set of criteria for accepting a feature. This is not for lack of trying. I have repeatedly and consistently articulated design criteria:

- The "rules of thumb" from *The Design and Evolution of C++* [Stroustrup 1994] (§2.1) including RAII (§2.2.1), object-oriented programming, generic programming, and static type safety.
- Make simple things simple! (§4.2) leading to the Onion principle (§4.2).
- Direct map to hardware and Zero-overhead abstraction (§1) (§11.2).
- Grow C++ based on feedback to solve real-world problems (§11.2).
- Stay stable and compatible [Koenig and Stroustrup 1991b; Stroustrup 1994].
- Deal directly with hardware, powerful compositional abstraction, and a minimal run-time system (the retrospective in my HOPL3 paper [Stroustrup 2007]).

The problem is that people find it too hard to agree on interpretation and too easy to ignore what they don't like. This allows for fundamental disagreements about what is important to fester. People make design decisions based on what they understand from their education and their day jobs. One problem is the variety of such backgrounds combined with the uneven coverage of the wide field of C++ use in the standards committee (§3.3). Many are simply too certain of their opinions [Stroustrup 2019b]. It is really hard to know exactly what's a fad and what will help the C++ community in the long term. Often, the first solution found is not the best.

It is very easy to get lost in details and lose sight of the bigger picture. It is very easy to focus on current problems and forget about the long term (decades). Conversely, members can get so focused on general principles and the distant future that they neglect urgent practical problems.

In 2017, at the urging of a group of heads of national standards body delegations [van Winkel et al. 2017] to become serious about direction, the WG21 established the *Direction Group* (often called the *DG*) to try to address issues of design aims and direction (§3.2). The DG issued its first extensive statement in 2018 [Dawes et al. 2018] emphasizing adherence to articulated principles, consistency, and encouraging processes to ensure those. For example:

> *"We fundamentally need:*
> - *Stability: Useful code "lives" for decades.*
> - *Evolution: The world changes and C++ must change to face new challenges.*
>
> *There is an inherent tension here."*

The DG emphasizes the need for coherence across the whole standard:

> *" Today, some of the most powerful design techniques combine aspects of traditional object-oriented programming, aspects of generic programming, aspects of functional programming, and some traditional imperative techniques. Such combinations, rather than theoretical purity, are the ideal.*

> • *Provide features that are coherent in style (syntax and semantics) and style of use. This applies to libraries, to language features, and combinations of the two."*

And of course, static types:

> *" C++ relies critically on static type safety for expressiveness, performance, and safety. The ideal is*
> • *Complete type-safety and resource-safety (no memory corruption and no leaks) This is achievable without added overhead, in particular without adding a garbage collector, and without restricting expressibility. "*

Both the NB head request [van Winkel et al. 2017] and the DG document [Dawes et al. 2018] emphasized the need for committee members to know the history of C++ to ensure a degree of continuity. An ahistorical group cannot maintain a coherent view of what they are designing. Thus, the HOPL papers [Stroustrup 1993, 2007] and *The Design and Evolution of C++* [Stroustrup 1994] play an essential role.

Traditionally, in line with the ISO charter for WG21, the work on the evolution of C++ has focused exclusively on language and library issues. However, a developer is not concerned exclusively with the language: a program is the product of a tool chain (§1). Astoundingly, C++ does not have a standard for dynamically linked libraries or a standard build system. The tools study group *SG15* was founded in 2018 to try to grapple with the diverse issues related to tooling (§3.2).

9.2 My C++17 List

As part of my continuing effort to encourage the committee to focus on significant improvements – rather than on just what can be easily done and easily agreed upon – I made a list of what I considered important and feasible for C++17 together with its rationale:

- *Concepts – they allow us to precisely specify our generic programs and address the most vocal complaints about the quality of error messages.*
- *Modules – provided they can demonstrate significant isolation from macros and a significant improvement in compile times.*
- *Ranges and other key STL components using concepts – to improve error messages for mainstream users and improve the precision of the library specification ("STL2").*
- *Uniform call syntax – to simplify the specification and use of template libraries.*
- *Coroutines – should be very fast and simple.*
- *Networking support – based on the asio library as described in the TS.*
- *Contracts – not necessarily used in the C++17 library specification.*
- *SIMD vector and parallel algorithms.*
- *Library vocabulary types, such as* ***optional****,* ***variant****,* ***string_view****, and* ***array_view****.*
- *A "magic type" providing arrays on the stack (****stack_array****) with support for reasonable safe and convenient use.*

In April 2015, I presented that list at an evening session at the WG21 meeting in Lenexa, Kansas, to a sympathetic audience. However, few seemed sufficiently motivated to change the focus of their work to align with that list. The list "leaked" and led to confused discussions on the web, so I had to write it up [Stroustrup 2015a].

In a united committee, everything on that list could be ready for C++17. In reality, I thought it would be feasible to complete about half of the proposals on that list if we focused on them. However, I was far too optimistic. We were only able to get consensus on the standard-library vocabulary types. The **array_view** was renamed **span** and is part of C++20 (§9.3.8).

Fortunately, most items on that list are in C++20. The exceptions are

- the networking library (§8.8.1) – now a TS [Wakely 2018]
- contracts (§9.6.1) – almost made C++20
- uniform function call (§8.8.3)
- the SIMD vector – now in a TS [Hoberock 2019]
- the **stack_array**

This list led to scheduling debates. As the 2016 defeat of the concepts proposal (§6.3.8) seemed inevitable, I was asked – in full committee – whether I would propose a one-or-two-year delay to get concepts in, making the standard C++18 or C++19. I declined because I consider a predictable release cycle more important to the community than any individual improvement. Also, there was no guarantee that a consensus would emerge and one schedule slip would most likely cause further slips. If one proposal is deemed worth delaying for, it will be argued that others are equally worth waiting for. Such logic caused C++0x to become C++11, even as some had hoped for C++06.

9.3 C++20 Features

WG21 set the deadline for new proposals for C++20 to November 2018 and declared a "feature freeze" after the February 2019 meeting. In February 2020, at a meeting in Prague in the Czech Republic, the technical vote was 79-0 with one abstention [Smith 2020]. All 15 national body heads of delegation voted for. The official standard will be issued by the ISO in late 2020. The C++20 features are:

- §6.4: *Concepts* – specification of requirements for generic code
- §9.3.1: *Modules* – support for modularity for code hygiene and improved compile times
- §9.3.2: *Coroutines* – stackless coroutines
- §9.3.3: *Compile-time* computation support
- §9.3.4: <=> – a three-way comparison operator
- §9.3.5: *Ranges* – a library flexible range abstractions
- §9.3.6: *Date* – a library providing date types, calendar, and time zones
- §9.3.8: *Span* – a library providing efficient and safe access to arrays
- §9.3.7: *Format* – a library providing type-safe printf-like output
- §9.4: *Concurrency improvements* – such as scoped threads and stop tokens
- §9.5: *Many minor features* – such as C99-style designated initializers and string literals as template arguments

The following are not ready for C++20 but may become major features of C++23:

- §8.8.1: *Networking* – a networking library (sockets, etc.)
- §9.6.2: *Static reflection* – facilities for generating code based on the surrounding program
- *Pattern Matching* – selecting code to be executed based on types and object values [Murzin et al. 2019]

C++20 offers a set of features reflecting long-standing aims for C++ and addresses some fundamental problems. For example, modules and concepts are mentioned in *The Design and Evolution of C++* [Stroustrup 1994] from 1994 and coroutines were part of "C with Classes" and C++ throughout the 1980s. C++20 will have as major an impact on C++ as C++11 had.

Unfortunately, C++20 doesn't have standard library support for modules and coroutines. That could become a significant problem, but there wasn't time to get that ready in time to ship on time. C++23 should provide that (§4.1.3).

9.3.1 Modules. There is an obvious need for improved modularity in C++ programs. From C, C++ inherited the **#include** mechanism that relies on textually including C++ source from "header files"

containing textual definitions of interfaces. A popular header can be **#include**d hundreds of times in various separately compiled sections of a large program. The fundamental problems are:

- *Lack of code hygiene*: the code in one header can affect the meaning of the code in another **#include**d in the same translation unit so that **#include**s are not order independent. Macros are a major problem here, though not the only one.
- *Separate-compilation inconsistencies*: declarations of the same entity in two translation units can be inconsistent, but not all such errors are caught by the compiler or linker.
- *Excessive compile times*: Compiling an interface from source text is relatively slow. Compiling an interface from source text repeatedly is very slow.

This has been known from "the dawn of time" (e.g., see *The Design and Evolution of C++* [Stroustrup 1994] Chapter 18), but the problems have been steadily growing over the years as more and more information is put into header files (**inline** functions, **constexpr** functions, and especially templates). In the early days of C++, typically 10% of the text came from headers, but currently it is more like 90% or even 99%. Consider:

```
#include<iostream>

int main()
{
    std::cout << "Hello, World\n";
}
```

This canonical program is 70 characters, but after the **#include** it yields 419,909 characters for the compiler to digest. Despite the impressive processing speeds of modern C++ compilers, the modularity problem has become urgent.

Encouraged by the committee (and supported by me) David Vandevoorde produced a series of module designs in the 2000s [Vandevoorde 2007, 2012], but progress was very slow. Finishing C++0x, rather than making progress on modules, was the priority of the committee. David mostly struggled on his own getting little more than moral support. In 2012, Doug Gregor presented a completely different module system design from Apple [Gregor 2012]. That design had been done for C and Objective C in the Clang compiler infrastructure [Team 2014] and relied on extra-linguistic file mapping directives, rather than C++ language constructs. There was also an emphasis on keeping header files unmodified.

In 2014 a group of people from Microsoft led by Gabriel Dos Reis presented a proposal based on their work [Dos Reis et al. 2014]. It was closer in spirit to David Vandevoorde's designs than the Clang/Apple proposal and to a large degree based on work on an optimal graph representation of C++ source code done by Gabriel Dos Reis and Bjarne Stroustrup at Texas A&M University (published and made open-source in 2007 [Dos Reis 2009; Dos Reis and Stroustrup 2009, 2011]).

This set the scene for major progress with modules, but also for a series of clashes between the Apple/Google/Clang approach (and implementation) and the Microsoft approach (and implementation).

A study group was created for modules and after 3 years, it produced a TS primarily based on Gabriel Dos Reis' design [Dos Reis 2018].

In 2017 and again in 2018, suggestions to move the Modules TS into the standard for C++20 was blocked by proposals for different designs from Google [Smith 2018a,b]. A major bone of contention was that in Gabriel Dos Reis' design macros could not be exported. The Google people argued that this was a fatal flaw, whereas Gabriel Dos Reis (and I) considered it essential for modularity [Stroustrup 2018c]:

> *"What do I mean by modularity? Order independence:* ***import X; import Y;*** *should be the same as* ***import Y; import X;*** *In other words, nothing can implicitly "leak" from one module into another. That's one key problem with* ***#include*** *files. Just about anything in an* ***#include*** *can affect any subsequent* ***#include****."*

I consider order independence key to both "code hygiene" and performance. By adhering to this Gabriel Dos Reis's implementation of modules achieved on the order of 10-times compile-time improvements over header files – even when pre-compiled headers were used for the old-style compilations. Approaches that catered to traditional headers and conventional uses of macros struggle to match that because of the need to keep module units in a form that allows macro substitution ("token soup") as opposed to graphs of C++ logical entities.

A series of compromises was crafted to converge on a solution with wide acceptance. The key people in that multi-year effort were Richard Smith (Google) and Gabriel Dos Reis (Microsoft) with GCC's module implementer, Nathan Sidwell (Facebook), and others contributing [Dos Reis and Smith 2018a,b; Smith and Dos Reis 2018]. From mid-2018, most discussions focused on technical details that needed precise specification to ensure portability among implementations [Sidwell 2018; Sidwell and Herring 2019].

Consider a simple example of C++20 modules:

```
export module map_printer;   // we are defining a module

import iostream;       // use iostreams
import containers;     // use my containers
using namespace std;

export                 // make print_map() available to users of map_printer
template<Sequence S>
    requires Printable<Key_type<S>> && Printable<Value_type<S>>
void print_map(const S& m) {
    for (const auto& [key,val] : m)   // break out key and value
        cout << key << " -> " << val << '\n';
}
```

This code fragment defines a module **map_printer** that offers the function **print_map** as its user interface and implements it using facilities imported from modules **iostream** and **containers**. To emphasize the difference from older C++ styles, I use concepts (§6) and structured bindings (§8.2).

Key ideas:

- An **export** directive makes an entity available for **import** into another module.
- An **import** directive makes **export**ed entities from another module accessible for use.
- An **import**ed entity is not implicitly **export**ed.
- An **import** doesn't add entities to a context; it only makes entities accessible (thus, an unused **import** is essentially cost free).

The two last points differ from **#include**s and are essential for modularity and compile-time performance.

That simple example is purely module based; this is the ideal. However, there may be half a trillion lines of C++ deployed and header files and **#include**s will not go away any day soon, possibly not for decades. Several individuals and organizations pointed to the need for mechanisms to allow for transition, for coexistence of header files and modules in programs, and for libraries to offer both

header file and module interfaces to users with code bases of different maturity. Remember that at any given time, there are users relying on 10-year old compilers.

Consider implementing the **map_printer** under the constraint that you cannot modify the **iostream** and **container** headers:

```
export module map_printer;    // we are defining a module

import <iostream>       // use iostream header
import "containers"     // use my containers header
using namespace std;

export                   // make print_map() available to users of map_printer
template<Sequence S>
    requires Printable<Key_type<S>> && Printable<Value_type<S>>
void print_map(const S& m) {
    for (const auto& [key,val] : m)   // break out key and value
        cout << key << " -> " << val << '\n';
}
```

An **import** directive that names a header file works almost exactly as an **#include** would – macros, implementation details, and recursively **#include**d headers. However, the compiler ensures that **import**ed "legacy headers" do not have mutual dependencies. That is, **import**s of headers is order independent, thus providing some, but not nearly all of the benefits of modularity. For example, importing individual headers, such as **import<iostream>**, leaves the programmer with the task of deciding which to import, slows down compilation by having unnecessarily many interactions with the file system, and limits pre-compilation of standard-library components from different headers. Personally, I'd like to see modules with a coarser granularity, e.g., a standard **import std** directive for making the complete standard library available. However, a more ambitious refactoring of the standard library [Clow et al. 2018] had to be postponed until C++23 (§11.5).

Facilities like **import** of headers were an essential part of the Google/Clang proposals. One reason for that is that there are libraries for which the primary interface is a set of macros.

Late in the design/implementation/standardization effort objections focused on the modules likely impact on build systems. Current build systems for C and C++ are heavily optimized for handling header files. Decades of work has been expended on optimizing that and several people associated with traditional build systems expressed doubts that modules could be fitted in without (unaffordable) major redesigns and/or that builds using modules would not allow for parallel compilation (because an importing module depends on the imported module being previously compiled) [Bindels et al. 2018; Lopes et al. 2019; Rivera 2019a]. Fortunately, early impressions had been overly pessimistic [Rivera 2019b], the build2 system had been modified to handle modules, Microsoft and Google reported that their build systems showed good results with modules, and finally Nathan Sidwell reported that he had modified GNU build to handle modules in only two weeks of his spare time [Sidwell 2019]. A final presentation of these experiences and a joint paper of the key module implementers (Gabriel Dos Reis, Nathan Sidwell, Richard Smith, and David Vandevoorde) swayed almost all nay-sayers [Dos Reis et al. 2019].

In February 2019, modules were voted into C++20 by a 46-6 majority including all implementers [Smith 2019]. By then, the major C++ implementations were already close approximations to the C++20 standard. Modules promise to be the most important single improvement offered by C++20.

9.3.2 Coroutines. Coroutines offer a model of cooperative multi-tasking that can be far more efficient than use of threads or processes. Coroutines were an essential part of early C++. Without

the task library offering coroutines, C++ would have been stillborn, but for a variety of reasons coroutines didn't make it into C++98 (§1.1).

The history of what became C++20 coroutines starts with a proposal from Niklas Gustafsson (Microsoft) for "resumable functions" [Gustafsson 2012]. The primary aim was to support asynchronous I/O; "server applications that handle many thousands or millions of clients" [Kohlhoff 2013]. It was equivalent to the async/await facility being introduced into C# at the time (in release 6.0 in 2015). Similar facilities have been introduced into Python, JavaScript, and other languages. Niklas' proposal triggered a competing proposal based on Boost::coroutine from Oliver Kowalke and Nat Goodspeed [Kowalke and Goodspeed 2013] and lots of interest. The **await** design was stackless, asymmetric, and language-based whereas the Boost-derived one used stacks, had symmetric control primitives, and was library-based. A stackless coroutine is one that can only be suspended in its own body and not in a function called from it. That way, a suspension involves only saving a single stackframe ("the coroutine state") and not a whole stack. For performance, this is a massive advantage.

The design space for coroutines is huge, so consensus was hard to achieve. Many in the committee (including me) hoped for a synthesis that gave the best of both approaches, so a group of interested members did an analysis of the alternatives [Goodspeed 2014]. The conclusion was that it might be possible to get the best of both approaches, but that would require serious study. That study took years, yielded no clear results, and in the meantime more proposals emerged.

As for the closely related topic of concurrency (§8.4), a complete explanation of the proposals written, presented, and discussed is beyond the scope of this paper. Here, I present only an outline. There are simply too many complex details for anything else; the papers alone run into many hundreds of pages and many discussions hinge on the performance of (sometimes hypothetical) highly-optimized implementations for advanced use cases. Discussions occurred in SG1 (concurrency), EWG (evolution), LEWG (library evolution), CWG (core language), LWG (library), and even in evening sessions and plenary.

Three ideas emerged repeatedly in these discussions and proposals:

- Represent the state of a coroutine and its operations as a lambda, thus fitting coroutines elegantly into the C++ type system and not requiring some of the "compiler magic" employed by the **await**-style coroutines [Kohlhoff 2013].
- Provide a common interface to both stackless and stackful coroutines – and possibly also to other kinds of concurrency mechanisms, such as threads and fibers. [Kowalke 2015; Riegel 2015].
- To gain optimal performance (runtime and space) for the simplest and most critical uses (generators and pipelines), stackless coroutines need compiler support and must have interfaces that are not compromised by the need to support more advanced use cases [Nishanov 2018, 2019b].

You can't have all three. I'm a great fan of the idea of common interfaces because that minimizes learning efforts and greatly eases experimentation. Similarly, using perfectly normal objects to represent coroutines would open up the whole language to support coroutines. However, in the end the performance argument won out.

In 2017, a proposal from Gor Nishanov based on the **await** stackless approach was accepted as a TS [Nishanov 2017]. The reason this proposal (inevitably nicknamed "Gor-routines") was approved was that its implementation had demonstrated superior performance for its key use cases (pipelines and generators) [Jonathan et al. 2018; Psaropoulos et al. 2017]. The reason that it was made a TS, rather than put into the standard was that many liked the more general (but slower) stackful coroutines and some still hoped for a zero-overhead synthesis of the two approaches. My opinion

at the time (and still today) was that a synthesis was impossible on a reasonable timescale. Having waited almost 30 year to get coroutines back into C++, I didn't want to wait for a breakthrough that might never come: "The best is the enemy of the good."

As usual, naming was a contentious issue. In particular, the draft TS used the keyword **yield**, which was quickly determined to be a popular identifier (e.g., in finance and agriculture). Also, the result produced by a coroutine needs to be wrapped into a structure that a caller can wait on (e.g., a **future** (§4.1.3)) so the semantics of a coroutine **return**-statement is not identical to that of an ordinary **return**-statement. Consequently, some people objected to the "re-use" of **return**. In response, the Evolution Working Group introduced the keywords **co_return**, **co_yield**, and **co_await** for the three key operations in a coroutine. The underscores were introduced to stop native English speakers from misreading **coreturn**, **coyield**, and **coawait** as **core-turn**, **coy-ield**, and **coa-wait**. Making **yield**, and **await** context-sensitive keywords was explored, but that didn't gain consensus. These new keywords aren't pretty and they quickly became a rallying point for people who disliked the TS coroutines for any reason.

In 2018, the TS coroutines were proposed to be put into the standard for C++20, but in the very last minute, Geoff Romer, James Dennett, and Chandler Carruth from Google proposed a rather novice-unfriendly proposal [Romer et al. 2018]. Like Gor's proposal, the Google proposal, named "Core Coroutines", required library support to make the basic mechanisms friendly to non-expert users. This library had yet to be designed. Core Coroutines were claimed to be more efficient than the TS coroutines and addressed a use case that Google had for non-exception-based error propagation. It was based on the idea of representing the state of a coroutine as a lambda. To avoid the widely despised keywords **co_return**, **co_yeld**, and **co_await**, the Core Coroutines offered the supposedly friendlier operators **[->]** and **[<-]**. Surprisingly for an operator, **[->]** was four characters long and took four operands; the "**[**" and "**]**" were part of the token. Unfortunately, the Core coroutines were not implemented so claims of usability and efficiency could not be verified. This delayed further decisions about coroutines.

One significant and potentially fatal problem with the TS coroutines was that they relied on free store (dynamic memory, heap) allocation. That's a significant overhead in some applications. Worse, for many critical real-time and embedded applications, free store use is simply not allowed because of its potential for unpredictable response time and the possibility of memory fragmentation. The Core coroutines didn't have this problem. However, Gor Nishanov and Richard Smith demonstrated that the TS coroutines could guarantee the absence of free store use for almost all uses (and detect and prevent the rest) in one of several ways [Smith and Nishanov 2018]. In particular, it is possible to optimize free store use into stack allocation ("Halo optimization") for almost all critical use cases.

The Core Coroutines evolved and improved over time [Romer et al. 2019a], but a complete implementation never emerged. In 2018, the Bulgarian National Body objected to the TS coroutine design [Mihaylov and Vassilev 2018] and proposed yet another design [Mihaylov and Vassilev 2019]. Again, elegance, generality, and performance were claimed, but again, no implementation existed.

At this point, the head of the Evolution group, Ville Voutilainen, requested that the authors of the three still-active proposals write two evaluation and comparison papers:

- *Coroutines: Use-cases and Trade-offs* [Romer et al. 2019b].
- *Coroutines: Language and Implementation Impact* [Smith et al. 2019].

All three proposals (Gor, Google, and Bulgaria) were stackless and use cases requiring stacks were left for future proposals. All had a bewildering number of customization points [Nishanov 2018], which were deemed essential by their implementers and expert users. It emerged that the

expression of key use cases was not dramatically different in the different proposals. Consequently, the differences could be dismissed as largely cosmetic. For example, is **co_await** uglier than **[<-]**?

This left performance. Gor's proposal had the clear advantage of four years of production use and being implemented in both the Microsoft and Clang compilers. Over the last few meetings before the key votes for C++20, the committee heard experience reports from people from Sandia [Hollman 2019], Microsoft [Jonathan et al. 2018], and Facebook [Howes et al. 2018] and considered some suggestions about improvements and simplifications based on such uses [Baker 2019]. However, the point that (as far as I can judge) swayed the committee to a solid 48-4 vote in favor was that a fundamental flaw was discovered in the strategy of using "ordinary lambdas" to represent the state of a coroutine. For a lambda representing a coroutine state to be just like other lambdas, its size must be known in the first stage of compilation. Only then can we allocate coroutine states on the stack, copy them, move them around, and use them in the various ways the language allows. However, the size of a stack frame (and that's what the state of a stackless coroutine fundamentally is) isn't known until the optimizer has run. There is no information path from the optimizer back to the early stages of the compiler. An optimizer can decrease the size of a frame by eliminating variables and increase it by adding useful temporaries. Thus, a lambda represented a coroutine state from cannot be "ordinary."

Finally, consider a trivial example of a C++20 coroutine:

```
generator<int> fibonacci() // generate 0,1,1,2,3,5,8,13 ...
{
    int a = 0;    // initial values
    int b = 1;

    while (true) {
        int next = a+b;
        co_yield a;    // return next Fibonacci number
        a = b;         // update values
        b = next;
    }
}

int main()
{
    for (auto v: fibonacci())
        cout << v << '\n';
}
```

The use of **co_yield** makes **fibonnacci()** a coroutine. The **generator<int>** return value will hold the next **int** generated and the minimal state needed for **fibonacci()** to wait for the next call. For asynchronous use, we'd use a **future<int>** instead of **generator<int>**. The standard library support for coroutine return types is still incomplete, but mature libraries are in production use.

Could the committee have handled coroutines better? Probably; the C++20 coroutines are remarkably similar to Niklas Gustafsson's 2012 proposal. It was good that we explored alternatives, but did we really need 7 years? Could the massive efforts by many competent people have been more collaborative and less competitive? I feel that better scholarship would have helped in the early stages. After all, coroutines have about 60 years of history, e.g., [Conway 1963]. People knew modern approaches in C++ and related languages, but our understanding wasn't shared or systematic. Had we spent a few months or a year on a thorough review of fundamental design choices, implementation techniques, key use cases, and the literature, I suspect we could have

reached the conclusions we arrived at in February 2019 as early as 2014. The years afterwards could then have been spent on incremental improvements and further functionality for our chosen fundamental approaches.

Much of the credit for the progress made and the final success go to Gor Nishanov. Without his tenacity and his solid implementation work (he did both the Microsoft and the Clang implementations), we would not have the C++20 coroutines. Perseverance is a key factor in success in the committee.

9.3.3 Compile-Time Computation Support. The importance of compile-time evaluation has been steadily increasing in C++ over the years. The STL relied critically on compile-time dispatch [Stroustrup 2007] and template metaprogramming was mostly aimed at moving computation from run time to compile time (§10.5.2). Even the reliance on overloading and the use of virtual function tables in early C++ can be seen as gaining performance by moving computation from run time to compile time. Thus, compile-time computation has always been a key part of C++.

From C, C++ inherited constant expressions that were restricted to integers and could not call functions. Macros were needed for anything nontrivial. This did not scale well. Once templates were introduced and template metaprogramming was discovered, template metaprogramming became widely used to compute values and types at compile time (§10.5.2). In 2010, Gabriel Dos Reis and Bjarne Stroustrup published a paper that pointed out that compile-time computation of values could (and should) be expressed just like other computations, relying on the usual rules for expressions and functions, including the use of user-defined types [Dos Reis and Stroustrup 2010]. This became **constexpr** functions in C++11 (§4.2.7) and the basis for modern compile-time programming. C++14 generalized **constexpr** functions (§5.5) and C++20 added several related features:

- **consteval** – a **constexpr** function guaranteed to be compile-time evaluated [Smith et al. 2018a]
- **constinit** – a declaration modifier to guarantee compile-time initialization [Fiselier 2019]
- the use of matching pairs of **new** and **delete** in **constexpr** functions [Dimov et al. 2019]
- **constexpr string** and **constexpr vector** [Dionne 2018]
- the use of **virtual** functions [Dimov and Vassilev 2018]
- the use of **union**s, exceptions, **dynamic_cast**, and **typeid** [Dionne and Vandevoorde 2018]
- user-defined types as value template arguments – finally allowing user-defined types wherever built-in types are [Maurer 2012]
- **is_constant_evaluated()** predicate – to enable library implementers to optimize code with far fewer platform dependent intrinsics [Smith et al. 2018b]

Along with this effort, the standard library is being made more friendly to compile-time evaluation.

The ultimate aim for much of this effort is to support static reflection for C++23 or later (§9.6.2). The use of user-defined types as template argument types and the use of strings as template arguments were anticipated when I first designed templates, but then they were beyond my ability to design and implement properly.

There are people who would like *every* C++ construct to be available at compile time. In particular, they would like to be able to use the complete standard-library in **constexpr** functions. That may be too much of a good thing. For example, do you really need threads at compile time? Yes, it is feasible. Not making all functions usable at compile time, leaves us with the problem of deciding which should be usable and which should not. So far, the answer is a bit ad hoc and not coherent. Further refinement is needed.

To make a language construct or library component **constexpr**, it has to be very precisely specified and stripped of the possibility of undefined behavior. Thus, the push for compile-time evaluation has been a major driver of more precise specification, for analysis of platform dependencies, and examination of the roots of undefined behavior.

Obviously, this push for compile-time computation creates more work for compilers. The increase in the amount of information in interfaces to allow the compiler to do all that work is being addressed through modules (§9.3.1). Compilers also compensate by caching results and systems relying on parallel builds are common. However, C++ programmers must learn to limit their use of compile-time computation and metaprogramming to code where the compactness and run-time performance makes it worthwhile.

9.3.4 <=>. See (§8.8.4). Immediately after the "spaceship operator" (<=>) was voted in for C++20, it became clear that serious further work was needed on both the language rules and its integration into the standard library. Out of over-enthusiasm and a desire to solve the thorny problems related to comparisons, the committee had fallen victim to the law of unintended consequences. Some committee members (including me) worried that the introduction of <=> had been too hurried. However, by the time our worries became concrete, work had been done assuming the availability of <=>. Also, many committee members and others in the wider C++ community had become excited by the possible performance advantages of a three-way comparison. It therefore came as an unpleasant surprise to find that <=> caused significant inefficiencies in important cases. Given <=> for a type, == was generated from <=>. For strings, == is usually optimized by first comparing sizes: if the number of characters differ, the strings are not equal. A == generated from <=> must read enough of the strings to determine their lexicographical order and that can be far more expensive. After long discussions, it was decided not to generate == from <=> . That and several other corrections [Crowl 2018; Revzin 2018, 2019; Smith 2018c] solved the problems in hand, but compromised the fundamental promise of <=>: that all comparison operators could be generated from a single, simple line of code. Also, thanks to <=>, == and < now have many rules that differ from the rules for other operators (e.g., == is assumed to be symmetric). For better and worse, most rules related to operator overloading have <=> as a special case.

9.3.5 Ranges. The *Ranges library* started as the work of Eric Niebler to generalize and modernize the STL notion of a sequence [Niebler et al. 2014]. It provides standard-library algorithms that are simpler to use, more general, and better performing . For example, the C++20 standard library offers the long-awaited simpler notation for operations on whole containers:

```
void test(vector<string>& vs)
{
    sort(vs);   // rather than sort(vs.begin(),vs.end())
}
```

The original STL as adopted for C++98 [Stroustrup 1993] defined a sequence as a pair of iterators. That left out two important ways of specifying a sequence. The range library provides the three major alternatives (now called *ranges*):

- **(first,one_beyond_last)** for when we know where the beginning and the end of the sequence are (e.g., "sort from the beginning to the end of a vector").
- **(first,number_of_elements)** for when we don't actually need to have the end of a sequence computed (e.g., "look at the first 10 elements of a list").
- **(first,termination_criteria)** for when we use a predicate (e.g., a sentinel) to define the end of the sequence (e.g., "read until end of input").

A **range** is a **concept** (§6). All C++20 standard-library algorithms are now precisely specified using concepts. This is in itself a major improvement and it is what enables the generalization to use ranges, rather than just iterators. That generalization allows for pipelining of algorithms:

```
vector<int> vec = {1, 2, 3, 4, 5, 6, 7 ,8, 9, 10 };

auto even = [](int i){ return i%2 == 0; }

for (int i : vec | view::filter(even)
                 | view::transform( [](int i) { return i*i; } )
                 | view::take(5))
    cout << i << '\n';    // print the squares of the first 5 even integers
```

Like in Unix, the pipeline operator, |, delivers the output of its left-hand operand as the input to its right-hand operand (e.g., **A**|**B** means **B(A)**. This will get much more interesting once people start using coroutines (§9.3.2) to write pipeline filters.

In 2017, the ranges library became a TS [Niebler and Carter 2017] and February 2019 it was voted into C++20 [Niebler et al. 2018].

9.3.6 Dates and Time Zones. The *date library* is the work of Howard Hinnant (Ripple, formerly Apple) providing standard calendar and time zone support for C++ [Hinnant and Kamiński 2018]. It is based on the **chrono** standard-library time support. Howard was also the main person behind **chrono** (§4.6). The date library is the result of years of work and real-world use. In 2018, it was voted into C++20 and placed in <**chrono**> together with the older time utilities.

Consider how to express a point in time (a **time_point**):

```
constexpr auto tp = 2016y/May/29d + 7h + 30min + 6s + 153ms;
cout << tp << '\n';    // 2016-05-29 07:30:06.153
```

The notation is conventional (using user-define literals (§4.2.8)) and a date is represented as a **year**,**month**,**day** structure. However, when needed, a date maps to a point on the standard time line (**system_time**) at compile time (using **constexpr** functions (§4.2.7)), so it is blindingly fast and can be used in constant expressions. For example:

```
static_assert(2016y/May/29==Thursday);  // check at compile time
```

By default, the time zone is UTC (aka Unix time), but conversion to different time zones is easy:

```
zoned_time zt = {"Asia/Tokyo", tp};
cout << zt << '\n';          // 2016-05-29 16:30:06.153 JST
```

The date library can also handle days of the week (e.g., **Monday** and **Friday**), multiple calendars (e.g., Gregorian and Julian), and more esoteric (but necessary) notions, such as leap seconds.

In addition to being useful and fast, the date library is interesting because it offers very fine-grained static type checking. Common mistakes are caught at compile time. For example:

```
auto d1 = 2019y/5/4;     // error: May 4 or April 12?
auto d2 = 2019y/May/4;   // OK
auto d2 = May/4/2019;    // OK (the day follows the month)
auto d3 = d2+10;         // error: add 10 days, 10 months, or 10 years?
```

The date library is a rare example of a standard-library component that directly addresses an application domain, rather than "just" offering a supporting "computer science" abstraction. I hope to see more such in future standards.

9.3.7 Format. The iostream library offers type-safe extensible I/O, but it's formatting facilities are weak. In addition, some people dislike the use of << to separate output values. The format library provides a **printf**-like way of composing strings and formatting values that is typesafe, fast, and works with iostreams. It is primarily the work of Victor Zverovich [Zverovich 2019].

Types that have a << operator can be output within a format string:

```
string s = "foo";
cout << format("The string '{}' has {} characters",s,s.size());
```

This outputs **The string 'foo' has 3 characters**.

This is a variant of the "typesafe **printf**" variadic template idea (§4.3.2). The **{}** simply indicates that a default representation of the value of an argument is to be inserted.

The argument values can be used in any order:

```
// s before s.size():
cout << format("The string '{0}' has {1} characters",s,s.size());
// s.size() before s:
cout << format("The string '{1}' has {0} characters",s.size(),s);
```

Like **printf()**, **format()** offers a whole little programming language for expressing formatting details, such as field width, floating-point precision, integer number base, and alignment in a field. Unlike **printf()**, **format()** is extensible and can handle user-defined types. Here is an example printing a **date** from the **<chrono>** library (§9.3.6) [Zverovich et al. 2019]:

```
string s1 = format("{}", birthday);
string s2 = format("{0:>15%Y-%m-%d}", birthday);
```

The year-month-day format is the default. The **>15** means use 15 characters and left-align the text. The date library contains yet-another small formatting language for use with **format()**. It can even handle time zones and locales:

```
std::format(std::locale{"fi_FI"}, "{}", zt);
```

This will give us the local time in Finland. By default, formatting is not locale dependent, but you can opt in. This significantly improves performance compared to traditional iostream performance when you don't need locale information.

There is no equivalent to **format** for input (**istream**).

9.3.8 Span. Out-of-range access, sometimes referred to as buffer overflow, has been a serious problem from the early days of C. Consider:

```
void f(int* p, int n)   // what is n?
{
   for (int i=0; i<n; ++i)
      p[i] = 7;     // OK?
}
```

How would a tool, e.g., a compiler, know that **n** was meant to be the number of elements in the array pointed to? How can a programmer get it consistently right in a large program?

```
int x = 100;
int a[100];
f(a,x);     // OK
f(a,x/2);   // OK: first half of a
f(a,x+1);   // disaster!
```

For decades, that "disaster" comment has been accurate and range errors have been the root of major security problems. Compilers do not catch range-errors and run-time checking of all subscripting has generally been deemed too expensive for production code.

The obvious solution is to supply an abstraction that holds a pointer plus a size. For example, in 1990, Dennis Ritchie proposed that to the C standard committee: "'fat pointers' whose representation will include the space to store the adjustable bounds at run-time" [Ritchie 1990]. For various reasons, the C committee didn't approve. At the time, I heard the priceless comment: "Dennis isn't a C expert; he never comes to the meetings." It is probably good that I don't remember who said that.

In 2015, Neil MacIntosh (then at Microsoft), revived that idea for the C++ Core Guidelines (§10.6) where we needed a mechanism to encourage and optionally enforce effective programming styles. This **span<T>** class template was put into the Core Guidelines support library and promptly ported to the Microsoft, Clang, and GCC C++ compilers. In 2018, it was voted into C++20.

Using **span**, that example can now be written:

```
void f(span<int> a)   // span holds a pointer and a size
{
   for (int& x : a)
      x = 7;   // OK
}
```

The range-**for** extracts the range from the span and traverses exactly the right number of elements (without costly range checking). This is an example of how an appropriate abstraction can simultaneously simplify notation and improve performance. It is easier and cheaper for an algorithm to explicitly use a range (here, **span**) than to check each individual element access.

If necessary, you can explicitly specify a size (e.g., to operate on a subrange), but then the risk is yours and the notation stands out as a warning:

```
int x = 100;
int a[100];
f(a);        // template argument deduction: f(span<int>{a,100})
f({a,x/2});  // OK: first half of a
f({a,x+1});  // disaster
```

Naturally, simple element access can also be done, e.g., **a[7]=9**, and can be checked at run time. Range checking of **span** is the default in the C++ Core Guideline Support Library (GSL).

The most contentious part of getting **span** into C++20 turned out to be the type of subscripts and sizes. The Core Guidelines **span::size()** was defined to return a signed **int** as opposed to the **unsigned** used by the standard-library containers. Similarly, subscripts were **int**s like for arrays, rather than **unsigned** as they are for standard-library containers. This led to a revival of an old and contentious issue:

- Some consider it obvious that subscripts, being non-negative, should be represented as **unsigned**.
- Some consider consistency with standard-library containers more important than any arguments about past mistakes related to **unsigned**.
- Some consider the use of **unsigned** to represent non-negative numbers misguided (giving a false sense of safety) and a significant source of errors.

Over the strenuous objections of the original designers (including me) and implementers of **span**, the second group won the votes, enthusiastically supported by the first group, so **std::span** has **unsigned** sizes and subscripts. I consider that a sad failure to take advantage of a rare opportunity

to remedy a nasty old mistake [Stroustrup 2018e]. Somewhat predictably and not irrationally, the committee chose bug compatibility over the work of removing a significant bug source.

But what could be wrong about representing subscripts as **unsigned**? This appears to be a rather emotional topic. I received several hate mails about this. There are two fundamental problems:

- **unsigned** does not model natural numbers: it has modular arithmetic and subtraction. For example, if **ch** is an **unsigned char**, **ch+100** will never overflow.
- **int** and **unsigned** convert to each other at the slightest provocation, turning negative values into huge signed values and vice versa. For example, **-2<2u** is false; **2u** is **unsigned**, so **-2** is converted into a huge positive integer before the comparison.

Here is an infinite loop occasionally seen "in the wild":

```
for (size_t i = n-1; i >= 0; --i) { /* ... */ }  // "reverse loop"
```

Unfortunately, the standard-library type **size_t** is **unsigned** and then obviously always **>=0**.

Basically, the rules for conversions among signed and unsigned types, as inherited by C++ from C, have been a major source of hard-to-find errors for decades, but it is hard to convince a committee to address old problems.

9.4 Concurrency

Despite valiant efforts and an emerging broad consensus, the hoped-for general concurrency model ("executors") wasn't ready for C++20 (§8.8.1). This was not for lack of effort, including a special two-day September 2018 meeting in Bellevue WA attended by about 25 people including representative from NVIDIA, Facebook, and the US National Labs. However, several less dramatic useful improvements were completed in time, including

- **jthread** and stop tokens [Josuttis et al. 2019a]
- **atomic<shared_ptr<T>>** [Sutter 2017b]
- classic semaphores [Lelbach et al. 2019]
- barriers and latches [Lelbach et al. 2019]
- minor memory model repairs and improvements to [Meredith and Sutter 2017]

The **jthread** (short for "joining thread") is a thread that obeys RAII; that is, it's destructor joins, rather than terminates, if the **jthread** goes out of scope:

```
void some_fct()
{
    thread t1;
    jthread t2;
    // ...
}
```

At the end of scope, **t1**'s destructor terminates the program unless **t1**'s task has completed, **join**ed, or **detach**ed, whereas **t2**'s destructor will wait for its task to complete.

From the start (pre-C++11), many (incl. me) had wanted **thread** to have what is now **jthread**'s behavior, but people grounded in traditional operating systems threads insisted that terminating a program was far preferable to a deadlock. In 2012 and 2013, Herb Sutter proposed a joining thread [Sutter 2012, 2013a]. This led to much discussion, but no decisions. In 2016, Ville Voutilainen summarized the issues and conducted votes for including joining threads into C++17 [Voutilainen 2016a]. The votes were so massively in favor that I (only partly joking) suggested that we could submit joining threads to C++14 as a bugfix, but somehow, the progress again stalled. In 2017, Nico Josuttis, re-raised the issue and eventually – after eight revisions and the addition of stop tokens – the proposal made it into C++20 [Josuttis et al. 2019a].

"Stop tokens" is a solution to the old problem of how to stop a thread when we are no longer interested in its result. The basic idea is to make thread cancellation (§4.1.2) cooperative. If I want a **jthread** to terminate, I set its stop token. It is the thread's obligation to occasionally test whether the stop token has been set and if so, clean up and exit. This technique is as old as the mountains and works nicely and efficiently for just about every thread that has a main loop where the test of the stop token can be placed.

As ever, naming became an issue: **safe_thread**, **ithread** ('**i**' for interruptible), **raii_thread**, **joining_thread**, and finally **jthread**. The Guideline Support Library calls it **gsl::thread**. Really, the proper name is **thread**, but unfortunately, that name was already taken for a less useful kind of thread.

9.5 Minor Features

C++20 offers many minor new features, such as:

- C99-style designated initializers [Shen et al. 2016]
- Refinements to lambda capture [Köppe 2017b]
- Template parameter lists for generic lambdas [Dionne 2017]
- Initialization of an additional variable within a range-**for** (§8.7)
- Lambdas in unevaluated contexts [Dionne 2016]
- Pack expansions in lambda capture [Revzin 2017]
- Removing the need for **typename** in some cases [Vandevoorde 2017]
- More attributes: **[[likely]]** and **[[unlikely]]** [Trychta 2016]
- **source_location** to give the source code location of a piece of code without the use of macros [Douglas and Jabot 2019]
- Feature test macros [Voutilainen and Wakely 2018]
- Conditional **explicit** [Revzin and Lavavej 2018]
- Signed integers are guaranteed to be two's complement [Bastien 2018]
- Mathematical constants, such as **pi** and **sqrt2** [Minkovsky and McFarlane 2019]
- Operations on bits, such as rotations and counting ones [Maurer 2019]

Some are improvements, but I worry that the sheer volume of obscure novelties does harm [Stroustrup 2018d]. They make the language harder to learn and code harder to read for non-experts. I opposed several features as likely to do as much harm as good (e.g., designated initializers will be used where constructors would lead to more maintainable code). Many are special-purpose and some "expert-only." However, there are people who have a hard time understanding that a feature that can do some good for someone, can be a net liability for C++. Small features that increase generality and uniformity of notation and semantics are of course always welcome.

From a standardization point of view, even the smallest feature takes time to process, document, and implement. That time cannot be used for something else.

9.6 Work in Progress

Naturally, much work is going on aimed at releases beyond C++20 and some work aimed at C++20 didn't complete in time. Notably:

- §8.8.1: Networking and executors – delayed yet again.
- §9.6.1: Contracts – assertions, preconditions, and postconditions; aimed for C++20, but delayed.
- §9.6.2: Reflection – injecting code into a program based on the code being compiled; aimed for C++23.

In addition, the working group and study group pipelines are not empty (§3.2) [Stroustrup 2018d].

9.6.1 Contracts. Contracts are special in that not only did most people expect them to make it into C++20, but contracts were voted into the working paper for C++20 just to be yanked out at the very last moment. A new Study Group, SG-21 chaired by John Spicer, has been created to try to get some form of contracts back for C++23 or C++26. The story of contracts for C++20 is sad but possibly enlightening.

Contracts in various guises have a long history in C++ and other languages. I remember being very attracted to Peter Naur's invariants [Naur 1966] when I first encountered them in the early 1970s. A system of assertions, called A++, was considered for C++ in the early 1990s, but was considered too extensive to be practical. In the late 1980s, Bertrand Meyer popularized the notion of "contracts" with Eiffel [Meyer 1994]. As part of the C++0x effort, a couple of proposals [Crowl and Ottosen 2006] received serious attention in the committee but eventually failed, mostly because of perceived excess complexity and inelegant notation.

For years, Bloomberg (the New York City financial information company) had used a system of run-time assertions, called contracts, to catch problems in their code. In 2013, John Lakos from Bloomberg proposed that system for standardization [Lakos and Zakharov 2013]. It was well received, but it ran into two problems:

- It was based on macros
- It was strictly assertions in implementation code, rather than something that enhanced interfaces.

Revisions followed, but no consensus emerged. To try to break the deadlock a group of people from Microsoft, Facebook, Google, and University Carlos III in Madrid proposed a system of "simple contracts" that did not use macros and added support for pre-conditions and post-conditions (as had the C++0x attempts) [García et al. 2015]. Like the proposal from Bloomberg, this proposal was backed by many years of large-scale industrial use, but had an emphasis on the use of contracts in static analysis. J-Daniel Garcia (University Carlos III) worked hard to produce a design that would satisfy all, but that proposal also ran into opposition.

After many meetings, papers, and (occasionally heated) discussions, it became clear that a compromise was elusive. The two groups then asked me to coordinate and to prove my conjecture that the discussions were focused on "details" and that there had to be a minimal proposal that included the essentials of both groups stated needs and no controversial "details." After a fair bit of work where I alternately met with representatives of the two groups, we finally produced a joint proposal coauthored by "all parties" [Dos Reis et al. 2016a]. I think that design was technically sound rather than a political compromise. It aimed to serve three needs (in order of importance):

- Systematic and controlled run-time tests
- Information for static analyzers
- Information for optimizers

After further work led by J-Daniel Garcia, the proposal was adopted for C++20 in June 2018 [Dos Reis et al. 2018].

To avoid introducing new keywords, we used the attribute syntax. For example, **[[assert: x+y>0]]**. A contract has no effect on a valid program, so this fits the original conception of attributes (§4.2.10).

There are three kinds of contracts:

- **assert** – assertions in executable code
- **expects** – pre-conditions on function declaration
- **ensures** – post-conditions on function declarations

There are three levels of contract checking:

- **audit** – for "expensive" predicates checked only in some "debug mode"
- **default** – for "cheap" predicates that you could plausibly check even in production code
- **axiom** – for predicates intended primarily for static analyzers and never checked at run time.

Upon a contract violation a (possibly user-installed) violation handler is executed. The default action is immediate termination.

I found one aspect interesting: There is a build-mode that allows a program to continue after a contract failure. My first reaction to that was "but that's insane! Contracts are intended to prevent programs with contract violations from running." That was by far the most common reaction. However, John Lakos insisted, based on experience with Bloomberg code, that when you add contracts to a large old code base, you invariably get violations:

- Some code will violate a contract without actually doing anything it protects against.
- Some new contracts will contain errors.
- Some new contracts will have unintended effects.

With continuation, you can use a violation handler to log violations and continue. That way, you can detect many violations in a single run and also leave contracts active in supposedly correct old code. This was accepted to be critical for gradual adoption of contracts.

We saw no sufficient reasons to add class invariants, to allow weakening pre-conditions for overriding functions, or to allow strengthening post-conditions for overriding functions. The emphasis was on simplicity. The ideal was to provide a minimal initial design for C++20 and then add to it later if needed.

This design was implemented by J-Daniel Garcia and in June 2018 it was voted into the committee's working paper for C++20. As usual, a few bugs remained in the specification but we were confident that those could be dealt with in the two years left before the release of the final standard. For example, it was discovered that the working paper text allowed a compiler to optimize based on all contracts (checked or not). That was unintended. It made sense from the point of view that all contracts are valid in a correct program, but disastrous for a program with contracts specifically written to catch "impossible errors." Consider:

```
[[assert: p!=nullptr]]
p->m = 7;
```

If **p==nullptr**, **p->m** is undefined behavior. A compiler is allowed to assume that undefined behavior doesn't happen; it can therefore optimize away code that leads to the undefined behavior. The results of that can be very surprising. In this case, if the execution might continue after a contract violation, the compiler would be allowed to assume that **p->m** is valid so that **p!=nullptr**; then, it could eliminate the contract's check of **p==nullptr**. This ("time-travel optimization") is of course contrary of the aims of contracts and several proposals to remedy that were submitted in a timely manner [Garcia 2018; Stroustrup 2019c; Voutilainen 2019a].

In October 2018, after the deadline for new proposals for C++20, a team from Bloomberg led by John Lakos, including Hyman Rosen and Joshua Berne presented a stream of proposals for redesigns [Berne et al. 2018; Berne and Lakos 2018a,b; Lakos 2018]. The date of the feature freeze (the last day to consider new proposals) had been fixed by a full plenary committee vote. The proposals were based on a scheme to specify the behavior of a contract in the contract itself. For example, **[[assert check_maybe_continue: x>0]]** and **[[assert assume: p!=nullptr]]**.

Instead of using build modes to control the behavior of all contracts (e.g., enable all default contracts or turn off all contract-based run-time checking), you would modify the source code of

individual contracts. In this, these new schemes differed radically from the agreed-upon design in the working paper. Consider:

```
[[assert assume: p!=nullptr]]
```

This would bring back the macro-based scheme that had been rejected in 2014 because the obvious way of managing code changes would involve macros in the code. For example:

```
[[assert MODE1: p!=nullptr]]
```

Here, **MODE1** could be **#define**d as one of the supported alternatives such as **assume** and **default**. Alternatively and roughly equivalently, the meaning of qualifiers such as **assume** could be defined by assignments on the command line (acting like command-line macros).

In essence, the combination of the possibility of continuation after a contract violation and programmer control of the meaning of a contract, would turn the contract mechanism from a system of assertions to a novel control-flow mechanism.

Some proposals even suggested to abolish support for static analysis. There were dozens of variations of these proposals, all late and none capable of increasing consensus.

The flood of novel proposals (from the Bloomberg team and others, e.g., [Berne 2019; Berne and Lakos 2019; Khlebnikov and Lakos 2019; Lakos 2019; Rosen et al. 2019]) and the hundreds of email messages discussing them blocked the needed discussion of bug fixes to the status quo design in the working paper. As I had repeatedly warned (e.g., in [Stroustrup 2019c]), the result of the proposals for a redesign was that contracts were removed from C++20 after a proposal to do so by Nico Josuttis [Josuttis et al. 2019b]. I consider the last year of discussions about contracts a classic case of nobody getting anything because someone wanted it exactly their way. Time will tell if the new study group, SG21, can deliver something more widely acceptable for C++23 or C++26.

9.6.2 Static Reflection. In 2013, a study group for "reflection" (SG-7) was formed and a call for ideas was issued [Snyder and Carruth 2013]. There was broad agreement that C++ needed a static reflection mechanism. That is, we needed a way to write code that examined the program of which it was part and to inject code based on that analysis into that program. That way, we could replace tedious and tricky boilerplate, macros, and extra-linguistic generators with clean code. For example, we could generate functions for stream I/O, logging, comparison, marshalling for storage and networking, building and using object maps, "stringification" of enumerators, test support, and more [Chochlík et al. 2017; Stroustrup 2018g]. The aim of the reflection study group was to get something ready for C++20 or C++23; it was understood that C++17 was not a realistic target.

It was agreed that reflection/introspection relying on run-time traversal of an ever-present data structure was unsuitable for C++ because of the size of such data, the complexity of a complete representation of the language constructs, and the run-time cost of traversal.

Several proposals promptly emerged [Chochlík 2014; Silva and Auresco 2014; Tomazos and Spertus 2014] and over the following years the study group, chaired by Chandler Carruth, held several meetings to try to decide on scope and direction. The approach chosen was based on types organized in classical object-oriented class hierarchies supported by concepts (§6) where they needed to be generic [Chochlík 2015; Chochlík and Naumann 2016; Chochlík et al. 2017]. This approach was developed and implemented primarily by Matóš Chochlík, Axel Naumann, and David Sankel. It resulted in a technical specification approved in 2019 [Sankel 2018].

During the (expected) long gestation period for static reflection, compile-time computation based on **constexpr** functions (§9.3.3) developed steadily and eventually proposals to base static reflection on functions rather than class hierarchies emerged. The main proponents were Andrew Sutton, Daveed Vandevoorde, Herb Sutter, and Faisal Vali [Sutton and Sutter 2018; Sutton et al.

2018]. The main arguments for the shift in focus was partly that analyzing and generating code are inherently functional and that the compile-time computation based on **constexpr** functions had developed to the point where metaprogramming and reflection merged. Another strong point (first presented by Daveed Vandevoorde) of that approach was that a compiler's internal data structures for functions were inherently smaller and more transient than for type hierarchies so they used significantly less memory and compilation was significantly faster.

At the February 2019 standards meeting in Cologne, David Sankel and Michael Park presented a design that combined the strengths of the two approaches [Sankel and Vandevoorde 2019]. At the most fundamental level only a single type exists. This maximizes flexibility and minimizes compiler overheads.

On top of that, a statically typed interface can be imposed by a form of safe-type conversion (from the low-level monotype **meta::info** to more specific types such as **meta::type_** and **meta::class_**). Here is an example based on [Sankel and Vandevoorde 2019]. It achieves the conversion from **meta::info** to more specific types through concept overloading (§6.3.2). Consider:

```
namespace meta {
    consteval std::span<type_> get_member_types(class_ c) const;
}

struct baz {
    enum E { /*...*/ };
    class Buz{ /*...*/ };
    using Biz = int;
};

void print(meta::enum_);     // print an enumeration
void print(meta::class_);    // print a class
void print(meta::type_);     // print any type

void f()
{
    constexpr meta::class_ metaBaz = reflexpr(baz);
    template for (constexpr meta::type member_ : get_member_types(metaBaz))
        print(meta::most_derived(member_));
}
```

The key new language features here is the **reflexpr** operator that returns a (meta) object describing its argument and **template for** [Sutton et al. 2019] that iterates over the elements of a heterogeneous structure by expanding each element according to its type.

In addition, there is a mechanism for injecting code into the program being compiled.

The likelihood is that something like this will become standard in C++23 or C++26.

As a side effect, the work on the ambitious reflection schemes spurred improvements to compile-time evaluation facilities:

- The set of type traits in the standard (§4.5.1)
- The macros for source location (e.g., **__FILE__** and **__LINE__**) were replaced by an intrinsic mechanism [Douglas and Jabot 2019]
- The facilities for compile-time computation (e.g., **consteval** for guaranteed compile-time evaluation)
- Expansion statements (**template for** – for iterating over tuple elements [Sutton et al. 2019]) for C++23.

10 C++ IN 2020

This section looks at how C++ was used in the second decade of 2000 and for what:

- §10.1: What is C++ used for?
- §10.2: The C++ community
- §10.3: Education and Research
- §10.4: Tools
- §10.5: Programming styles
- §10.6: Core Guidelines

Areas of use are much as in 2006 (§2.3). Some new areas been added, but mostly what we see is a wider and deeper use in the same and similar areas. C++ hasn't suddenly become a "web-app language," though there is some use even there [Obiltschnig et al. 2005]. For most programmers, C++ is still something in the background that offers stability, reliability, portability, and performance. For end users, C++ is invisible.

The change in programming styles has been more dramatic. C++11 is a far better language than C++98. It is easier to use well, more expressive, and deliver better performance. C++20, as deployed in 2020, is a similar improvement over C++11.

10.1 What Is C++ Used For?

To a first approximation, C++ is used everywhere and for everything. However, most uses are invisible, deep in the infrastructure of important systems.

Nobody knows everything about where C++ is used, or how. In 2015, the Czech company JetBrains commissioned a study [Kazakova 2015] that showed massive use in North America, Europe and the Middle East, and the Asia Pacific areas plus some use in South America. "Some use in South America" was 400,000 developers. The total was 4.4 million developers. The industries listed were (in order) finance, banking, games, front office, telecom, electronics, investment banking, marketing, manufacturing, and retail. All indications are that the size of the user population and the areas of use have steadily increased since 2015.

Here I will give a – necessarily somewhat personal, impressionistic, and very incomplete – view of the range of C++ usage in the 2006-2020 time span:

- *Industry*: Telecom (e.g., AT&T, Ericsson, Huawei, and Siemens), mobile devices (essentially all; signal processing, screen rendering, apps with significant performance or portability requirements), microelectronics (e.g., AMD, Intel, Mentor Graphics, and Nvidia), finance (e.g., Morgan Stanley and Renaissance), games (almost all), graphics and animation (e.g., Maya, Disney, and SideFx), block chain implementation (e.g., Ripple), databases (e.g., SAP, Mongo, MySQL, and Oracle), cloud (e.g., Google, Microsoft, IBM, and Amazon), AI and ML (e.g., the TensorFlow library), operations support (e.g., Maersk and AT&T).
- *Science*: Aerospace (e.g., Space X, Mars Rovers, the Orion crew vehicle, the James Webb Space Telescope), high Energy Physics (e.g., CERN, SLAC, FermiLab), biology (genetics, genome sequencing), exascale computing.
- *Teaching*: Most engineering schools worldwide.
- *Software development*: TensorFlow, tools, libraries, compilers, Emscripten (generating asm.js and WebAssembly from C++), run-time code generation, LLVM (the backend backbone of many new languages and of much tool building), XML and JSON parsers, heterogenous computing (e.g., SYCL [Khronos Group 2014–2020] and HPX [Stellar Group 2014–2020]).
- Web infrastructure: Browsers (Chrome, Edge, FireFox, and Safari), JavaScript engines (V8 and SpiderMonkey), JVMs (HotSpot and J9), Google and similar organizations (search, map-reduce, and file system).

- *Major Web applications*: Alibaba, Amadeus (airline ticketing), Amazon, Apple, Facebook, PayPal, Tencent (WeChat), Yandex.
- *Engineering*: Dassault (CAD/CAM), Lockheed Martin (airplanes).
- *Automotive*: Assisted driving [ADAS Wikipedia 2020; Mobileye 2020; NVIDIA 2020], software architecture [Autosar 2020; Autosar Wikipedia 2020], machine vision [OpenCV 2020; OpenCV Wikipedia 2020], BMW, GM, Mercedes, Tesla, Toyota, Volvo, Volkswagen, Waymo (Google self-driving cars).
- *Embedded systems*: smart watches and health monitors (e.g., Garmin), cameras and video equipment (e.g., Olympus and Canon), navigation aids (e.g., TomTom), coffee machines (e.g., Nespresso), farm-animal monitoring (e.g., Big Dutchman), production-line temperature control (e.g., Carlsberg).
- *Security*: Kaspersky, NSA, Symantec.
- *Medicine and biology*: Medical monitoring and imaging (e.g., Siemens, GE, Toshiba, and Phillips), tomography (e.g., CAT scanners), genome analysis, bioinformatics, radiation oncology (e.g., Elekta and Varian).

This only scratches the surface, but it demonstrates the breadth and depth of C++ use. Most C++ uses are invisible to its (indirect) users. Some of these uses started pre-2006, but many were initiated later. No major modern system is written exclusively in a single language, but C++ plays a major role in what is mentioned here.

It is easy to forget that many uses are quite mundane, yet important in our lives. Yes, C++ helps run NASA's deep space network, but it also runs in familiar gadgets, such as coffee machines, stereo speakers, and dishwashers. I was surprised to find it used in the advanced systems used to run modern pig farms.

10.2 The C++ Community

Compared to 2006, the 2020 C++ community is larger, growing, optimistic, vibrant, productive, and impatient for further improvements.

Compared to most programming language communities, the C++ community has always been amazingly disorganized and fragmented. The problem started early because I had (and have) no talent for building organizations. My employer at the time, AT&T Bell Labs, had no desire to develop a C++ community, but it seemed that everybody else was keenly interested and willing to spend money to build up their user base. The net effect was that many companies, such as Apple, Borland, GNU, IBM, Microsoft, and Zortech formed C++ communities focused on their customers, but there were no general C++ community – no center to the C++ community. There were magazines, but few (relative to the size of the C++ community) read them. There were conferences, but they tended to get absorbed into or mutated into general "object oriented" or "software development" conferences. There was no general C++ users' group.

Today there are many dozen local, national, and international C++ users' groups with some cooperation, and dozens of C++ conferences each with attendance into the hundreds:

- *The C++ Foundation* – founded in 2014 as non-profit organization to promote ISO C++ (and not any particular vendor's C++). It hosts the CppCon annual conference.
- *Boost* – a collection of peer-reviewed libraries and the community that builds and uses them. Boost holds an annual conference. Founded in 1999.
- *Meeting C++* – a very active network of user groups holding conferences (initially active in Germany). There are dozens of Meeting C++ conferences and meetings ("meetups") in various places. Founded in 2012.

- *ACCU* – the grandfather of all surviving C++ organizations; it publishes two magazines and holds annual conferences (primarily active in England). Founded as a C users' group in 1984.
- *isocpp.org* – the website of the C++ Foundation with C++-related news, information of the standards process, and useful links.
- *cppreference.com* – an excellent online reference; it even has a history section!
- *Conferences* – CppCon, ACCU, Meeting++, C++ Now (formerly BoostCon), Qt, NDC, std::cpp, plus several in Poland, Russia, China, Israel, and elsewhere. C++ tracks at general software conferences are also on the rise.
- *Blogs* – many, and podcasts.
- *Videos* – Videos have become a major source of information about new developments in C++. The major C++ conferences typically video the talks and post them for free access (e.g., CppCon, C++ Now, and Meeting++). Video interviews have become popular. The most popular hosting site is YouTube, which unfortunately is blocked in some countries with large C++ developer communities (e.g., China).
- *GitHub* – makes it easier to share code and to organize joint projects.

This is nowhere near a match for the centralized organizations of some languages and vendors, but it is a lively and varied set of interrelated communities and far, far more active than what the C++ community had in 2006. Also, several corporate user's groups and conferences are still active.

10.3 Education and Research

Has C++ education improved since the sorry state in 2006 (§2.3)? Maybe, but it is certainly still not a strength for C++ and most new efforts focus on information and training for people in industry. In most countries, most students emerge from universities with only weak and inaccurate understanding of C++ and the key techniques for using it. This is a serious problem for the C++ community. No language can succeed at an industrial scale without a steady stream of enthusiastic programmers mastering its key design and implementation techniques. So much could be done to improve software if more developers using C++ knew how to use it better! So much would be easier if graduates entered the workforce with a more accurate view of C++!

A problem for C++ is that educational establishments often treat programming as a low-level skill, rather than a foundational topic. Good software is essential to our civilization. To manage, we need to treat software development for critical systems as seriously as we treat mathematics and physics. A "one size fits all" approach to education and software development doesn't work. One semester's teaching isn't sufficient. We would never expect to teach English in just a few months and then expect the students to appreciate Shakespeare. Similarly, there is a difference between knowing the basics mechanics of a language and mastering the idioms and techniques used by professionals. C++ – like any major modern programming language – needs a variety of teaching approaches adjusted to the learners backgrounds and needs. Even when an educational institution is aware of these problems and would like to compensate, curricula are overcrowded and teachers have a hard time keeping up with industrial practice. SG20 (Education) is trying to help by outlining approaches to teaching and using modern C++. SG15 (Tooling) could become important by making tools supporting teaching more available.

Since C++11, there has been an increasing awareness of this. For example, Kate Gregory has produced some great videos on how to teach C++ [Gregory 2015, 2017, 2018]. Some recent books address the education problems head on by recognizing that there are several constituencies with different needs when it comes to support for education:

- *Programming: Principles and Practice using C++* [Stroustrup 2008a] – a textbook aimed at beginning university students and people doing self-study

- *A Tour of C++* [Stroustrup 2014d, 2018f] – a short (200 pages) overview aimed at more experienced programmers
- *Discovering Modern C++* [Gottschling 2015] – a book specifically for students with a strong mathematical background

I wrote a couple of semi-academic papers (*Software Development for Infrastructure* [Stroustrup 2012] and *What should we teach software developers? Why?* [Stroustrup 2010b]) and devoted the opening keynote of CppCon 2017 to education (*Learning and Teaching Modern C++* [Stroustrup 2017c]).

The use of videos and online courses has dramatically increased since about 2014. This serves C++ well because that doesn't require a central organization or significant funding.

Here is a sampling of academic research related to the C++ language in the 2006-2020 timeframe:

- *Concepts*: generic programming [Dehnert and Stepanov 2000], C++0x Concepts [Gregor et al. 2006], use patterns [Dos Reis and Stroustrup 2006], library design [Sutton and Stroustrup 2011].
- *Theory and formalism*: inheritance model [Wasserrab et al. 2006], templates and overloading [Dos Reis and Stroustrup 2005a], template semantics [Siek and Taha 2006], object layout [Ramananandro et al. 2011], construction and destruction [Ramananandro et al. 2012], a representation for code manipulation [Dos Reis and Stroustrup 2009, 2011], resource model [Stroustrup et al. 2015].
- *Dynamic lookup*: fast dynamic cast [Gibbs and Stroustrup 2006], pattern matching [Solodkyy et al. 2012], multimethods [Pirkelbauer et al. 2010].
- *Static Analysis*: sound representation [Yang et al. 2012], practical experience [Bessey et al. 2010].
- *Performance*: code bloat [Bourdev and Järvi 2006, 2011], exception implementation [Renwick et al. 2019].
- *Language comparisons*: generic programming [Garcia et al. 2007].
- *Concurrent and parallel programming*: memory model [Batty et al. 2013, 2012, 2011], HPX (a general-purpose C++ runtime system for parallel and distributed applications of any scale [Kaiser et al. 2009Sept.]), STAPL (An adaptive, generic parallel C++ library [Zandifar et al. 2014]), TBB (Intel's task parallelism library [Reinders 2007]).
- *Coroutines*: Database optimization [Jonathan et al. 2018; Psaropoulos et al. 2017].
- *Software engineering*: Code organization and optimization [Garcia and Stroustrup 2015], Constant expression evaluation [Dos Reis and Stroustrup 2010].

There seem to be opportunities for more academic research related to C++'s features and techniques (e.g., exception handling, compile-time programming, and resource management) as well as to the effectiveness of its use (e.g., static analysis or real-world code and empirical studies).

Few of the most active members of the C++ community would ever consider writing an academic paper; writing books seems more popular (e.g., [Čukić 2018; Gottschling 2015; Meyers 2014; Stepanov and McJones 2009; Vandevoorde et al. 2018; Williams 2018]).

10.4 Tools

In the early-to-mid 1990s, compared to other languages, C++ was doing reasonably well in the areas of tools and programming environments for industrial use. For example, graphical user interfaces and integrated software development environments were pioneered for C++. Later, the focus of development and investment shifted to proprietary languages, such as Java (Sun), C# (Microsoft), and Objective-C (Apple), and to simpler languages, such as C (GNU).

I see two major reasons:

- *Funding*: Organizations tend to prefer languages and tools that they can control and that give then a differential advantage over competitors. From that perspective, the fact that C++ is controlled by a formal standards committee emphasizing benefits for all is a disadvantage – a variant of the tragedy of the commons.
- *Macros and textual definition*: C++ did not have a simple, widely available internal representation to simplify tool building based on source code and the heavy use of macros ensured that what a programmer saw wasn't what the compiler analyzed. Like C, C++ is defined in terms of sequences of characters, rather than constructs that directly represent abstractions and are easier to manipulate. Together with Gabriel Dos Reis, I defined such a representation [Dos Reis and Stroustrup 2009, 2011], but the character-oriented traditions in the C++ community proved hard to overcome. It is hard to retrofit a regular structure on something not built with that in mind.

Consequently, in the 2006-2020 time frame, C++ suffered badly in the area of support tools compared to other languages. However, the situation also improved slightly by the emergence of

- *Industrial-strength integrated software development environments*: e.g., Microsoft's Visual Studio [Microsoft 2020; VStudio Wikipedia 2020] and JetBrain's Clion [Clion Wikipedia 2020; JetBrains 2020]. These environments support not just editing and debugging, but also various forms of analysis and simple code transformation.
- *Online compilers*: e.g., Compiler Explorer [Godbolt 2016] and Wandbox [Wandbox 2016–2020]. These systems allow the compilation of and sometimes execution of C++ programs from any browser. They are used for experimentation, examination of code quality, and comparison of different compilers and versions of compilers and libraries.
- *GUI libraries and tools*: e.g., Qt [Qt 1991–2020], GTKmm [GTKmm 2005–2020], and wxWidgets [wxWidgets 1992–2020]. Unfortunately, Qt relies on a meta-object protocol (MOP), so that Qt programs are not just ISO C++. Static reflection (§9.6.2) should eventually allow us to solve that problem. The problem for the C++ community is not that there are no good GUI libraries, but that there are so many that choosing one is hard.
- *Analyzers*: e.g., coverity [Coverity 2002–2020], Visual Studio's analyser for the C++ Core Guidelines (§10.6), and Clang tidy [Clang Tidy 2007–2020].
- *Compiler tool support*: e.g., the LLVM compiler back-end infrastructure simplifying code generation and code analysis [LLVM 2003–2020]. This provided a boon to many new languages, in addition to C++ itself.
- *Build systems*: e.g., build2 [Build2 2014–2020] and Cmake [Cmake 2000–2020], and GNUmake [GNUmake 2006–2020]. Again, in the absence of a standard, choosing one is hard.
- *Package managers*: e.g., Conan [Conan 2016–2020] and vcpkg [vcpkg 2016–2020].
- *Run-time environments*: e.g., WebAssembly: a system for compiling ISO C+ into bytecodes for deployment in browsers [WebAssembly 2017–2020].
- *Run-time compilation, JIT-ing, and linking*: e.g., Cling [Cling 2014–2020; Naumann 2012; Naumann et al. 2010] and RC++ [RC++ 2010–2020].

What is listed above are just examples. As usual, the problem for the C++ users is the number of alternatives: [RC++ 2010–2020] lists 26 systems for generating code at compile time and there are dozens of package managers. What is needed is some form of standardization.

As of 2020, tools are still not a strength of C++, but progress is being made over a broad front.

10.5 Programming Styles

A key driver of C++'s evolution is that optimal solutions to most real-world problems involve a combination of techniques. Naturally, this offends everybody who claims to have a single simple best solution (a "programming paradigm"), but supporting a variety of styles has always been a fundamental strength of C++. Consider the "draw all shapes" example that has been used to illustrate object-oriented programming since the early days of Simula (where the drawing device was a wet-ink plotter). In C++20, we might write:

```
void draw_all(range auto& seq)
{
    for (Shape& s : seq)
        s.draw();
}
```

What programming paradigm is that code?

- It is clearly object-oriented programming: it uses a virtual function and a class hierarchy.
- It is clearly generic programming: it uses a template (by parameterizing with the **range** concept, we get a template).
- It is clearly ordinary imperative programming: it uses a **for**-loop and defines a function to be called with the conventional **f(x)** syntax.

I could elaborate this example: A **Shape** usually has mutable state. I could use a lambda. I could call a C function. I could further constrain the argument with a **Drawable** concept. For a variety of definitions of "better", a suitable mix of techniques is a better solution than any I could come up with using a single paradigm.

The idea behind C++'s support of several programming styles ("paradigms" if you must) is not that you can choose to program in one favorite style, but that you can use them in combination to express better solutions than you could in a single style.

10.5.1 Generic Programming. In 2006, much C++ code was still a mixture of object-oriented styles and C-style programming. Naturally, there is still a lot of that in 2020. However, with C++98, the STL style of generic programming (often referred to as *GP*) became widely known and slowly user code started using GP beyond simple applications of the standard library. The improved support for GP in C++11 opened the floodgates for its use in production code. However, the lack of concepts (§6) in C++17 is still a drag on the use of generic programming in C++.

Essentially all experts read Alex Stepanov's *The Elements of Programming* (often referred to as *EoP*) [Stepanov and McJones 2009] and were influenced by it.

Generic programming using templates is the backbone of the C++ standard library: containers, ranges (§9.3.5), algorithms, iostreams, file system (§8.6), random numbers (§4.6), threads (§4.1.2) (§9.4), locking (§4.1.2) (§8.4), time (§4.6) (§9.3.6), strings, regular expressions (§4.6), and formats (§9.3.7).

10.5.2 Metaprogramming. Metaprogramming in C++ grew out of generic programming in the sense that both rely on templates. Its roots go back to the early days of templates in C++ when people discovered that templates were Turing complete [Vandevoorde and Josuttis 2002; Veldhuizen 2003] and offered compile-time pure functional programming in a somewhat useful form.

Template metaprogramming (often referred to as *TMP*) is often very ugly. Sometimes, that ugliness is hidden through the use of macros, creating further problems. It is a testimony to TMP's utility that it became almost universal. For example, you can't implement the C++14 standard library without metaprogramming. Many techniques and experiments were pre-2006, but it was C++11

with better compilers, variadic templates (§4.3.2), and lambdas (§4.3.1) that gave TMP the boost needed for mainstream use. The C++ standard library also added support, such as the compile-time selection template, **conditional**, type traits allowing code to depend on properties of types, such as "can type X safely be bitwise copied?" (§4.5.1), and **enable_if** (§4.5.1). For example:

```
conditional<(sizeof(int)<4),double,int> x; // if ints are small use double
```

Computing types to precisely mirror requirements is arguably the essential part of TMP. We can also calculate values:

```
template <unsigned n>
struct fac {
    enum { val = n * fac<n-1>::val };
};

template <>
struct fac<0> {   // specialization for 0: fac<0> is 1
    enum { val = 1 };
};

constexpr int fac7 = fac<7>::val;  // 5040
```

Note the key role of template specialization; it is essential in most TMP. It has been used to compute non-trivial values and to express control flows (e.g., to compute decisions tables at compile time and unroll loops). Specialization is a largely underappreciated feature of C++98 [Stroustrup 2007].

In clever libraries and also in real-world code, primitives like **enable_if** have become the basis of programs of many hundreds or even thousands of lines. Early examples of TMP included a complete compile-time Lisp interpreter [Czarnecki and Eisenecker 2000]. Debugging is extremely hard, maintenance of such code is horrendous, and I have seen a couple of hundred lines of (admittedly clever) TMP lead to compile times in minutes and failure to compile by running out of memory on a 30GB computer. The compiler error messages from even simple errors can run into many thousands of lines. And yet TMP is widely used. Sane programmers have found TMP, warts and all, preferable to alternatives. I have seen code generated from TMP that was better than I would expect a competent human to write in assembler.

Thus, the problem becomes to better serve such needs. I worry when people start to consider code like **fac<>** normal. That is not a good way to express an ordinary numerical algorithm. Concepts (§6) and compile-time evaluated functions (**constexpr** (§4.2.7)) can dramatically simplify metaprogramming. For example:

```
constexpr int fac(int n)
{
    int r = 1;
    while (n>1) r*=n--;
    return r;
};

constexpr int fac7 = fac(7);  // 5040
```

This example illustrates that when you want a value, a function is the best way to compute it, even – and especially – at compile time. Traditional *template* metaprogramming is best reserved for computing new types and control structures.

Jaakko Järvi's **boost::lambda** [Järvi and Powell 2002; Järvi et al. 2003a] was an early use of TMP that helped convince people that lambdas were useful and also that they needed direct language support.

The Boost metaprogramming library, **boost::MPL** [Gurtovoy and Abrahams 2002–2020], offered the best and worst of traditional TMP. A more modern library, **boost::Hana** [Boost Hana 2015–2020], uses **constexpr** functions. WG21's SG7 (§3.2) is trying to develop a better and standard metaprogramming system that also includes compile-time reflection (§9.6.2).

10.6 Coding Guidelines

My ultimate aim for C++ is a language that is

- Significantly simpler to use and to learn than C or current C++
- Completely type safe – no implicit type violations and no dangling pointers
- Completely resource safe – no leaks and no need for a garbage collector
- Relatively simple to build tools for – no macros
- As fast or faster than current C++ – the zero-overhead principle
- Predictable performance – suitable for embedded systems
- Not less expressive than current C++ – handle hardware well

This is not all that different from the design aims articulated in *The Design and Evolution of C++* [Stroustrup 1994] and earlier. Obviously, it's a tall order and incompatible with most uses of older C and C++.

From the earliest days of "C with Classes", people have suggested creating safe subsets of the language and compiler switches to enforce such safety. However, those suggestions failed for one of many reasons:

- Not enough people could agree on a definition of "safe."
- The unsafe features (for every definition of "unsafe") are the ones needed to build the basic safe abstractions.
- The safe subset is insufficiently expressive.
- The safe subset is inefficient.

The second reason implies that you cannot define a safe C++ simply by banning unsafe features. "Perfection through restriction" approaches to programming language design at best works in very limited situations. You need to consider the context and nature of the use of features that in general are unsafe but have safe uses. Furthermore, the standard cannot abandon backwards compatibility (§1.1), so we need a different approach.

From the start, C++ adopted a different philosophy [Stroustrup 1994]:

> *It is more important to enable good programming than to prevent errors.*

That implies that we need guidelines for "good use", rather than language rules. However, to be useful at an industrial scale, guidelines must be enforceable by tools. For example, from the earliest days of C and C++, we have known of the problems with dangling pointers. For example:

```
int* p = new int{7,9,11,13};
// ...
delete p;       // delete the array pointed to by p
                // p now doesn't point to a valid object; it "dangles"
// ...
*p = 7;        // likely disaster
```

Many programmers have developed techniques to prevent pointers from dangling. However, dangling pointers are still a major problem in most large code bases and security issues are much more critical than in the past. Some dangling pointers can be exploited as a security hole.

10.6.1 General Approach. In 2004, I helped develop a set of coding guidelines for flight control software [Lockheed Martin Corporation 2005] that approximated my ideals of safety, flexibility, and performance. In 2014, I started writing a set of coding guidelines to address this for use in a broad range of applications. This was in part a response to loud demands for practical guidelines for using C++11 well and part in horror over seeing what some people considered good C++11. Talking to people, I soon discovered the obvious: I wasn't the only one thinking and working along these lines. So, some experienced C++ programmers, tool builders, and library builders joined forces and started the *C++ Core Guidelines* project [Stroustrup and Sutter 2014–2020] with contributors from a broad spectrum of the C++ community. The project is open-source (MIT license) and the list of contributors can be found on GitHub. Early on, contributors from Morgan Stanley (notably me), Microsoft (notably Herb Sutter, Gabriel Dos Reis, and Neil Macintosh), Red Hat (notably Jonathan Wakely), CERN, Facebook, and Google were prominent.

The Core Guidelines are by no means the only C++ guidelines project, but they are the most prominent and most ambitious. They have the explicit and articulated aim of dramatically improving the quality of C++ code. For example, the ideals and basic model for complete type and resource safety was articulated early on in a paper by Bjarne Stroustrup, Herb Sutter, and Gabriel Dos Reis [Stroustrup et al. 2015].

To reach the ambitions goals, we apply a "cocktail" of approaches applied in concert:

- *Rules*: A large set of rules, aimed at type-safe and resource-safe use of C++, recommending known effective practices, and banning known sources of errors and inefficiencies.
- *Foundation libraries*: A set of library components to allow programmers to write efficient low-level programs without using known error-prone features and in general to provide a higher-level basis for programming. Most components are from the standard library and some from a small *Guidelines Support Library* (GSL) written in ISO standard C++.
- *Static analysis*: Tools to detect violations and enforce key parts of the guidelines.

Each of these techniques has a long history and each cannot by itself handle the problems at an industrial scale. For example, I am a great fan of static analysis, but the kinds of analysis I am most interested in (e.g., the elimination of dangling pointers) cannot be solved if a programmer can write arbitrarily complex code in a separately compiled program potentially using dynamic linking. Here "cannot" means "theoretically impossible in general" as well as "too computationally expensive for industrial-scale programs."

The basic approach is not simple restriction, but what I called the subset-of-superset approach or SELL [Stroustrup 2005]:

- *First, extend the language with library facilities to makes a solid base for use.*
- *Then, subset by removing unsafe, error-prone, and overly-costly facilities.*

For libraries, we primarily rely on parts of the standard library, such as **variant** (§8.3) and **vector**. The small *Guidelines Support Library* (GSL) offer support for type-safe access, such as **span** for range checked access to a contiguous sequence of elements of a given type (§9.3.8). The idea is to eventually eliminate the GSL by getting it absorbed into the ISO standard library. For example, **span** was added to the C++20 standard library and the feeble contract support in the GSL should be replaced by proper contracts if and when they become available (§9.6.1).

10.6.2 Static Analysis. To scale, the static analysis is completely local (one function or one class at a time). The hardest problems relate to lifetime of objects. RAII is essential: manual resource

management has repeatedly showed itself seriously error-prone in the many languages in which it is used. Furthermore, there are plenty of programs "out there" that use pointers and iterators in principled ways. Such uses must be accepted. To make a program safe is easy, you simply prohibit everything. However, maintaining both the expressiveness of C++ and its performance are among the aims of the Core Guidelines, so we can't gain safety simply through restriction. The aim is a better C++, not a slow or neutered subset.

These guidelines can help focus education on the more effective aspects of C++ by articulating principles, making good practices explicit, and mechanizing checking for known problems. They also help relieve pressures on the language to directly accommodate fashions.

For object lifetimes, there are two major needs:

- Never point to an object that has gone out of scope.
- Never access an invalidated object.

Consider an example from "the basic model" paper [Stroustrup et al. 2015]):

```
int glob = 666;

int* f(int* p)
{
    int x = 4;          // local variable
    // ...
    return &x;          // No! would point to destroyed stack frame
    // ...
    return &glob;       // OK: points to something that "lives forever"
    // ...
    return new int{7};  // OK (sort of: doesn't dangle,
                        //     but returns an owner as an int*)
    // ...
    return p;           // OK: came from caller
}
```

We can return a pointer that points to something known to outlive the function (e.g., came to the function as an argument) but not a pointer to a local. In a program following the guidelines, it is guaranteed that a pointer passes as an argument points to something or be the **nullptr**).

To avoid leaks, "naked **new**s" as in the example above are eliminated through the use of resource handles (RAII) or ownership annotations.

A pointer can be invalidated if the object it points to is reallocated. For example:

```
vector<int> v = { 1,2,3 };
int* p = &v[2];
v.push_back(4); // v's elements may be relocated
*p = 5;         // bad: p might have been invalidated

int* q = &v[2];
v.clear();     // all v's elements are deleted
*q = 7;        // bad: q is invalidated
```

Checking for invalidation is even harder than checking for simple dangling pointers because it is hard to be sure which functions move objects and whether that is to be considered invalidation (the pointer **p** still points to something, but conceptually to a different element). It is not yet clear if invalidation checking can be completely handled without annotation or non-local state in the static analyzer. In initial implementations, every function manipulating an object as a non-**const**

is assumed to invalidate, but that is too conservative, leading to too many false positives. The initial detailed specification of lifetime checking was written by Herb Sutter [Sutter 2019] and implemented by his colleagues at Microsoft.

Range checking and checking for **nullptr** is done with library support (GSL). Static analysis is then used to ensure that the libraries are used consistently.

The initial implementation of the static analysis ideas was by Neil Macintosh and is currently deployed as part of Microsoft Visual Studio. Some rules are checked as part of Clang and HSR's Cevelop (Eclipse plugins) [Cevelop 2014–2020]. Some courses and books are incorporating rules (e.g., [Stroustrup 2018f]).

The Core Guidelines are designed for gradual and selective adoption. As a result, we see widespread adoption of some guidelines in industry and education, but little complete adoption. For complete adoption good tool support is necessary.

11 RETROSPECTIVE

The ultimate aim of a programming language design is to improve the way programmers think and work when delivering useful applications. There are languages deemed "just experimental" but as soon as a language is used for work that is not related to the language itself, the language designers acquire a responsibility to their users. Correctness, appropriateness, stability, and adequate performance become important issues. For C++, that happened in 1979 after only 6 months. C++ has thrived for 40 years. Why and how?

My previous HOPL papers [Stroustrup 1993, 2007] offer answers from the perspective of 1991 and 2006. Apart from the language features and library components, what has changed since then is mostly the role and impact of the standards committee (§3).

Here, I consider

- §11.1: The C++ model
- §11.2: Technical successes
- §11.3: Areas that need work
- §11.4: Lessons learned
- §11.5: The future

11.1 The C++ Model

C++ emerged as a major – and in some areas dominant – language for demanding applications. It did so without serious commercial backing and with no marketing. Many modern languages copied features and ideas from it (§2.4). The key language-technical areas of contribution are:

- A static type system with equal support for built-in types and user-defined types (§2.1)
- Value *and* reference semantics (§4.2.3)
- Systematic and general resource management (RAII) (§2.2)
- Support for efficient object-oriented programming (§2.1)
- Support for flexible and efficient generic programming (§10.5.1)
- Support for compile-time programming (§4.2.7)
- Direct use of machine and operating system resources (§1)
- Concurrency support through libraries (often implemented using intrinsics) (§4.1) (§9.4)

C++ offers a different – and for many application areas better – model for software than the currently dominant "managed" model relying on garbage collectors and extensive run-time support typified by languages such as Java, C#, Python, and JavaScript (§2.3). By "better" I mean "simpler to write, more likely to be correct, more maintainable, using less memory, using less energy, and faster."

The areas of contribution are mutually supportive. For example

- Reference semantics (e.g., pointers and smart pointers) enables the efficient implementation of advanced types with value semantics (e.g., **jthread** and **vector**).
- Uniform rules for built-in and user-defined types simplify generic programming (built-in types are not special cases).
- Compile-time programming makes a range of abstraction techniques affordable for effective use of hardware.
- RAII allows use of user-defined types without taking specific actions to support their implementations' use of resources (including non-memory resources).

11.2 Technical Successes

The fundamental reason for C++'s success is simple: It fills an important "niche" in the programming landscape:

Applications that need to use hardware efficiently and to manage significant complexity.

If you can afford to "waste" 25% or 99% of your hardware capacity, there is no shortage of programming languages and environments to choose from and if your low-level needs represent only a few thousand lines of low-level code, C or assembler will serve. For 40 years, C++'s "niche" has been sufficient to keep its community growing.

Here is a modern (2014) summary of C++:

- *Direct map to hardware*
  - *of instructions and fundamental data types*
  - *Initially from C*
- *Zero-overhead abstraction*
  - *Classes with constructors and destructors, inheritance, generic programming, function objects*
  - *Initially from Simula (where it wasn't zero-overhead)*

Simula pioneered abstraction mechanisms and a flexible type system, but they came with heavy costs in run-time and space. Compared to the 1995 description of C++ (§2.1), the focus has shifted from programming techniques to problem areas. That's more a difference in explanation style and people's interest than in language design. Both summaries are accurate now and were then.

Building on the previous decades, the key technical advances of the 2000s include:

- The memory model (§4.1.1)
- Type-safe concurrency support: threads and locks (§4.1.2), parallel algorithms (§8.5), joining threads (§9.4)
- Type deduction: **auto** (§4.2.1), concepts (§6), template argument deduction (§8.1), variadic templates (§4.3.2)
- Simplification of use: **auto** (§4.2.1), range-**for**, (§4.2.2), parallel algorithms(§8.5), ranges(§9.3.5), lambdas (§4.3.1)
- Move semantics (§4.2.3)
- Compile-time programming: **constexpr** (§4.2.7), compile-time loops (§5.5), guaranteed compile-time evaluation and containers (§9.3.3), metaprogramming (§10.5.2)
- Generic programming: the STL (§10.5.1), concepts (§6), user-defined types as template parameters (§9.3.3), lambdas (§4.3.1)
- Metaprogramming (§10.5.2)

They all relate to the zero-overhead principle, but the last two are a bit surprising because of C++'s incomplete support for them in the 2006-2020 timeframe.

None of this would have mattered had C++ broken into incompatible dialects or become something you couldn't rely on for the long term:

- Stability and compatibility are essential (§1.1) (§11.4)

The new features (from C++11 onwards) led to improvements in the standard library (e.g., **unique_ptr**, **chrono**, **format**, and **scoped_lock**) and in many other libraries.

The purpose of C++ is to be a tool for building applications, so the many great applications of C++, such as those mentioned in (§2.3) and (§10.1), are the real successes of C++.

11.3 Areas That Need Work

No language is perfect for everybody and everything. Nobody knows this better than people who know several languages, seriously use a language, and try to support it. It is rarely simple ignorance that stands in the way of progress. Rather, the major obstacles to significant improvement are lack of direction, lack of development resources, and fear of breaking existing code.

C++ suffers from having been born before modern IDEs, build systems, GUI systems, and Unicode. I expect C++ to slowly catch up. For example:

- *Tooling*: Starting from C with a semantics specified in terms of characters and lexical tokens and source code organized with **#include**s and macros have been major barriers to effective tool building. Modules should help (§9.3.1) and it is possible to devise a sane internal representation for C++ [Dos Reis and Stroustrup 2009, 2011].
- *Education*: The C++ taught today is still mostly outdated and backwards looking (§2.3). The Core Guidelines (§10.6) is one approach to modernizing practice. WG21's education study group (§3.2) and the many education-oriented conference presentations (§10.3) show that the problems are appreciated and being addressed.
- *Packaging and distribution*: C++ was born before composing software out of independently developed and maintained parts was common. Today, there are build systems and package managers for C++. However, none are standard, some are hard to use for simple tasks, and others don't generalize to cope with the massive systems built using C++. My 2017 CppCon keynote challenged the C++ community to address this [Stroustrup 2017c] and I think we are seeing progress. In addition, the C++ community suffers from a lack of a standard place to find useful libraries. Boost [Boost 1998–2020] was an effort to address this and GitHub is emerging as a common repository, but there is a long way to go before a relative novice can find, download, install, and run a couple of major libraries.
- *Character sets and graphics*: The C++ language and standard library depend on ASCII, yet most applications use some form of Unicode. WG21 now has a study group to find a way to standardize Unicode support (§3.2). The lack of standard Graphics and GUI is a harder problem.
- *Clean-up of old messes*: This is unpleasantly hard. For example, we know that the implicit narrowing conversions among the built-in types cause endless problems (§9.3.8), but there are hundreds of billions of lines of C++ code that depend on those conversions in unpredictable ways. Attempts to improve by adding "more modern" features to replace the old ones easily fall prey to the N+1 problem (§4.2.5). Improved tooling (e.g., static program analysis and program transformation) offers hope.

The challenges for a large language community are diverse and not amenable to a single simple solution. It is not just an issue of syntax, type theory, or basic language design. Some problems are commercial. The range of skills needed for success at an industrial scale is daunting. Time will tell whether the C++ community will get to grips with all of this, and more. I'm modestly optimistic because there are initiatives in all areas (§3.2).

11.4 Lessons Learned

C++ is controlled by a large committee with a diverse and changing membership (§3.2). So in addition to technical issues, we must consider what works in the process of evolving the language:

- *Problem driven*: C++'s development should be driven by needs of specific real-world problems.
- *Simple*: C++ should be grown by generalization from simple, efficient, easy-to-use solutions.
- *Efficient*: The C++ language and standard library should obey the zero-overhead principle.
- *Stable*: Don't break my code!

The development of most (all?) of the most successful parts of C++ obeyed those "rules of thumb." They naturally limit the scope of the language, but that's good. C++ is not meant to be all things to all people. Furthermore, these principles forces C++ to grow relatively slowly based on real-world challenges and enable it to benefit from feedback. See also the other "rules of thumb" in *The Design and Evolution of C++* [Stroustrup 1994] and my HOPL2 paper [Stroustrup 1993]. There is continuity.

To contrast, facilities designed without a clear focus on the problems of the larger community usually fail:

- *Experts only*: A facility has to serve all the experts' needs from the start.
- *Imitative*: We need this facility because it is popular in "language X."
- *Theoretical*: Language theory says that a language must have this feature.
- *Revolutionary*: This feature is so important that we must break compatibility or deprecate "the bad old ways" of doing things.

I conclude that setting direction and expectations early is essential. Later, there will be too many people with too diverse opinions to get agreement on a coherent set of ideas.

Given a direction and a set of principles, a language can grow based on feedback, user experience, experiments, and theory as tools. This is good engineering as opposed to unprincipled pragmatism or dogmatic idealism.

The C++ standards committee's charter focuses almost exclusively on language and library design. That's limiting. Important topics such as dynamic linking, build systems, and static analysis have been mostly ignored. That's a mistake. Tools are a huge part of a software developer's world and it would be nice if they were not peripheral to language design.

Enthusiasm for a diverse set of idea is dangerous. In a 2018 paper [Stroustrup 2018d], I listed 51 recent proposals:

> *"I list papers that I think have the potential to significantly change the way we write code, so that each has significant implications on teaching, maintenance, and coding guidelines. Many also have implications for implementations.*
> *Individually, many (most?) proposals make sense. Together they are insanity to the point of endangering the future of C++."*

That paper was entitled *Remember the Vasa!*. The Vasa was a magnificent 17th century Swedish battleship that because of late additions to its design and insufficient testing sank in Stockholm harbor on its maiden voyage. In the 1990s, the committee often reminded itself of the Vasa, but in the 2010s that lesson seemed to have been forgotten.

In an attempt to put organizational constraints on the committee process, the DG proposed "The C++ Programmers' Bill of Rights" [Dawes et al. 2018]:

(1) *Compile-time stability*: Every significant change in behavior in a new version of the standard is detectable by a compiler for the previous version.
(2) *Link-time stability*: ABI breakage is avoided except in rare cases, which will be well documented and supported by a written rationale.

(3) *Compiler performance stability*: Changes will not imply significant added compile-time costs for existing code.
(4) *Run-time Performance stability*: Changes will not imply significant added run-time costs to existing code.
(5) *Progress*: Every revision of the standard will offer improved support for some significant programming activity or community.
(6) *Simplicity*: Every revision of the standard will offer simplification for some significant programming activity.
(7) *Timeliness*: Every revision of the standard will be shipped on time according to a published schedule.

The next decades will show how this works out.

11.5 The Future

For the near term, C++20 will be as great a boon to the C++ community as C++11 was. At the February 2020 meeting in Prague where the committee finalized C++20, it also voted in favor of Ville Voutilainen's "bold plan for C++23" [Voutilainen 2019b]:

"work towards having the following things in C++23: "

- Library support for coroutines (§9.3.2)
- A modular standard library (§9.3.1)
- The general asynchronous computing model (executors) (§8.8.1)
- Networking (§8.8.1)

Note the focus is on libraries. "Also make progress on:"

- Static reflection facilities (§9.6.2)
- Functional-programming style pattern matching (§8.2)
- Contracts (§9.6.1)

Given that work on these topics is quite advanced, the odds are that the committee will accomplish most. What else that large group of enthusiasts can develop and agree upon is less predictable. For the next few years, the direction group (of which I am a member) mentions some promising areas for further work [Hinnant et al. 2020]:

- Improved Unicode support
- Support for simple graphics and simple user interaction
- Support for new kinds of hardware
- Exploration of improved styles and implementations of error-handling

Outside the committee, I expect to see significant progress on build systems, package management, and static analysis (§10.4).

Beyond that – five, ten, or more years into the future – my crystal ball gets a bit cloudy. For that time scale, we need to look at fundamentals rather than specific language features. I hope the standards committee will heed the lessons learned (§11.4) and focus on the fundamentals (§11.1):

- Pursue the goal of a completely resource-safe and type-safe C++
- Support a wide variety of hardware well
- Maintain C++'s record of stability (compatibility)

Maintaining stability requires a focus on compatibility as well as resisting the urge to try to radically improve C++ by adding a multitude of "perfect" features to replace imperfect or unfashionable older ways of doing things. New features invariably cause surprises (some pleasant, some not so pleasant) and the older features will not simply disappear. Remember the Vasa!

[Stroustrup 2018d] (§11.4). Often libraries, guidelines, and tools are superior alternatives to language changes.

Hardware isn't getting any faster for single threaded computation, so the premium on efficiency will remain and the pressure to effectively support various forms of concurrency and parallelism will increase (§2.3). Specialized hardware will proliferate (e.g., a variety of memory architectures and special-purpose processors); that will benefit languages, such as C++, that can take advantage of it. The only thing that increases faster than hardware performance is human expectation.

Systems are getting increasingly complex, so affordable abstraction mechanisms will also increase in importance. For systems relying on real-time interaction, predictable performance is essential (e.g., many real-time systems ban the use of free store (dynamic memory)).

Security concerns will only increase in importance as our dependence on computerized systems increases and the number of sophisticated hackers grow. To defend, I'd bet on hardware protection and on more structured systems supporting better static analysis, rather than endless ad-hoc run-time checks and low-level code.

Interoperability among languages and systems will remain essential; few major systems will be written in a single language.

As systems become more complex and the requirements for dependability increase, the need for quality of design and coding grow dramatically. I think C++ is well prepared for this and the plans for C++23 are to strengthen it further. However, language features alone are not sufficient to satisfy future demands. We need guidelines supported by tools to ensure effective use (§10.6). In particular, we need to ensure complete type safety and resource safety. This has to be reflected in education. To thrive, C++ needs better educational materials for novices and some to help experienced programmers master modern C++. Presentations of clever tricks and advanced uses are not sufficient and can harm the language by reinforcing its reputation of complexity.

For many reasons, we need to simplify the majority of C++ use. C++ has evolved to make that possible and I expect this trend will continue (§4.2). Improved optimizers – capable of taking advantage of the type system and abstractions used in the code – make a difference. Over the last few years, this has dramatically changed the way I approach optimizing code: I start by chucking away the clever and complicated stuff. That's where the bugs hide and if I have trouble understanding what's going on, so will the compiler and optimizer. I find that this approach usually gives me modest to spectacular performance improvements, as well as simplifying future maintenance. Only if this doesn't give me the performance I want, do I resort to advanced (aka complicated) data structures and algorithms. This is a triumph of the design of the C++ abstraction mechanisms.

I look forward to seeing many more exciting applications built with C++ and to seeing new programming idioms and design techniques developed.

I hope other languages learn from C++'s successes. It would be sad if the lessons learned from C++'s evolution were limited to the C++ community. I hope and expect to see key aspects of C++'s model in other languages and systems; that would be a true measure of success. To a limited extent, this has already happened (§2.4).

ACKNOWLEDGMENTS

I am painfully aware that

- This paper is far too long.
- The description of most technical topics leaves out much that could be considered essential. Often, the work of many people over many months spread over many years is reduced to a single page or even a single sentence. In particular, I shortchanged the immensely important topic of concurrency; it deserves a long, detailed paper of its own.

Thanks to the millions of programmers who made C++ a success. Their applications are critical parts of our world.

Thanks to reviewers of drafts of this paper, including Al Aho, A. Bratterud, Shigeru Chiba, J. Daniel Garcia, Brent Hailpern, Howard Hinnant, Roger Orr, Aaron Satlow, Yannis Smaragdakis, David Vandevoorde, J.C. Van Winkel, and Michael Wong. This paper relies strongly on reviewers for completeness and accuracy. Errors are of course my own.

Thanks to Guy Steele who helped me navigate the mysteries of LaTex and BibTex to bring the references into the form required by the ACM.

Thanks to all who worked so hard on the standards. The names of many more than I could mention can be found as authors of WG21 papers and in the acknowledgement sections of those papers. The many 'P' and 'N' numbered papers that I reference and quote can all be found at open-std.org/jtc1/sc22/wg21/docs/papers/. Without those papers, this paper would have had to rely far too much on my memory.

REFERENCES

David Abrahams, Rani Sharoni, and Doug Gregor. 2010. Allowing Move Constructors to Throw (Rev. 1). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3050. 12 March 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3050.html (also at Internet Archive 4 Sept. 2019 02:05:46).

ADAS Wikipedia 2020. ADAS: Advanced driver-assistance systems. https://en.wikipedia.org/wiki/Advanced_driver-assistance_systems (also at Internet Archive 9 May 2019 04:16:15).

James Adcock. 1990. Request for Consideration – Overloadable Unary operator.(). 8 Oct. 1990. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1990/WG21%201990/X3J16_90%20WG21%20Request%20for%20Consideration%20-%20Overloadable%20Unary%20operator.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Doug Lea, and Bill Pugh. 2004. Memory model for multithreaded C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1680. 10 Sept. 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Matthew Austern. 2001. A Proposal to Add Hashtables to the Standard Library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1326. 17 Oct. 2001. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2001/n1326.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Autosar 2020. AUTOSAR: The standardized software framework for intelligent mobility (website). https://www.autosar.org/ (also at Internet Archive 23 Aug. 2019 18:35:20).

AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor and software industry.

Autosar Wikipedia 2020. AUTOSAR (AUTomotive Open System ARchitecture). https://en.wikipedia.org/wiki/AUTOSAR (also at Internet Archive 7 Aug. 2019 19:39:43).

Lewis Baker. 2019. Coroutines TS Simplifications. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1477R1. 12 Feb. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1477r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. 1963. The Main Features of CPL. *The Computer Journal* 6, 2 (Aug.), 134–143. 0010-4620 https://doi.org/10.1093/comjnl/6.2.134 Also at https://academic.oup.com/comjnl/article-pdf/6/2/134/1041447/6-2-134.pdf

JF Bastien. 2018. Signed Integers are Two's Complement. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0907R0. 9 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0907r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library Abstraction for C/C++ Concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy, Jan.) (*POPL '13*). Association for Computing Machinery, New York, NY, USA, 235–248. 978-1450318327 https://doi.org/10.1145/2429069.2429099

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA, Jan.) (*POPL '12*). Association for Computing Machinery, New York, NY, USA, 509–520. 978-1450310833 https://doi.org/10.1145/2103656.2103717

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2010. Mathematizing C++ Concurrency: The Post-Rapperswil Model. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3132. 23 Aug. 2010. http://www.open-std.org/jtc1/sc22/

wg21/docs/papers/2010/n3132.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA, Jan.) (*POPL '11*). Association for Computing Machinery, New York, NY, USA, 55–66. 978-1450304900 https://doi.org/10.1145/1926385.1926394

Pete Becker. 2004. A Multi-threading Library for Standard C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1682. 10 Sept. 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1682.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Pete Becker (Ed.). 2009. Working Draft, Standard for Programming Language C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2914. 2 June 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2914.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Pete Becker (Ed.). 2011. *ISO/IEC 14882:2011: Information Technology — Programming languages — C++*. ISO (International Organization for Standardization), Geneva, Switzerland (Sept.). 1338 book pages. https://www.iso.org/standard/50372.html Status: withdrawn.

Joshua Berne. 2019. "Axiom" is a False Friend. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1672R0. 16 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1672r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Joshua Berne, Nathan Burgers, Hyman Rosen, and John Lakos. 2018. Contract Checking in C++: A (long-term) Road Map. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1332R0. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1332r0.txt (also at Internet Archive 4 Sept. 2019 02:05:46).

Joshua Berne and John Lakos. 2018a. Assigning Concrete Semantics to Contract-Checking Levels at Compile Time. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1333R0. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1333r0.txt (also at Internet Archive 4 Sept. 2019 02:05:46).

Joshua Berne and John Lakos. 2018b. Specifying Concrete Semantics Directly in Contract-Checking Statements. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1334R0. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1334r0.txt (also at Internet Archive 4 Sept. 2019 02:05:46).

Joshua Berne and John Lakos. 2019. Contracts That Work. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1429R2. 16 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1429r2.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb.), 66–75. 0001-0782 https://doi.org/10.1145/1646353.1646374

Peter Bindels, Ben Craig, Steve Downey, Rene Rivera, Tom Honermann, Corentin Jabot, and Stephen Kelly. 2018. Concerns about module toolability. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1427R0. 20 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1427r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Hans Boehm and Michael Spertus. 2005. Transparent Garbage Collection for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1833. 24 June 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1833.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Hans-J. Boehm, Mike Spertus, and Clark Nelson. 2008. Minimal Support for Garbage Collection and Reachability-Based Leak Detection (revised). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2586. 16 March 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2586.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Boost 1998–2020. Boost C++ Libraries (website). http://www.boost.org/ (also at Internet Archive 7 April 2020 20:27:31).

Boost provides free peer-reviewed portable C++ source libraries.

We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages the use of Boost libraries for all users with minimal restrictions.

We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Beginning with the ten Boost Libraries included in the Library Technical Report (TR1) and continuing with every release of the ISO standard for C++ since 2011, the C++ Standards Committee has continued to rely on Boost as a valuable source for additions to the Standard C++ Library.

Boost Hana 2015–2020. Boost Hana (website). https://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/index.html Archived at https://web.archive.org/web/2019*/https://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/index.html

Hana is a header-only library for C++ metaprogramming suited for computations on both types and values. The functionality it provides is a superset of what is provided by the well established Boost.MPL and Boost.Fusion libraries. By leveraging C++11/14 implementation techniques and idioms, Hana boasts faster compilation times and runtime performance on par or better than previous metaprogramming libraries, while noticeably increasing the level of expressiveness in the process. Hana is easy to extend in a ad-hoc manner and it provides out-of-the-box

inter-operation with Boost.Fusion, Boost.MPL and the standard library.

Vicente Botet and JF Bastien. 2018. std::expected. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0323R6. 2 April 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0323r6.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lubomir Bourdev and Jaakko Järvi. 2006. Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat. In *Proceedings of the Second International Workshop on Library-Centric Software Design (LCSD '06), co-located with OOPSLA 2006* (Portland, Oregon, USA, 22 Oct.), Andreas Priesnitz and Sibylle Schupp (Eds.). Computer Science and Engineering Department, Chalmers University of Technology, Göteborg, Sweden, 15–24. http://sms.cs.chalmers.se/bibliography/proceedings/2006-LCSD.pdf (also at Internet Archive 6 Feb. 2007 22:16:12). Technical Report 06-18.

Lubomir Bourdev and Jaakko Järvi. 2011. Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat. *Science of Computer Programming* 76, 4 (April), 243–257. https://doi.org/10.1016/j.scico.2008.06.003

Walter E. Brown, Mark Fischler, Jim Kowalkowski, and Marc Paterno. 2006. Random Number Generation in C++0X: A Comprehensive Proposal. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1932. 23 Feb. 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1932.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Build2 2014–2020. Build2, an open source, cross-platform build toolchain for developing and packaging C and C++ code. (website). http://build2.org/ (also at Internet Archive 17 Dec. 2019 16:03:15).

`build2` is an open source (MIT), cross-platform build toolchain for developing and packaging C and C++ code. It is a hierarchy of tools that includes the build system, package dependency manager (for package consumption), and project dependency manager (for project development).

Casey Carter. 2018. Standard Library Concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0898R0. 12 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0898r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Cevelop 2014–2020. Cevelop: An Eclipse-based IDE (website; initial release: 21 July 2014). https://www.cevelop.com/ (also at Internet Archive 10 Oct. 2019 02:56:01).

IDE++: Cevelop extends Eclipse CDT with many additional features: CUTE unit testing with Test Driven Development support, new refactorings and quick fixes, and much more. Cevelop comes with many tools to upgrade your code to C++11/14.

Matúš Chochlík. 2014. Static reflection. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3996. 26 May 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3996.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Matúš Chochlík. 2015. A case for strong static reflection. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4452. 11 April 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4452.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Matúš Chochlík and Axel Naumann. 2016. Static reflection (revision 4). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0194R0. 8 Feb. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0194r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Matúš Chochlík, Axel Naumann, and David Sankel. 2017. Static Reflection in a Nutshell. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0578R1. 18 June 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0578r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Clang Tidy 2007–2020. Clang Tidy: A collection of rule checkers for the Clang compiler (website). https://clang.llvm.org/extra/clang-tidy/ (also at Internet Archive 4 April 2020 14:39:53).

clang-tidy is a clang-based C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. **clang-tidy** is modular and provides a convenient interface for writing new checks.

Cling 2014–2020. Cling, an interactive C++ interpreter from CERN (website). https://root.cern.ch/cling (also at Internet Archive 28 Jan. 2020 05:14:14).

Cling is an interactive C++ interpreter, built on the top of LLVM and Clang libraries. Its advantages over the standard interpreters are that it has command line prompt and uses just-in-time (JIT) compiler for compilation. Many of the developers (e.g. Mono in their project called CSharpRepl) of such kind of software applications name them interactive compilers.

One of Cling's main goals is to provide contemporary, high-performance alternative of the current C++ interpreter in the ROOT project - CINT. The backward-compatibility with CINT is major priority during the development.

Clion Wikipedia 2020. JetBrains (formerly IntelliJ Software). https://en.wikipedia.org/wiki/JetBrains (also at Internet Archive 10 April 2020 22:12:43).

Marshall Clow, Beman Dawes, Gabriel Dos Reis, Stephan T. Lavavej, Billy O'Neal, Bjarne Stroustrup, and Jonathan Wakely. 2018. Standard Library Modules. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0581R1. 11 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0581r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Cmake 2000–2020. Cmake, an open-source, cross-platform family of tools designed to build, test and package software (website). https://cmake.org/ (also at Internet Archive 8 April 2020 14:02:11).

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. The suite of CMake tools were created by Kitware in response to the need for a powerful, cross-platform build environment for open-source projects such as ITK and VTK. CMake is part of Kitware's collection of commercially supported open-source platforms for software development.

Jonathan Coe and Roger Orr. 2015. Extension methods for C++ (Uniform-function-calling syntax lite). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0079R0. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0079r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Conan 2016–2020. Conan, the C / C++ Package Manager for Developers (website). https://conan.io/ (also at Internet Archive 28 Jan. 2020 04:44:48).

The open source, decentralized and multi-platform package manager to create and share all your native binaries.

Melvin E. Conway. 1963. Design of a Separable Transition-Diagram Compiler. *Commun. ACM* 6, 7 (July), 396–408. 0001-0782 https://doi.org/10.1145/366663.366704

Coverity 2002–2020. Coverity: static analysis tools (website). https://scan.coverity.com/ (also at Internet Archive 5 March 2020 17:57:39).

Find and fix defects in your Java, C/C++, C#, JavaScript, Ruby, or Python open source project for free. Test every line of code and potential execution path. The root cause of each defect is clearly explained, making it easy to fix bugs. Integrated with GitHub. More than 6600 open source projects and 33000 developers use Coverity Scan.

Cppreference 2011–2020. Cppreference: An online reference for the C++ language and standard library (website). https://en.cppreference.com/w/Main_Page (also at Internet Archive 31 March 2020 00:33:10).

Our goal is to provide programmers with a complete online reference for the C and C++ languages and standard libraries, i.e., a more convenient version of the C and C++ standards.

Lawrence Crowl. 2009. Reaching Scope of Lambda Expressions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2957. 25 Sept. 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2957.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lawrence Crowl. 2013. Digit Separators. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3661. 19 April 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3661.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lawrence Crowl. 2015a. Comparison in C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4367. 8 Feb. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4367.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lawrence Crowl. 2015b. Handling Disappointment in C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0157R0. 7 Nov. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0157r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lawrence Crowl. 2018. Ambiguity and Insecurities with Three-Way Comparison. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1380R0. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1380r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lawrence Crowl and Thorsten Ottosen. 2006. Proposal to add Contract Programming to C++ (revision 4). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1962. 25 Feb. 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lawrence Crowl, Richard Smith, Jeff Snyder, and Daveed Vandevoorde. 2013. Single-Quotation-Mark as a Digit Separator. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3781. 25 Sept. 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3781.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Ivan Čukić. 2018. *Functional Programming in C++: How to improve your C++ programs using functional techniques.* Manning Publications, Shelter Island, NY, USA. 320 book pages. 978-1617293818

Krysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley Professional, Reading, Massachussets, USA. 864 book pages. 978-0201309775

Beman Dawes. 2002–2014. The Boost Filesystem Library Design (website). https://www.boost.org/doc/libs/1_67_0/libs/filesystem/doc/design.htm (also at Internet Archive 10 April 2020 20:04:42).

The key feature C++ lacked for script-like applications was the ability to perform portable filesystem operations on directories and their contents. The Filesystem Library was developed to fill that void.

The intent is not to compete with traditional scripting languages, but to provide a solution for situations where C++ is already the language of choice.

Beman Dawes. 2006. Raw String Literals. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2053. 6 Sept. 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2053.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Beman Dawes (Ed.). 2014. Programming languages — C++ — File System Technical Specification. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4100. 4 July 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4100.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Beman Dawes (Ed.). 2015. *ISO/IEC TS 18822:2015: Programming languages — C++ — File System Technical Specification.* ISO (International Organization for Standardization), Geneva, Switzerland (July). 63 book pages. https://www.iso.org/standard/63483.html Status: withdrawn.

Beman Dawes, Howard Hinnant, Bjarne Stroustrup, David Vandevoorde, and Michael Wong. 2018. Direction for ISO C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0939R0. 10 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0939r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Beman Dawes, Eric Niebler, and Casey Carter. 2016. Iterator Facade Library Proposal for Ranges. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0186R0. 11 Feb. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0186r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Beman G. Dawes. 1998. Proposal for a C++ Library Repository Web Site. 6 May 1998. https://www.boost.org/users/proposal.pdf (also at Internet Archive 16 Dec. 2019 15:10:40).

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, California, USA, Dec.) (*OSDI '04*). USENIX Association, Berkeley, California, USA, 137–149. https://www.usenix.org/legacy/events/osdi04/tech/full_papers/dean/dean.pdf (also at Internet Archive 8 Feb. 2020 05:00:02).

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan.), 107–113. 0001-0782 https://doi.org/10.1145/1327452.1327492

James C. Dehnert and Alexander Stepanov. 2000. *Fundamentals of Generic Programming.* LNCS, Vol. 1766. Springer-Verlag, Berlin, 1–11. 978-3-540-41090-4 https://doi.org/10.1007/3-540-39953-4_1 Also at http://stepanovpapers.com/DeSt98.pdf (also at Internet Archive 6 June 2019 02:08:06).

Peter Dimov, Beman Dawes, and Greg Colvin. 2003. A Proposal to Add General Purpose Smart Pointers to the Library Technical Report. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1431. 28 Feb. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1431.htm (also at Internet Archive 24 Dec. 2016 19:20:32).

Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, and Daveed Vandevoorde. 2019. More constexpr containers. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0784R5. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r5.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Peter Dimov and Vassil Vassilev. 2018. Allowing Virtual Function Calls in Constant Expressions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1064R0. 4 May 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1064r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Louis Dionne. 2016. Lambdas in unevaluated contexts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0315R0. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0315r0.pdf (also at Internet Archive 21 April 2020 02:00:28).

Louis Dionne. 2017. Familiar template syntax for generic lambdas. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0428R2. 13 July 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0428r2.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Louis Dionne. 2018. Making std::vector constexpr. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1004R1. 7 Oct. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1004r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Louis Dionne and David Vandevoorde. 2018. Changing the active member of a union inside constexpr. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1330R0. 10 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1330r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis. 2003. Generalized Constant Expressions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1521. 21 Sept. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1521.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis. 2009. IPR: Compiler-neutral Internal Program Representation for C++ (code repository). http://github.com/GabrielDosReis/ipr (also at Internet Archive 10 Jan. 2019 00:58:20).

Gabriel Dos Reis. 2012. A System for Axiomatic Programming. In *Intelligent Computer Mathematics: 11th International Conference (CICM 2012)* (Bremen, Germany, July). Springer-Verlag, Berlin, Germany, 295–309. 978-3-642-31373-8 https://doi.org/10.1007/978-3-642-31374-5_20 LNCS 7362.

Gabriel Dos Reis (Ed.). 2018. Working Draft, Extensions to C++ for Modules. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4720. 29 Jan. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis, J-Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. 2016a. A Contract Design. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0380R0. 28 May 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0380r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis, J-Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. 2018. Support for contract-based programming in C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0542R4. 2 April 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r4.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis, Mark Hall, and Gor Nishanov. 2014. A Module System for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4047. 27 May 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis and Mat Marcus. 2003. Proposal to add template aliases to C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1449. 7 April 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1449.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis, Nathan Sidwell, Richard Smith, and David Vandevoorde. 2019. Modules are ready. 14 Feb. 2019. Note on the WG21 mailing list.

Gabriel Dos Reis and Richard Smith. 2018a. Modules for Standard C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1087R0. 7 May 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1087r0.pdf (also at Internet Archive 21 April 2020 03:04:09).

Gabriel Dos Reis and Richard Smith. 2018b. Plan of Record for Making C++ Modules Available in C++ Standards. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0983R0. 1 April 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0983r0.pdf (also at Internet Archive 21 April 2020 16:08:45).

Gabriel Dos Reis and Bjarne Stroustrup. 2005a. A Formalism for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1885. 20 Oct. 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1885.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis and Bjarne Stroustrup. 2005b. Specifying C++ concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1886. 20 Oct. 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1886.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis and Bjarne Stroustrup. 2006. Specifying C++ Concepts. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA, Jan.) (*POPL '06*). Association for Computing Machinery, New York, NY, USA, 295–308. 1595930272 https://doi.org/10.1145/1111037.1111064

Gabriel Dos Reis and Bjarne Stroustrup. 2007. Constant Expressions in the Standard Library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2219. 11 March 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2219.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis and Bjarne Stroustrup. 2009. A Principled, Complete, and Efficient Representation of C++. In *Proceedings of the Joint Conference of the 9th Asian Symposium on Computer Mathematics (ASCM 2009) and 3rd International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS 2009)* (Fukuoka, Japan, 14 Dec.), Masakazu Suzuki, Hoon Hong, Hirokazu Anai, Chee Yap, Yousuke Sato, and Hiroshi Yoshida (Eds.). Kyushu University, Fukuoka, Japan, 407–421. https://catalog.lib.kyushu-u.ac.jp/opac_download_md/16844/miln_22.pdf (also at Internet Archive 13 April 2020 14:57:31). Also at http://stroustrup.com/macis09.pdf http://hdl.handle.net/2324/16844 MI Lecture Note Series (formerly COE Lecture Note Series) Vol. 22.

Gabriel Dos Reis and Bjarne Stroustrup. 2010. General Constant Expressions for System Programming Languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland, March) (*SAC '10*). Association for Computing Machinery, New York, NY, USA, 2131–2136. 978-1605586397 https://doi.org/10.1145/1774088.1774537

Gabriel Dos Reis and Bjarne Stroustrup. 2011. A Principled, Complete, and Efficient Representation of C++. *Mathematics in Computer Science* 5, 3 (Sept.), 335–356. 1661-8270 https://doi.org/10.1007/s11786-011-0094-1

Gabriel Dos Reis, Bjarne Stroustrup, and Alisdair Meredith. 2009. Axioms: Semantics Aspects of C++ Concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2887. 21 June 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2887.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Gabriel Dos Reis, Herb Sutter, and Jonathan Caves. 2016b. Refining Expression Evaluation Order for Idiomatic C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0145R2. 3 March 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0145r2.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Robert Douglas and Corentin Jabot. 2019. Adopt source location from Library Fundamentals V3 for C++20. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1208R5. 17 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1208r5.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Stefanus du Toit (Ed.). 2014. *ISO/IEC 14882:2014: Information Technology — Programming languages — C++*. ISO (International Organization for Standardization), Geneva, Switzerland (Dec.). 1358 book pages. https://www.iso.org/standard/64029.html Status: withdrawn.

ECMA International. 2005. *C++/CLI Language Specification*. ECMA International, Geneva, Switzerland (Dec.). https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-372.pdf (also at Internet Archive 13 April 2019 01:09:02). ECMA Standard ECMA-372.

Robert Klarer (editor). 2007. Extension for the programming language C++ to support decimal floating-point arithmetic. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2198. 12 March 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2198.pdf (also at Internet Archive 4 Sept. 2019 02:05:46). ISO/IEC WDTR 24733.

David Eisenbud, Daniel R. Grayson, Michael Stillman, and Bernd Sturmfel (Eds.). 2001. *Computations in Algebraic Geometry with Macaulay 2.* Algorithms and Computation in Mathematics, Vol. 8. Springer, Berlin, Germany (Oct.). 345 book pages. 978-3540422303

Margaret A. Ellis and Bjarne Stroustrup. 1990. *The C++ Annotated Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA. 0-201-51459-1

Hal Finkel and Richard Smith. 2016. Inline Variables. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0386R0. 30 May 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0386r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Eric Fiselier. 2019. Adding the 'constinit' keyword. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1143R1. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1143r1.md (also at Internet Archive 4 Sept. 2019 02:05:46).

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, Reading, Massachussets, USA. 416 book pages. 978-0201633610

J. Daniel Garcia. 2018. Avoiding undefined behavior in contracts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1290R0. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1290r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

J. Daniel García, Francesco Logozzo, Gabriel Dos Reis, Manuel Fähndrich, and Shuvendu Lahiri. 2015. Simple Contracts for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4415. 12 April 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4415.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

J. Daniel Garcia and Bjarne Stroustrup. 2015. Improving performance and maintainability through refactoring in C++11. 27 Aug. 2015. https://pdfs.semanticscholar.org/1a84/826ad4968082761952b7ef0f5f6bc65f39e7.pdf (also at Internet Archive 19 Feb. 2019 20:02:29). Also at Research Gate: https://www.researchgate.net/publication/285576155_Improving_performance_and_maintainability_through_refactoring_in_C11_Improving_performance_and_maintainability_through_refactoring_in_C11

Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. 2007. An Extended Comparative Study of Language Support for Generic Programming. *Journal of Functional Programming* 17, 2 (March), 145–205. https://doi.org/10.1017/S0956796806006198

Mathias Gaunard and Dietmar Kühl. 2015. Function Object-Based Overloading of Operator Dot. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0060R0. 18 Sept. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0060r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Michael Gibbs and Bjarne Stroustrup. 2006. Fast Dynamic Casting. *Software: Practice and Experience* 36, 2, 139–156. https://doi.org/10.1002/spe.686

GNUmake 2006–2020. GNUmake, a tool for generating executable code from multiple source files (website). http://www.gnu.org/software/make/ (also at Internet Archive 8 April 2020 02:17:10).
GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.

Matt Godbolt. 2016. Compiler Explorer, an online interactive environment for compiling C++ using a variety of compilers and seeing the assembly code they generate (blog post). 4 Sept. 2016. https://xania.org/201609/how-compiler-explorer-runs-on-amazon (also at Internet Archive 1 April 2020 22:07:49). https://godbolt.org

Nat Goodspeed. 2014. Stackful Coroutines and Stackless Resumable Functions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4232. 13 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4232.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Peter Gottschling. 2015. *Discovering Modern C++.* Addison-Wesley, Boston, Massachussets, USA. 978-0-13-438358-3

Douglas Gregor. 2006. A Brief Introduction to Variadic Templates. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2087. 10 Sept. 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2087.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Doug Gregor. 2010. Deprecating Exception Specifications. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3051. 12 March 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3051.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Douglas Gregor. 2012. Modules (slides). Nov. 2012. http://llvm.org/devmtg/2012-11/Gregor-Modules.pdf (also at Internet Archive 20 Dec. 2019 10:43:30). Video at https://youtu.be/586c_QMXir4 Meeting proceedings at http://llvm.org/devmtg/2012-11/ Presentation at 2012 LLVM Developers' Meeting, San Jose, California, USA.

Douglas Gregor and David Abrahams. 2009. Rvalue References and Exception Safety. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2855. 23 March 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2855.

html (also at Internet Archive 4 Sept. 2019 02:05:46).

Douglas Gregor, Jaakko Järvi, and Gary Powell. 2004. Variadic Templates. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1603. 17 Feb. 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1603.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: Linguistic Support for Generic Programming in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, Oct.) (*OOPSLA '06*). Association for Computing Machinery, New York, NY, USA, 291–310. *SIGPLAN Notices* 41, 10. 1595933484 0362-1340 https://doi.org/10.1145/1167473.1167499

Douglas Gregor and Andrew Lumsdaine. 2008. Core Concepts for the C++0x Standard Library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2502. 3 Feb. 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2502.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek. 2008. Proposed Wording for Concepts (Revision 6). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2676. 30 June 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2676.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Kate Gregory. 2015. Stop Teaching C (video). 25 Sept. 2015. NON-ARCHIVAL https://youtu.be/YnWhqhNdYyk (63 minutes). NON-ARCHIVAL https://cppcon2015.sched.com/event/3vm1/stop-teaching-c

To this day most people who set out to help others learn C++ start with "introduction to C" material. I think this actively contributes to bad C++ code in the world. For the past few years I've been teaching C++ (and making suggestions to folks who intend to teach themselves) in an entirely different way. No `char*` strings, no `strlen`, `strcmp`, `strcpy`, no `printf`, and no `[]` arrays. Pointers introduced very late. References before pointers, and polymorphism with references rather than with pointers. Smart pointers as the default pointer with raw pointers (whether from new or &) reserved for times they're needed. Drawing on the Standard Library sooner rather than later, and writing modern C++ from lesson 1.

Talk at CppCon: The C++ Conference.

Kate Gregory. 2017. 10 Core Guidelines You Need to Start Using Now (video). 27 Sept. 2017. NON-ARCHIVAL https://youtu.be/XkDEzfpdcSg (63 minutes). NON-ARCHIVAL https://cppcon2017.sched.com/event/Bgt1/10-core-guidelines-you-need-to-start-using-now

The C++ Core Guidelines were announced at CppCon 2015, yet some developers have still never heard of them. It's time to see what they have to offer for you, no matter how much C++ experience you have. You don't need to read and learn the whole thing: in this talk I am pulling out some highlights of the Guidelines to show you why you should be using these selected guidelines. For each one I'll show some examples, and discuss the benefit of adopting them for new code or going back into old code to make a change.

Talk at CppCon: The C++ Conference.

Kate Gregory. 2018. Simplicity: Not Just For Beginners (video). 26 Sept. 2018. NON-ARCHIVAL https://youtu.be/n0Ak6xtVXno (89 minutes). NON-ARCHIVAL https://cppcon2018.sched.com/event/FnKB/simplicity-not-just-for-beginners

Many people say that simple code is better code, but fewer put it into practice. In this talk I'll spend a little time on why simpler is better, and why we resist simplicity. Then I'll provide some specific approaches that are likely to make your code simpler, and discuss what you need to know and do in order to consistently write simpler code and reap the benefits of that simplicity.

Keynote talk at CppCon: The C++ Conference.

GTKmm 2005–2020. GTKmm: an open-source, cross-platform GUI library (used for GTK+ and Gnome) (website). http://www.gtkmm.org/en/ (also at Internet Archive 25 Jan. 2020 02:17:50).

gtkmm is the official C++ interface for the popular GUI library GTK+. Highlights include typesafe callbacks, and a comprehensive set of widgets that are easily extensible via inheritance. You can create user interfaces either in code or with the Glade User Interface designer, using Gtk::Builder. There's extensive documentation, including API reference and a tutorial.

Aleksey Gurtovoy and David Abrahams. 2002–2020. The Boost MPL Library (website). https://www.boost.org/doc/libs/1_73_0/libs/mpl/doc/index.html Archived at https://web.archive.org/web/*/https://www.boost.org/doc/libs/1_73_0/libs/mpl/doc/index.html

The Boost.MPL library is a general-purpose, high-level C++ template metaprogramming framework of compile-time algorithms, sequences and metafunctions. It provides a conceptual foundation and an extensive set of powerful and coherent tools that make doing explict metaprogramming in C++ as easy and enjoyable as possible within the current language.

Niklas Gustafsson. 2012. Resumable Functions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3328. 12 Jan. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3328.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard Hinnant, Roger Orr, Bjarne Stroustrup, David Vandevoorde, and Michael Wong. 2019. Direction for ISO C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0939R2. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0939r2.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard Hinnant, Roger Orr, Bjarne Stroustrup, David Vandevoorde, and Michael Wong. 2020. Direction for ISO C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P2000R0. 13 Jan. 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2000r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard E. Hinnant, Dave Abrahams, and Peter Dimov. 2004. A Proposal to Add an Rvalue Reference to the C++ Language. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1690. 7 Sept. 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno. 2008. A Foundation to Sleep On: Clocks, Points in Time, and Time Durations. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2615. 18 May 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2615.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard E. Hinnant, Peter Dimov, and Dave Abrahams. 2002. A Proposal to Add Move Semantics Support to the C++ Language. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1377. 10 Sept. 2002. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard E. Hinnant and Tomasz Kamiński. 2018. Extending <chrono> to Calendars and Time Zones. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0355R7. 16 March 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0355r7.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki. 2006. A Brief Introduction to Rvalue References. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2027. 12 June 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jared Hoberock (Ed.). 2019. Working Draft, C++ Extensions for Parallelism Version 2. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4796. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4796.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, David Hollman, and other contributors: Hans Boehm, Thomas Heller, Lee Howes, Bryce Lelbach, Hartmut Kaiser, Bryce Lelbach, Gor Nishanov, Thomas Rodgers, and Michael Wong. 2019. A Unified Executors Proposal for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0443R10. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0443r10.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, and Michael Wong. 2018. Executors Design Document. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0761R2. 12 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0761r2.pdf (also at Internet Archive 21 April 2020 17:01:39).

David S. Hollman. 2019. Experience Report: Implementing a Coroutines TS Frontend to an Existing Tasking Library Draft Proposal. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1403R0. 16 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1403r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Tom Honermann. 2017. Remove abbreviated functions and template-introduction syntax from the Concepts TS. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0696R0. 19 June 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0696r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Lee Howes, Eric Niebler, and Lewis Baker. 2018. In support of merging coroutines into C++20. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1241. 8 Oct. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1241r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jaakko Järvi. 2002. Proposal for adding tuple types into the standard library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1382. 10 Sept. 2002. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1382.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Jaakko Järvi and Gary Powell. 2002. Boost.Lambda: The Boost Lambda Library (website). https://www.boost.org/doc/libs/1_72_0/doc/html/lambda.html (also at Internet Archive 29 March 2020 02:26:54). http://boost.org/libs/lambda

The Boost Lambda Library (BLL in the sequel) is a C++ template library, which implements a form of lambda abstractions for C++. The term originates from functional programming and lambda calculus, where a lambda abstraction defines an unnamed function. The primary motivation for the BLL is to provide flexible and convenient means to define unnamed function objects for STL algorithms.

Chapter 20 of the Boost Library Documentation.

Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. 2003a. The Lambda Library: Unnamed functions in C++. *Software: Practice and Experience* 33, 3, 259–291. https://doi.org/10.1002/spe.504

Jaakko Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. 2007. Decltype (revision 7): proposed wording. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2343. 18 July 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2343.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Jaakko Järvi, Bjarne Stroustrup, Doug Gregor, and Jeremy Siek. 2003b. Decltype and auto. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1478. 28 April 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

JetBrains. 2020. CLion: A cross-platform IDE for C and C++ (website). https://www.jetbrains.com/clion/ (also at Internet Archive 10 April 2020 00:41:01).

Smart C and C++ editor: Thanks to native C and C++ support, including modern C++ standards, libc++ and Boost, CLion knows your code through and through and takes care of the routine while you focus on the important things.

Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (July), 1702–1714. 2150-8097 https://doi.org/10.14778/3236187.3236216

Nicolai Josuttis, Lewis Baker, Billy O'Neal, Herb Sutter, and Anthony Williams. 2019a. Stop Token and Joining Thread, Rev 9. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0660R9. 10 March 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0660r9.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Nicolai Josuttis, Ville Voutilainen, Roger Orr, Daveed Vandevoorde, John Spicer, and Christopher Di Bella. 2019b. Remove Contracts from C++20. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1823R0. 18 July 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1823r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. 2009Sept.. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *38th International Conference on Parallel Processing Workshops (ICPPW 2009)* (Vienna, Austria). IEEE Computer Society, Los Alamitos, California, USA, 394–401. 978-0-7695-3803-7 https://doi.org/10.1109/ICPPW.2009.14

D. Kapur, D. R. Musser, and A. A. Stepanov. 1981. Operators and Algebraic Structures. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (Portsmouth, New Hampshire, USA) (*FPCA '81*). Association for Computing Machinery, New York, NY, USA, 59–64. 0897910605 https://doi.org/10.1145/800223.806763

Anastasia Kazakova. 2015. Infographic: C/C++ Facts We Learned Before Going Ahead with CLion (blog post). 27 July 2015. http://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/ (also at Internet Archive 28 Jan. 2020 05:02:34).

Erich Keane, Adam David Alan Martin, and Allan Deutsch. 2017. A Case for Simplifying/Improving Natural Syntax Concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0782R0. 25 Sept. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0782r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Brian Kernighan and Dennis M. Ritchie. 1978. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, USA. 978-0131101630

Rostislav Khlebnikov and John Lakos. 2019. Contracts, Undefined Behavior, and Defensive Programming. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1743R0. 17 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1743r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

The Khronos Group. 2014–2020. C++ Single-source Heterogeneous Programming for OpenCL (website). https://www.khronos.org/sycl/ Archived at https://web.archive.org/web/*/https://www.khronos.org/sycl/

SYCL (pronounced 'sickle') is a royalty-free, cross-platform abstraction layer that builds on the underlying concepts, portability and efficiency of OpenCL that enables code for heterogeneous processors to be written in a "single-source" style using completely standard C++. SYCL single-source programming enables the host and kernel code for an application to be contained in the same source file, in a type-safe way and with the simplicity of a cross-platform asynchronous task graph. SYCL includes templates and generic lambda functions to enable higher-level application software to be cleanly coded with optimized acceleration of kernel code across the extensive range of shipping OpenCL 1.2 implementations.

Andrew Koenig and Bjarne Stroustrup. 1989. Exception Handling for C++. In *Proceedings of the USENIX "C++ at Work" Conference* (Nov.). USENIX Association, Berkeley, California, USA, 31 pages. http://www.rajatorrent.com.stroustrup.com/except89.pdf (also at Internet Archive 11 April 2020 01:00:36).

Andrew Koenig and Bjarne Stroustrup. 1991a. Analysis of Overloaded operator.(). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N0054. 21 Sept. 1991. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1991/WG21%201991/X3J16_91-0121%20WG21_N0054.pdf (also at Internet Archive 12 April 2016 17:17:26).

Andrew Koenig and Bjarne Stroustrup. 1991b. C++: As close to C as possible but no closer. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N0089. May 1991. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1991/WG21%201991/X3J16_91-0089%20WG21_N0007.pdf (also at Internet Archive 4 Sept. 2019 02:05:46). Also bears the document number N0007 and the date 11 May 1989.

Andrew R. Koenig (Ed.). 1998. *ISO/IEC 14882:1998: Programming languages — C++.* ISO (International Organization for Standardization), Geneva, Switzerland (Sept.). 732 book pages. https://www.iso.org/standard/25845.html Status: withdrawn.

Christopher Kohlhoff. 2005. Boost ASIO (website). https://www.boost.org/doc/libs/1_67_0/doc/html/boost_asio.html (also at Internet Archive 11 April 2020 01:04:37).

Boost.Asio is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach.

Christopher Kohlhoff. 2006. Networking Library Proposal for TR2. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2054. 8 Sept. 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2054.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Christopher Kohlhoff. 2013. A Universal Model for Asynchronous Operations. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3747. 1 Sept. 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3747.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Christopher M. Kohlhoff. 2018. ASIO C++ Library (website). http://think-async.com/Asio/ (also at Internet Archive 23 March 2020 15:37:13).

Asio is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach.

Thomas Köppe. 2016a. Hexadecimal floating literals for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0245R0. 9 Feb. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0245r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Thomas Köppe. 2016b. Selection statements with initializer. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0305R1. 24 June 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0305r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Thomas Köppe. 2017a. An Adjective Syntax for Concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0807R0. 12 Oct. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0807r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Thomas Köppe. 2017b. Allow lambda capture [=, this]. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0409R2. 4 March 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0409r2.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Thomas Köppe. 2017c. Range-based for statements with initializer. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0614R1. 6 Nov. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0614r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Oliver Kowalke. 2015. A low-level API for stackful coroutines. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4397. 9 April 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4397.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Oliver Kowalke and Nat Goodspeed. 2013. A proposal to add coroutines to the C++ standard library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3708. 4 March 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3708.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

John Lakos. 2018. "Avoiding undefined behavior in contracts" [P1290R0] Explained. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1335R0. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1335r0.txt (also at Internet Archive 4 Sept. 2019 02:05:46).

John Lakos. 2019. United Amendment to Contracts Facility for C++20. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1486R1. 21 Feb. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1486r1.txt (also at Internet Archive 4 Sept. 2019 02:05:46).

John Lakos and Alexei Zakharov. 2013. Centralized Defensive-Programming Support for Narrow Contracts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3604. 18 March 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3604.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Leslie Lamport. 1978. Time, Clocks and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July), 558–565. 0001-0782 https://doi.org/10.1145/359545.359563

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sept.), 690–691. 2326-3814 https://doi.org/10.1109/TC.1979.1675439

Bryce Adelstein Lelbach, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen. 2019. The C++20 Synchronization Library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1135R4. 4 March 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1135r4.html (also at Internet Archive 4 Sept. 2019 02:05:46).

LLVM 2003–2020. LLVM, a collection of tools and libraries for building compilers and tools (website). https://llvm.org/ (also at Internet Archive 7 April 2020 15:19:33).

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research.

Lockheed Martin Corporation. 2005. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.* Lockheed Martin Corporation (Dec.). http://stroustrup.com/JSF-AV-rules.pdf (also at Internet Archive 8 Jan. 2020 08:09:03). Document Number 2RDU00001 Rev C.

Bruno Cardoso Lopes, Michael Spencer, and JF Bastien. 2019. Modules Feedback. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1482R0. 8 Feb. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1482r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Macaulay2 2005–2020. Macaulay2 (website). http://www2.macaulay2.com/Macaulay2/ (also at Internet Archive 21 Sept. 2019 08:34:37).

Macaulay2 is a software system devoted to supporting research in algebraic geometry and commutative algebra, whose creation has been funded by the National Science Foundation since 1992.

Macaulay2 includes core algorithms for computing Gröbner bases and graded or multi-graded free resolutions of modules over quotient rings of graded or multi-graded polynomial rings with a monomial ordering. The core algorithms are accessible through a versatile high level interpreted user language with a powerful debugger supporting the creation of new classes of mathematical objects and the installation of methods for computing specifically with them Macaulay2 can compute Betti numbers, Ext, cohomology of coherent sheaves on projective varieties, primary decomposition of ideals, integral closure of rings, and more.

John Maddock. 2002. A Proposal to add Regular Expressions to the Standard Library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1386. 6 Sept. 2002. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1386.htm (also at Internet Archive 4 Sept. 2019 02:05:46).

Jens Maurer. 2002. A Proposal to Add an Extensible Random Number Facility to the Standard Library. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1398. 10 Nov. 2002. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1398.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jens Maurer. 2012. Allowing arbitrary literal types for non-type template parameters. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3413. 19 Sept. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3413.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jens Maurer. 2019. Bit operations. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0553R4. 1 March 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0553r4.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jens Maurer and Michael Wong. 2007. Towards support for attributes in C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2236. 4 May 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2236.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Maurer, Alisdair Meredith, Bjarne Stroustrup, and David Vandevoorde. 2008. User-defined Literals (aka. Extensible Literals (revision 5)). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2765. 18 Sept. 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2765.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Paul McJones (Ed.). 2007–2020. *C++ Historical Sources Archive* (website). Computer History Museum Software Preservation Group. http://www.softwarepreservation.org/projects/c_plus_plus/ (also at Internet Archive 29 March 2020 16:45:18).

This is a collection of design documents, source code, and other materials concerning the birth, development, standardization, and use of the C++ programming language.

C++ added January 2007.

Paul E. McKenney, Mark Batty, Clark Nelson, Hans Boehm, Anthony Williams, Scott Owens, Susmit Sarkar, Peter Sewell, Tjark Weber, Michael Wong, Lawrence Crowl, and Benjamin Kosnik. 2010. Omnibus Memory Model and Atomics Paper. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3196. 11 Nov. 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3196.htm (also at Internet Archive 4 Sept. 2019 02:05:46).

Alisdair Meredith. 2007. Seeking a Syntax for Attributes in C++09. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2224. 12 March 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2224.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Alisdair Meredith. 2012. Call for Library Proposals. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3370. 26 Feb. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3370.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Alisdair Meredith and Herb Sutter. 2017. Revising atomic_shared_ptr for C++20. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0718R2. 10 Nov. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0718r2.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Bertrand Meyer. 1994. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, USA. 534 book pages. 978-0136290490

Scott Meyers. 2014. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14.* O'Reilly Media, Sebastopol, California. 978-1491903995

Microsoft. 2020. Visual Studio: Best-in-class tools for any developer (website). https://visualstudio.microsoft.com/ (also at Internet Archive 13 April 2020 16:16:48).
Visual Studio: Full-featured IDE to code, debug, test, and deploy to any platform
Visual Studio Code: Editing and debugging on any OS
Visual Studio for Mac: Develop apps and games for iOS, Android, and web using .NET

Mihail Mihaylov and Vassil Vassilev. 2018. On the Coroutines TS. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1329R0. 2 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1329r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Mihail Mihaylov and Vassil Vassilev. 2019. First-class symmetric coroutines in C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1430R0. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1430r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

William M. Miller. 2010. Rvalue References as "Funny" Lvalues. Edison Design Group. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3030. 14 Feb. 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3030.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Lev Minkovsky and John McFarlane. 2019. Math Constants. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0631R7. 24 May 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0631r7.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Mobileye. 2020. ADAS (Advanced Driver Assistance Systems) (website). https://www.mobileye.com/our-technology/adas/ (also at Internet Archive 13 Jan. 2020 03:29:15). Mobileye is an Intel company.

Sergei Murzin, Michael Park, David Sankel, and Dan Sarginson. 2019. Pattern Matching. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1371R0. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1371r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Sergei Murzin, Michael Park, David Sankel, and Dan Sarginson. 2020. Pattern Matching. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1371R2. 13 Jan. 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

David R. Musser and Alexander A. Stepanov. 1987. A Library of Generic Algorithms in Ada. In *Proceedings of the 1987 Annual ACM SIGAda International Conference on Ada* (Boston, Massachusetts, USA, Dec.) (*SIGAda '87*). Association for Computing Machinery, New York, NY, USA, 216–225. 0897912438 https://doi.org/10.1145/317500.317529 Also at NON-ARCHIVAL http://stepanovpapers.com/p216-musser.pdf (retrieved 28 Feb. 2020)

Axel Naumann. 2012. Introducing cling, a C+++ Interpreter Based on clang/LLVM (video). 15 March 2012. https://youtu.be/f9Xfh8pv3Fs Google Tech Talk (48 minutes).

Axel Naumann, Philippe Canal, Paul Russo, and Vassil Vassilev. 2010. Creating cling, an interactive interpreter interface for clang (video). 4 Nov. 2010. http://llvm.org/devmtg/2010-11/ (also at Internet Archive 2 April 2020 04:03:23). Video at https://youtu.be/BjmGOMJWeAo Slides at http://llvm.org/devmtg/2010-11/Naumann-Cling.pdf Talk at 2010 LLVM Developers' Meeting (26 minutes).

Peter Naur. 1966. Proof of Algorithms by General Snapshots. *BIT (Nordisk Tidskrift för Informationsbehandling) Numerical Mathematics* 6, 4, 310–316. https://doi.org/10.1007/BF01966091

Eric Niebler and Casey Carter. 2017. C++ Extensions for Ranges. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4651. 15 March 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4651.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Eric Niebler, Casey Carter, and Christopher Di Bella. 2018. The One Ranges Proposal (was Merging the Ranges TS). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0896R2. 25 June 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r2.pdf (also at Internet Archive 21 April 2020 20:59:57).

Eric Niebler, Sean Parent, and Andrew Sutton. 2014. Ranges for the Standard Library, Revision 1. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4128. 10 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Gor Nishanov. 2017. *ISO/IEC TS 22277:2017: Technical Specification — C++ Extensions for Coroutines.* ISO (International Organization for Standardization), Geneva, Switzerland (Nov.). 18 book pages. https://www.iso.org/standard/73008.html

Gor Nishanov. 2018. Incremental Approach: Coroutine TS + Core Coroutines. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1362R0. 15 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1362r0.pdf (also at Internet Archive 21 April 2020 21:01:42).

Gor Nishanov. 2019a. C++ Exception Optimizations. An experiment. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1676R0. 4 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1676r0.pdf (also at Internet

Archive 4 Sept. 2019 02:05:46).

Gor Nishanov. 2019b. Response to response to "Fibers under the magnifying glass". ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1520R0. 8 March 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1520r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

NVIDIA. 2020. Advanced Driver Assistance Systems (ADAS) (website). https://www.nvidia.com/en-us/self-driving-cars/adas/ (also at Internet Archive 10 April 2020 17:56:28).

Günter Obiltschnig et al. 2005. POCO C++ Libraries (website). http://pocoproject.org/ (also at Internet Archive 3 March 2020 05:39:34).

The POCO C++ Libraries are powerful cross-platform C++ libraries for building network- and internet-based applications that run on desktop, server, mobile, IoT, and embedded systems.

Initial release: February 21, 2005.

Objective C++ Wikipedia 2020. Objective C++ (a language variant accepted by GNU GCC and Clang). https://en.wikipedia.org/wiki/Objective-C#Objective-C++ (also at Internet Archive 9 April 2020 21:16:54).

OpenCV 2020. OpenCV (website). https://opencv.org/about/ (also at Internet Archive 25 March 2020 14:29:00).

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms.

OpenCV Wikipedia 2020. OpenCV (Open source Computer Vision). https://en.wikipedia.org/wiki/OpenCV (also at Internet Archive 3 April 2020 14:24:23).

Thorsten Ottosen. 2005. Proposal for new for-loop. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1796. 27 April 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1796.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Thorsten Ottosen, Lawrence Crowl, and Douglas Gregor. 2007. Wording for range-based for-loop (revision 1). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2196. 7 March 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2196.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. 2010. Design and Evaluation of C++ Open Multi-Methods. *Science of Computer Programming* 75, 7 (July), 638–667. 0167-6423 https://doi.org/10.1016/j.scico.2009.06.002 Accepted June 2009. Expanded version of a paper presented at the conference on Generative Programming and Component Engineering (GPCE 2007).

Gary Powell, Doug Gregor, and Jaakko Järvi. 2004. Overloading Operator.() & Operator.*(). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1671. 10 Sept. 2004. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1671.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. In *Proceedings of the 44th International Conference on Very Large Data Bases* (Rio de Janeiro, Brazil, 31 Aug.). VLDB Endowment Inc., Los Angeles, California, 230–242. *Proceedings of the VLDB Endowment* 11, 2. https://doi.org/10.14778/3149193.3149202

William Pugh. 2004. JSR 133: Java Memory Model and Thread Specification Revision (website). Sept. 2004. http://jcp.org/en/jsr/detail?id=133 (also at Internet Archive 29 Sept. 2019 09:53:52).

The proposed specification describes the semantics of threads, locks, volatile variables and data races. This includes what has been referred to as the Java memory model.

Qt 1991–2020. Qt, a platform for GUI and related software (website). http://www.qt.io/ (also at Internet Archive 3 April 2020 19:32:36).

One framework. One codebase. Any platform. Everything you need for your entire software development life cycle. Qt is the fastest and smartest way to produce industry-leading software that users love.

Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2011. Formal Verification of Object Layout for C++ Multiple Inheritance. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (*POPL '11*). Association for Computing Machinery, New York, NY, USA, 67–80. 978-1450304900 https://doi.org/10.1145/1926385.1926395

Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2012. A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) (*POPL '12*). Association for Computing Machinery, New York, NY, USA, 521–532. 978-1450310833 https://doi.org/10.1145/2103656.2103718

RC++ 2010–2020. RC++, a system for runtime-compiled C++: Alternatives (website). https://github.com/RuntimeCompiledCPlusPlus/RuntimeCompiledCPlusPlus/wiki/Alternatives (also at Internet Archive 27 April 2015 23:18:58).

Runtime-Compiled C++ (RCC++) enables you to reliably make major changes to your C++ code at runtime and see the results immediately. The technique is aimed at games development but can also be useful in any industry where turnaround times are a bottleneck.
We believe RCC++ is a good overall standard C++ approach to runtime editing of code which is easy to use, available across a number of platforms and relatively easy to port. There are however a few alternative implementations for compiling C++ at runtime.

James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ For Multi-Core Processor Parallelism.* O'Reilly Media, Sebastopol, California (July). 344 book pages. 978-0596514808 http://shop.oreilly.com/product/9780596514808.do Code repository at http://threadingbuildingblocks.org/

James Renwick, Tom Spink, and Björn Franke. 2019. Low-Cost Deterministic C++ Exceptions for Embedded Systems. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA, Feb.) (*CC 2019*). Association for Computing Machinery, New York, NY, USA, 76–86. 978-1450362771 https://doi.org/10.1145/3302516.3307346

Vincent Reverdy. 2012. A proposal to add special mathematical functions according to the ISO/IEC 80000-2:2009 standard. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3494. 19 Dec. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3494.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Barry Revzin. 2017. Allow pack expansion in lambda init-capture. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0780R0. 8 Oct. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0780r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Barry Revzin. 2018. `<=> != ==`. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1185R0. 7 Oct. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1185r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Barry Revzin. 2019. Spaceship needs a tune-up: Addressing some discovered issues with P0515 and P1185. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1630R1. 17 July 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1630r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Barry Revzin and Stephan T. Lavavej. 2018. explicit(bool). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0892R2. 8 June 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0892r2.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jakob Riedle. 2017. Concepts are Adjectives, not Nouns. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0791R0. 10 Oct. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0791r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Torvald Riegel. 2015. On unifying the coroutines and resumable functions proposals. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0073R0. 25 Sept. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0073r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Dennis M. Ritchie. 1990. Variable-Size Arrays in C. *Journal of C Language Translation* 2, 2 (Sept.), 81–86. http://jclt.iecc.com/Jct22.pdf (also at Internet Archive 2 April 2016 11:29:25). See also https://www.bell-labs.com/usr/dmr/www/vararray.pdf

Rene Rivera. 2019a. Are modules fast? (revision 0). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1441R0. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1441r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Rene Rivera. 2019b. Are modules fast? (revision 1). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1441R1. 6 March 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1441r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Geoff Romer, James Dennett, and Chandler Carruth. 2018. Core Coroutines: Making coroutines simpler, faster, and more general. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1063R0. 6 May 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1063r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Geoff Romer, James Dennett, and Chandler Carruth. 2019a. Core Coroutines: Making coroutines simpler, faster, and more general. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1063R2. 16 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1063r2.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Geoffrey Romer, Gor Nishanov, Lewis Baker, and Mihail Mihailov. 2019b. Coroutines: Use-cases and Trade-offs. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1493R0. 19 Feb. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1493r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Hyman Rosen, John Lakos, and Alisdair Meredith. 2019. Adding a global contract assumption mode. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1730R0. 14 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1730r0.md (also at Internet Archive 4 Sept. 2019 02:05:46).

David Sankel (Ed.). 2018. Working Draft, C++ Extensions for Reflection. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4766. 11 Aug. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4766.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

David Sankel and Daveed Vandevoorde. 2019. User-friendly and Evolution-friendly Reflection: A Compromise Document. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1733R0. 15 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1733r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Tim Shen, Richard Smith, Zhihao Yuan, and Chandler Carruth. 2016. Designated Initialization. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0329R0. 9 May 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0329r0.pdf (also at Internet Archive 21 April 2020 21:37:11).

Nathan Sidwell. 2018. Module Preamble is Unnecessarily Fragile (re N4720 Merging Modules). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1299R3. 13 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1299r3.pdf (also at Internet Archive 21 April 2020 21:39:18).

Nathan Sidwell. 2019. Make Me A Module (re P1184 A Module Mapper). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1602R0. 1 March 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1602r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Nathan Sidwell and Davis Herring. 2019. Modules: ADL & Internal Linkage (re P1103R1 Merging Modules). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1347R1. 17 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1347r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Jeremy Siek and Andrew Lumsdaine. 2000–2007. The Boost Concept Check Library (BCCL) (website). https://www.boost.org/doc/libs/1_69_0/libs/concept_check/concept_check.htm (also at Internet Archive 16 Feb. 2019 12:45:11).

The Concept Check library allows one to add explicit statement and checking of concepts in the style of the proposed C++ language extension.

The mechanisms use standard C++ and introduce no run-time overhead. The main cost of using the mechanism is in compile-time.

Jeremy Siek and Walid Taha. 2006. A Semantic Analysis of C++ Templates. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)* (Nantes, France). Springer-Verlag, Berlin and Heidelberg, Germany, 304–327. 3540357262 https://doi.org/10.1007/11785477_19 LNCS 4067.

Cleiton Santoia Silva and Daniel Auresco. 2014. C++ type reflection via variadic template expansion. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3951. 7 Feb. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3951.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith. 2013. Relaxing constraints on constexpr functions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3597. 15 March 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3597.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith. 2015. Guaranteed copy elision through simplified value categories. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0135R0. 27 Sept. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith (Ed.). 2017. *ISO/IEC 14882:2017: Programming languages — C++*. ISO (International Organization for Standardization), Geneva, Switzerland (Dec.). 1605 book pages. https://www.iso.org/standard/68564.html

Richard Smith. 2018a. Another take on Modules. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0947R0. 12 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0947r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith. 2018b. Another take on Modules (Revision 1). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0947R1. 6 March 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0947r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith. 2018c. Towards consistency between <=> and other comparison operators. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0946R0. 10 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0946r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith. 2019. Merging Modules. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1103R3. 22 Feb. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1103r3.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith (Ed.). 2020. *Working Draft, Standard for Programming Language C++*. Number N4861. (April). 1834 book pages. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4861.pdf (also at Internet Archive 27 April 2020 14:58:44).

Richard Smith and Gabriel Dos Reis. 2018. Merging Modules. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1103R0. 22 June 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1103r0.pdf (also at Internet Archive 21 April 2020 22:21:27).

Richard Smith and Gor Nishanov. 2018. Halo: coroutine Heap Allocation eLision Optimization: The joint response. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0981R0. 18 March 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith and Andrew Sutton. 2017. Semantic constraint matching for concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0717R0. 19 June 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0717r0.

pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith, Andrew Sutton, and Daveed Vandevoorde. 2018a. Immediate functions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1073R3. 6 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1073r3.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith, Andrew Sutton, and Daveed Vandevoorde. 2018b. `std::is_constant_evaluated()`. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0595R2. 9 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0595r2.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Richard Smith, Daveed Vandevoorde, Geoffrey Romer, Gor Nishanov, Nathan Sidwell, Iain Sandoe, and Lewis Baker. 2019. Coroutines: Language and Implementation Impact. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1492R0. 19 Feb. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1492r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Oleg Smolsky. 2014. Defaulted comparison operators. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3950. 19 Feb. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3950.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Jeff Snyder and Chandler Carruth. 2013. Call for Compile-Time Reflection Proposals. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3814. 6 Oct. 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3814.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. 2013. Open Pattern Matching for C++. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity* (Indianapolis, Indiana, USA, Oct.) (*SPLASH '13*). Association for Computing Machinery, New York, NY, USA, 97–98. 978-1450319959 https://doi.org/10.1145/2508075.2508098

Yuriy Solodkyy, Gabriel Dos Reisl, and Bjarne Stroustrup. 2012. Open and Efficient Type Switch for C++. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA, Oct.) (*OOPSLA '12*). Association for Computing Machinery, New York, NY, USA, 963–982. *SIGPLAN Notices* 47, 10 (Nov. 2012). 978-1450315616 https://doi.org/10.1145/2384616.2384686

Yuriy Solodkyy, Gabriel Dos Reisl, and Bjarne Stroustrup. 2014. Pattern Matching for C++. Nov. 2014. http://wiki.edg.com/pub/Wg21urbana-champaign/EveningSessions/pattern_matching.pdf Evening session at WG21 meeting in Urbana-Champaign, Illinois.

Mike Spertus. 2018. Extensions to Class Template Argument Deduction. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1021R0. 7 May 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1021r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Mike Spertus, Timur Doumler, and Richard Smith. 2018. Filling holes in Class Template Argument Deduction. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1021R3. 26 Nov. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1021r3.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Mike Spertus and Richard Smith. 2015. Template parameter deduction for constructors (Rev. 3). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0091R0. 24 Sept. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

The Stellar Group. 2014–2020. HPX (High Performance ParalleX) (website). http://stellar.cct.lsu.edu/projects/hpx/ (also at Internet Archive 27 Dec. 2019 05:13:30).

HPX (High Performance ParalleX) is a general purpose C++ runtime system for parallel and distributed applications of any scale. It strives to provide a unified programming model which transparently utilizes the available resources to achieve unprecedented levels of scalability. This library strictly adheres to the C++11 Standard and leverages the Boost C++ Libraries which makes HPX easy to use, highly optimized, and very portable. HPX is developed for conventional architectures including Linux-based systems, Windows, Mac, and the BlueGene/Q, as well as accelerators such as the Xeon Phi. High Performance ParalleX is the first open-source implementation of the ParalleX execution model.

Alexander Stepanov. 1986. Scheme higher order programming library (code repository plus documentation). Aug. 1986. http://stepanovpapers.com/schemenotes/index.html (also at Internet Archive 16 Aug. 2018 07:27:46).

Alex Stepanov and Paul McJones. 2009. *Elements of Programming*. Addison-Wesley, Reading, Massachusetts, USA. 288 pages. 978-0321635372

Arthur G. Stephenson et al. 1999. *Mars Climate Orbiter Mishap Investigation Board Phase I Report.* Technical Report. NASA (10 Nov.). https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf (also at Internet Archive 5 March 2020 02:00:59).

Bjarne Stroustrup. 1982. Classes: An Abstract Data Type Facility for the C Language. *SIGPLAN Notices* 17, 1 (Jan.), 42–51. 0362-1340 https://doi.org/10.1145/947886.947893 Published internally in Bell Labs in 1981.

Bjarne Stroustrup. 1985a. AT&T C++ Translator Release Notes. Nov. 1985. Archived at the Computer History Museum: http://www.softwarepreservation.org/projects/c_plus_plus/ (also at Internet Archive 29 March 2020 16:45:18). Contains the manual and tutorials (e.g., for stream I/O and tasks).

Bjarne Stroustrup. 1985b. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, USA. 0-201-12078-X

Bjarne Stroustrup. 1985c. A Set of C++ Classes for Co-routine Style Programming. In *C++ Translator Release Notes*. AT&T, New York, NY, USA (Nov.). http://www.softwarepreservation.org/projects/c_plus_plus/cfront/release_e/doc/ClassesForCoroutines.pdf (also at Internet Archive 18 July 2019 02:33:30). First published internally in 1980.

Bjarne Stroustrup. 1990–2020. My C++ Standards committee papers (website with links to documents). http://www.stroustrup.com/WG21.html (also at Internet Archive 20 Feb. 2020 04:39:09).

This is an incomplete list of papers that I (Bjarne Stroustrup) have written or co-authored for the C++ standards committee. Please remember that these are incomplete proposals, discussions of alternatives, and exploration of the design space. Note also that these papers were all written to a specific audience (the members of the C++ standards committee), at a specific time (relative to the then current draft of the standard), and as part of a specific discussion. They are typically not ideal introcutions for a general audience. Not all has led to or will lead to changes to C++, and most of those will "mutate" significantly as they progress from idea/proposal to final language change.

Bjarne Stroustrup. 1991. *The C++ Programming Language (2th Edition)*. Addison-Wesley, Reading, Massachusetts, USA. 0-201-53992-6

Bjarne Stroustrup. 1993. A History of C++. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA, March) (*HOPL-II*). Association for Computing Machinery, New York, NY, USA, 271–297. 0897915704 https://doi.org/10.1145/154766.155375 Also published in *History of Programming Languages* (edited by T. J. Bergin and R. G. Gibson), Addison-Wesley, ISBN 1-201-89502-1, 1996.

Bjarne Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, USA. 0-201-54330-3

Bjarne Stroustrup. 1997. *The C++ Programming Language (3rd Edition)*. Addison-Wesley, Reading, Massachusetts, USA. 0-201-88954-4

Bjarne Stroustrup. 1998. Generalizing Overloading for C++2000. *Overload Journal* 25 (1 April), 20–24. 1354-3172 https://accu.org/var/uploads/journals/overload25.pdf (also at Internet Archive 13 May 2011 09:37:09).

Bjarne Stroustrup. 2003. Concept checking — A more abstract complement to type checking. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1510. 22 Oct. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1510.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2004–2020. Why can't I define constraints for my template parameters? (website). http://www.stroustrup.com/bs_faq2.html#constraints (also at Internet Archive 10 April 2020 05:20:05).

Bjarne Stroustrup. 2005. A rationale for semantically enhanced library languages. In *Proceedings of the Workshop on Library-Centric Software Design (LCSD'05), co-located with OOPSLA 2005* (San Diego, California, USA, 16 Oct.). Department of Computer Science & Engineering, Texas A&M University, College Station, Texas, USA, 44. Archived at https://web.archive.org/web/20051118062506/http://lcsd05.cs.tamu.edu/papers/stroustrup.pdf

Bjarne Stroustrup. 2007. Evolving a Language in and for the Real World: C++ 1991–2006. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California, June) (*HOPL III*). Association for Computing Machinery, New York, NY, USA, 4–1–4–59. 978-1595937667 https://doi.org/10.1145/1238844.1238848

Bjarne Stroustrup. 2008a. *Programming – Principles and Practice Using C++*. Addison-Wesley, Boston, Massachusetts, USA. 978-0321543721 There is also a second edition, ISBN 978-0321992789, 2014.

Bjarne Stroustrup. 2008b. Uniform initialization design choices (Revision 2). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2532. 2 Feb. 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2532.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2009a. The C++0x "Remove Concepts" Decision. *Dr. Dobb's Journal* (July). https://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111 (also at Internet Archive 9 Oct. 2012 01:02:52).

Bjarne Stroustrup. 2009b. The C++0x "Remove Concepts" Decision. *Overload Journal* 92 (Aug.). https://accu.org/index.php/journals/1576 (also at Internet Archive 4 March 2010 22:57:32). Reprinted with permission from Dr. Dobb's Journal.

Bjarne Stroustrup. 2009c. Simplifying the use of concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2906. 21 June 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2906.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2010a. "New" Value Terminology. April 2010. http://stroustrup.com/terminology.pdf (also at Internet Archive 2 Oct. 2012 20:01:36).

Bjarne Stroustrup. 2010b. What Should We Teach New Software Developers? Why? *Communications of the ACM* 53, 1 (Jan.), 40–42. 0001-0782 https://doi.org/10.1145/1629175.1629192

Bjarne Stroustrup. 2012. Software Development for Infrastructure. *Computer* 45, 1 (Jan.), 47–58. 0018-9162 https://doi.org/10.1109/MC.2011.353

Bjarne Stroustrup. 2013. *The C++ Programming Language (4th Edition)*. Addison-Wesley, Boston, Massachusetts, USA. 978-0321563842

Bjarne Stroustrup. 2014a. Call syntax: x.f(y) vs. f(x,y). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4174. 11 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4174.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2014b. Default comparisons. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4175. 11 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4175.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2014c. Thoughts about Comparisons. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4176. 11 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4176.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2014d. *A Tour of C++*. Addison-Wesley, Boston, Massachusetts, USA. 978-0321958310

Bjarne Stroustrup. 2015a. Thoughts about C++17. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4492. 15 May 2015. http://open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4492.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2015b. Unified call syntax concerns. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0131R0. 27 Sept. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0131r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2017a. Concepts: The Future of Generic Programming; or, How to Design Good Concepts and Use Them Well. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0557R1. 31 Jan. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0557r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2017b. Function declarations using concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0694R0. 18 June 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0694r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2017c. Learning and Teaching Modern C++ (video). 25 Sept. 2017. https://youtu.be/fX2W3nNjJIo Opening keynote for CppCon (the C++ Conference), Bellevue, Washington, USA (99 minutes).

Bjarne Stroustrup. 2018a. The Evils of Paradigms; or, Beware of one-solution-fits-all thinking. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0976R0. 6 March 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0976r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2018b. A minimal solution to the concepts syntax problems. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1079. 6 May 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1079r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2018c. Modules and macros. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0955R0. 11 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2018d. Remember the Vasa! ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0977R0. 6 March 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0977r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2018e. Subscripts and sizes should be signed. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1428R0. 18 Jan. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1428r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2018f. *A Tour of C++ (Second Edition)*. Addison-Wesley, Boston, Massachusetts, USA. 978-0134997834

Bjarne Stroustrup. 2018g. What do we want to do with reflection? ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0954R0. 11 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0954r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2019a. C++ exceptions and alternatives. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1947R0. 18 Nov. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1947r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2019b. How can you be so certain? ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1962R0. 18 Nov. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1962r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup. 2019c. What to do about contracts? ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1711R0. 13 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1711r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup et al. 1992. How to Write a C++ Language Extension Proposal for ANSI-X3J16/ISO-WG21. *SIGPLAN Notices* 27, 6 (June), 64–71. 0362-1340 https://doi.org/10.1145/130981.130989

Bjarne Stroustrup and Gabriel Dos Reis. 2003a. Concepts — Design choices for template argument checking. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1522. 22 Oct. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1522.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Gabriel Dos Reis. 2003b. Concepts – Syntax and composition. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1536. 22 Oct. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1536.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Gabriel Dos Reis. 2003c. Templates aliases for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1489. 16 Sept. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1489.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Gabriel Dos Reis. 2005a. A concept design (Rev. 1). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1782. April 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1782.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Gabriel Dos Reis. 2005b. Initialization and initializers. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1890. 22 Sept. 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1890.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Gabriel Dos Reis. 2014. Operator Dot. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4173. 11 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4173.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Gabriel Dos Reis. 2016. Operator Dot (R3). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0416R1. 16 Oct. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0416r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup and Herb Sutter (Eds.). 2014–2020. C++ Core Guidelines (online document repository). http://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md (also at Internet Archive 24 Feb. 2020 12:59:15). Continually updated.

Bjarne Stroustrup and Herb Sutter. 2015. Unified Call Syntax: x.f(y) and f(x,y). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4474. 12 April 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4474.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Bjarne Stroustrup, Herb Sutter, and Gabriel Dos Reis. 2015. A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. Oct. 2015. http://www.stroustrup.com/resource-model.pdf (also at Internet Archive 10 April 2020 05:20:23). Revised Dec. 2015.

Bjarne Stroustrup and Andrew Sutton (Eds.). 2012. A Concept Design for the STL. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3351. 13 Jan. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2002. Proposed Addition to C++: Typedef Templates. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1406. 21 Oct. 2002. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1406.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2012. async and ~future. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3451. 23 Sept. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3451.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2013a. ~thread Should Join. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3636. 17 April 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3636.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2013b. AAA Style (Almost Always Auto) (blog post). 12 Aug. 2013. http://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/ (also at Internet Archive 3 Dec. 2019 14:58:42).

Herb Sutter. 2014. Unified Call Syntax. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4165. 4 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4165.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2017a. Consistent comparison. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0515R0. 5 Feb. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2017b. Merge Concurrency TS atomic pointers into C++20 working draft. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0673R0. 16 June 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0673r0.pdf (also at Internet Archive 22 April 2020 02:23:22).

Herb Sutter. 2018a. Concepts in-place syntax. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0745R0. 11 Feb. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0745r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2018b. Zero-overhead deterministic exceptions: Throwing values. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0709R0. 2 May 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0709r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter. 2019. Lifetime safety: Preventing common dangling — version 1.1. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1179R1. 22 Nov. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter, Casey Carter, Gabriel Dos Reis, Eric Niebler, Bjarne Stroustrup, Andrew Sutton, and Ville Voutilainen. 2019. Rename concepts to standard_case for C++20, while we still can. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1754R0. 16 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1754r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter and Bjarne Stroustrup. 2003. A name for the null pointer: nullptr. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1488. 10 Sept. 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1488.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. 2015. Structured bindings. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0144R0. 14 Oct. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Andrew Sutton. 2017. Working Draft, C++ extensions for Concepts. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4674. 19 June 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4674.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Andrew Sutton, Sam Goodrick, and Daveed Vandevoorde. 2019. Expansion statements. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1306R1. 21 Jan. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1306r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Andrew Sutton and Richard Smith. 2014. Folding expressions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4295. 7 Nov. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4295.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Andrew Sutton and Bjarne Stroustrup. 2011. Design of Concept Libraries for C++. In *Software Language Engineering: 4th International Conference (SLE 2011)* (3 July 2011–). Springer, Berlin and Heidelberg, Germany, 97–118. 978-3-642-28829-6 https://doi.org/10.1007/978-3-642-28830-2_6 LNCS 6940.

Andrew Sutton and Herb Sutter. 2018. Value-based Reflection. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0993R0. 2 April 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0993r0.pdf (also at Internet Archive 22 April 2020 02:28:24).

Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2018. Scalable Reflection in C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1240R0. 8 Oct. 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1240r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Clang Team. 2014. Clang 3.5 Documentation: Modules. http://clang.llvm.org/docs/Modules.html (also at Internet Archive 5 Dec. 2014 14:06:59).

Andrew Tomazos. 2015. Proposal of [[unused]], [[nodiscard]] and [[fallthrough]] attributes. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0068R0. 3 Sept. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0068r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Andrew Tomazos and Michael Spertus. 2014. Defaulted Comparison Using Reflection. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4239. 12 Oct. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4239.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Hubert Tong and Faisal Vali. 2016. Smart References through Delegation: An Alternative to N4477's Operator Dot. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0352R0. 30 May 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0352r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

James Touton and Mike Spertus. 2015. Template Argument Type Deduction. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4469. 10 April 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4469.html (also at Internet Archive 4 Sept. 2019 02:05:46).

James Touton and Mike Spertus. 2016. Declaring non-type template arguments with auto. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0127R1. 4 March 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0127r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Clay Trychta. 2016. Attributes for Likely and Unlikely Branches. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0479R0. 16 Oct. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0479r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

TSO Wikipedia 2020. Memory ordering. https://wikipedia.org/wiki/Memory_ordering Archived at https://web.archive.org/web/20200301003203/https://en.wikipedia.org/wiki/Memory_ordering

Faisal Vali, Herb Sutter, and Dave Abrahams. 2012. Proposal for Generic (Polymorphic) Lambda Expressions. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3418. 21 Sept. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3418.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Jan Christiaan van Winkel, Jose Daniel Garcia, Ville Voutilainen, Roger Orr, Michael Wong, and Sylvain Bonnal. 2017. Operating principles for evolving C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0559R0. 31 Jan. 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0559r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Daveed Vandevoorde. 2007. Modules in C++ (Revision 5). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2316. 19 June 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2316.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Daveed Vandevoorde. 2009. New wording for C++0x Lambdas (rev. 2). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2927. 15 July 2009. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2927.pdf (also at Internet

Archive 4 Sept. 2019 02:05:46).

Daveed Vandevoorde. 2012. Modules in C++ (Revision 6). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3347. 11 Jan. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Daveed Vandevoorde. 2017. Down with typename! ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0634. 5 March 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0634r0.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates: The Complete Guide.* Addison-Wesley, Reading, Massachussets, USA. 978-0201734843

David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. 2018. *C++ Templates — The Complete Guide (Second Edition).* Addison-Wesley, Boston, Massachussets, USA. 978-0-321-71412-1

vcpkg 2016–2020. vcpkg, a C++ package manager for Windows, Linux and MacOS (part of Microsoft's Visual Studio) (website). https://docs.microsoft.com/en-us/cpp/build/vcpkg?view=vs-2019 (also at Internet Archive 25 Dec. 2019 14:16:14).

> vcpkg is a command-line package manager for C++. It greatly simplifies the acquisition and installation of third-party libraries on Windows, Linux, and MacOS. If your project uses third-party libraries, we recommend that you use vcpkg to install them. vcpkg supports both open-source and proprietary libraries. All libraries in the vcpkg Windows catalog have been tested for compatibility with Visual Studio 2015, Visual Studio 2017, and Visual Studio 2019. Between the Windows and Linux/MacOS catalogs, vcpkg now supports over 1900 libraries. The C++ community is adding more libraries to both catalogs on an ongoing basis.

Todd L. Veldhuizen. 2003. C++ Templates are Turing Complete. Indiana University Computer Science. Archived at http://web.archive.org/web/20040919170502/http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf Also at http://rtraba.files.wordpress.com/2015/05/cppturing.pdf

Ville Voutilainen. 2016a. Discussion about std::thread and RAII. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0206R0. 27 Jan. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0206r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Ville Voutilainen. 2016b. Presentation at WG21 meeting in Jacksonville, Florida. Feb. 2016.

Ville Voutilainen. 2016c. Why I want Concepts, and why I want them sooner rather than later. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0225R0. 5 Feb. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0225r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Ville Voutilainen. 2019a. Adding a global contract assumption mode. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1710R0. 17 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1710r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Ville Voutilainen. 2019b. To boldly suggest an overall plan for C++23. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0592R4. 25 Nov. 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Ville Voutilainen, Thomas Köppe, Andrew Sutton, Herb Sutter, Gabriel Dos Reis, Bjarne Stroustrup, Jason Merrill, Hubert Tong, Eric Niebler, Casey Carter, Tom Honermann, and Erich Keane. 2018. Yet another approach for constrained declarations. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1141R0. 23 June 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1141r0.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Ville Voutilainen and Daveed Vandevoorde. 2016. constexpr if. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0128R1. 10 Feb. 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0128r1.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Ville Voutilainen and Jonathan Wakely. 2018. Integrating feature-test macros into the C++ WD (rev. 2). ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0941R2. 8 June 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0941r2.html (also at Internet Archive 4 Sept. 2019 02:05:46).

VStudio Wikipedia 2020. Microsoft Visual Studio. https://en.wikipedia.org/wiki/Microsoft_Visual_Studio (also at Internet Archive 10 April 2020 22:12:27).

Philip Wadler and Stephen Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA, Jan.) (*POPL '89*). Association for Computing Machinery, New York, NY, USA, 60–76. 0897912942 https://doi.org/10.1145/75277.75283

Jonathan Wakely. 2018. Working Draft, C++ Extensions for Networking. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N4734. 4 April 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4734.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Wandbox 2016–2020. Wandbox, a Japanese online environment for compiling and running C++ using a variety of compilers (website). https://wandbox.org/ (also at Internet Archive 2 Jan. 2020 21:23:50).

Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. 2006. An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, Oct.) (*OOPSLA '06*). Association for Computing Machinery, New York, NY, USA, 345–362. 1595933484 https://doi.org/10.1145/1167473.1167503

WebAssembly 2017–2020. WebAssembly (Wasm), a portable binary code format primarily for executing in browsers (website). https://webassembly.org/ (also at Internet Archive 1 April 2020 05:02:38).

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

WG14. 2007. Thread Cancellation. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N2455. 11 Oct. 2007. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2455.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

WG21. 1989–2020. C++ Standards Committee Papers (website). http://open-std.org/jtc1/sc22/wg21/docs/papers/ (also at Internet Archive 4 Sept. 2019 02:05:46). Links to all official papers for ISO/IEC JTC1/SC2/WG21.

Jeremiah Willcock, Jaakko Järvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine. 2006. Lambda expressions and closures for C++. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N1968. 26 Feb. 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).

Anthony Williams. 2012. *C++ Concurrency in Action – Practical Multithreading*. Manning Publishing, Shelter Island, NY, USA. 978-1933988771

Anthony Williams. 2018. *C++ Concurrency in Action – Practical Multithreading (2nd edition)*. Manning Publishing, Shelter Island, NY, USA. 978-1617294693

wxWidgets 1992–2020. wxWidgets, an open-source, cross-platform GUI library (website). http://www.wxwidgets.org/ (also at Internet Archive 9 April 2020 07:40:49).

wxWidgets is a C++ library that lets developers create applications for Windows, macOS, Linux and other platforms with a single code base. It has popular language bindings for Python, Perl, Ruby and many other languages, and unlike other cross-platform toolkits, wxWidgets gives applications a truly native look and feel because it uses the platform's native API rather than emulating the GUI. It's also extensive, free, open-source and mature.

Jing Yang, Gogul Balakrishnan, Naoto Maeda, Franjo Ivančić, Aarti Gupta, Nishant Sinha, Sriram Sankaranarayanan, and Naveen Sharma. 2012. Object Model Construction for Inheritance in C++ and Its Applications to Program Analysis. In *Compiler Construction: 21st International Conference (CC 2012)* (Tallinn, Estonia), Michael O'Boyle (Ed.). Springer, Berlin and Heidelberg, Germany, 144–164. 978-3-642-28651-3 https://doi.org/10.1007/978-3-642-28652-0_8 Also at https://link.springer.com/content/pdf/10.1007/978-3-642-28652-0_8.pdf (also at Internet Archive 29 July 2018 19:36:26). LNCS 7210. Held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2012).

Jeffrey Yasskin. 2014. String_view: a non-owning reference to a string, revision 7. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper N3921. 14 Feb. 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3921.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Mani Zandifar, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. 2014. The STAPL Skeleton Framework. In *The 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2014)* (Hillsboro, Oregon, USA, 17 Sept.), James Brodman and Peng Tu (Eds.). Springer International Publishing, Cham, Switzerland, 176–190. 978-3-319-17472-3 https://doi.org/10.1007/978-3-319-17473-0_12 Also at https://www.researchgate.net/profile/Mani_Zandifar/publication/279286000_The_stapl_Skeleton_Framework/links/56327c9208ae242468d9f8df.pdf LNCS 8967.

Victor Zverovich. 2019. Text Formatting. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P0645R9. 16 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r9.html (also at Internet Archive 4 Sept. 2019 02:05:46).

Victor Zverovich, Daniela Engert, and Howard E. Hinnant. 2019. Integration of chrono with text formatting. ISO/IEC JTC1/SC2/WG21: C++ Standards Committee paper P1361R1. 14 June 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1361r1.pdf (also at Internet Archive 4 Sept. 2019 02:05:46).