C++ Experiments: Cpp2 vs Carbon

Alexander Enaldiev

Kaspersky

C++ Russia 2023

Tur1m

Содержание

- Предпосылки к созданию так называемых «преемников C++» (C++ successors)
- Преемник: Carbon
- Другой преемник: Cpp2
- И снова про C++

Предпосылки

Предпосылки

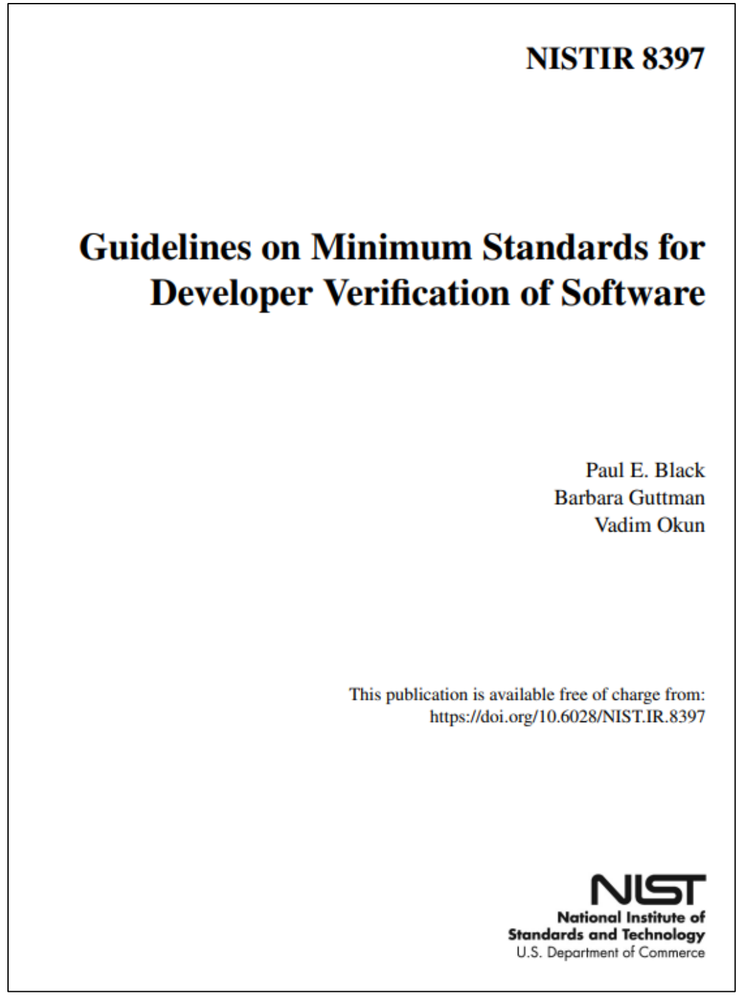
- NISTIR 8397 “Guidelines on Minimum Standards for Developer Verification of Software” (Oct 2021)

  3.2 Supplemental: Memory-Safe Compilation

  Some languages, such as C and C++, are not memory-safe. A minor memory access error can lead to vulnerabilities such as privilege escalation, denial of service, data corruption, or exfiltration of data. Many languages are memory-safe by default but have mechanisms to disable those safeties when needed, e.g., for critical performance requirements.

  Where practical, use memory-safe languages and limit disabling memory safety mechanisms.

  For software written languages that are not memory-safe, consider using automated source code transformations or compiler techniques that enforce memory safety. Requesting memory mapping to a fixed (hardcoded) address subverts address space layout randomization (ASLR)


NISTIR 8397

Guidelines on Minimum Standards for Developer Verification of Software

Paul E. Black
Barbara Guttman
Vadim Okun

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8397

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

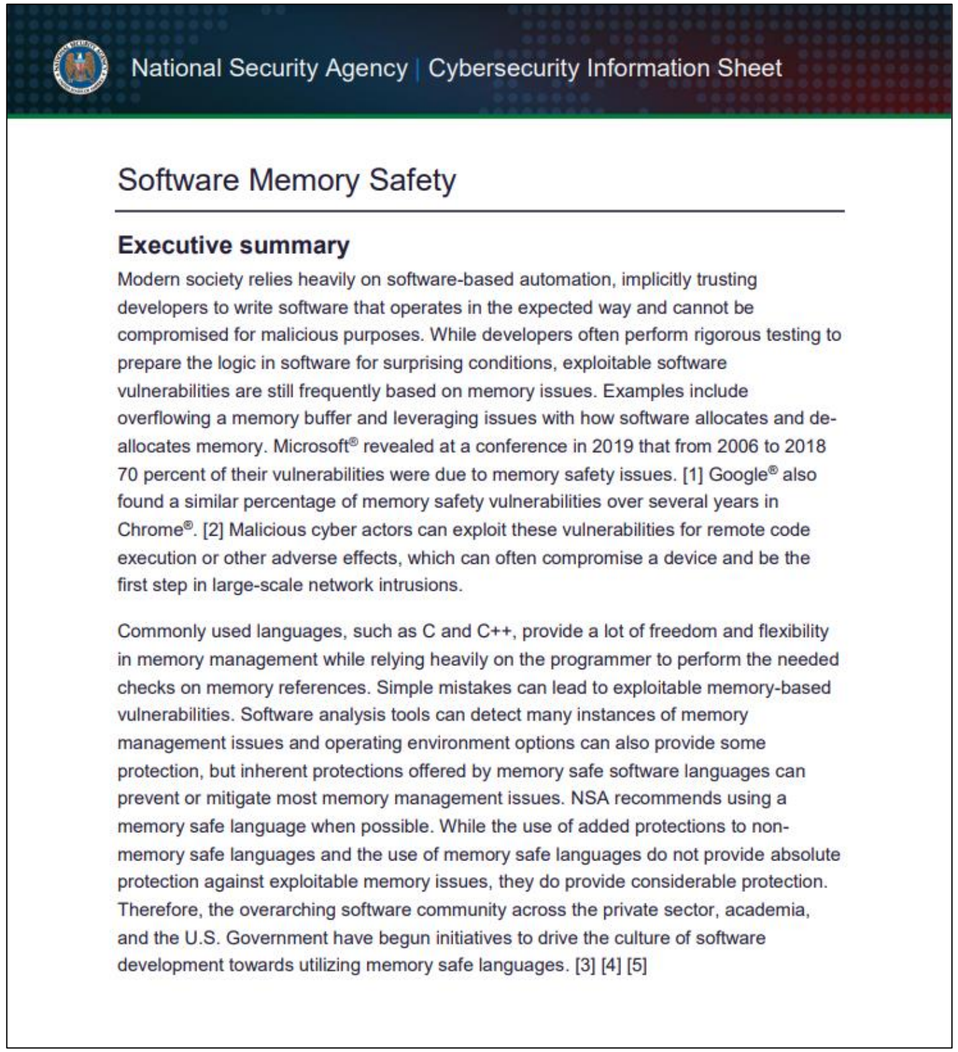
https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8397.pdf

Предпосылки

- NISTIR 8397 “Guidelines on Minimum Standards for Developer Verification of Software” (Oct 2021)
- NSA “Software Memory Safety” (Nov 2022)

  …Microsoft revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues ... Google also found a similar percentage of memory safety vulnerabilities over several years in Chrome...

  …Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities….



National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

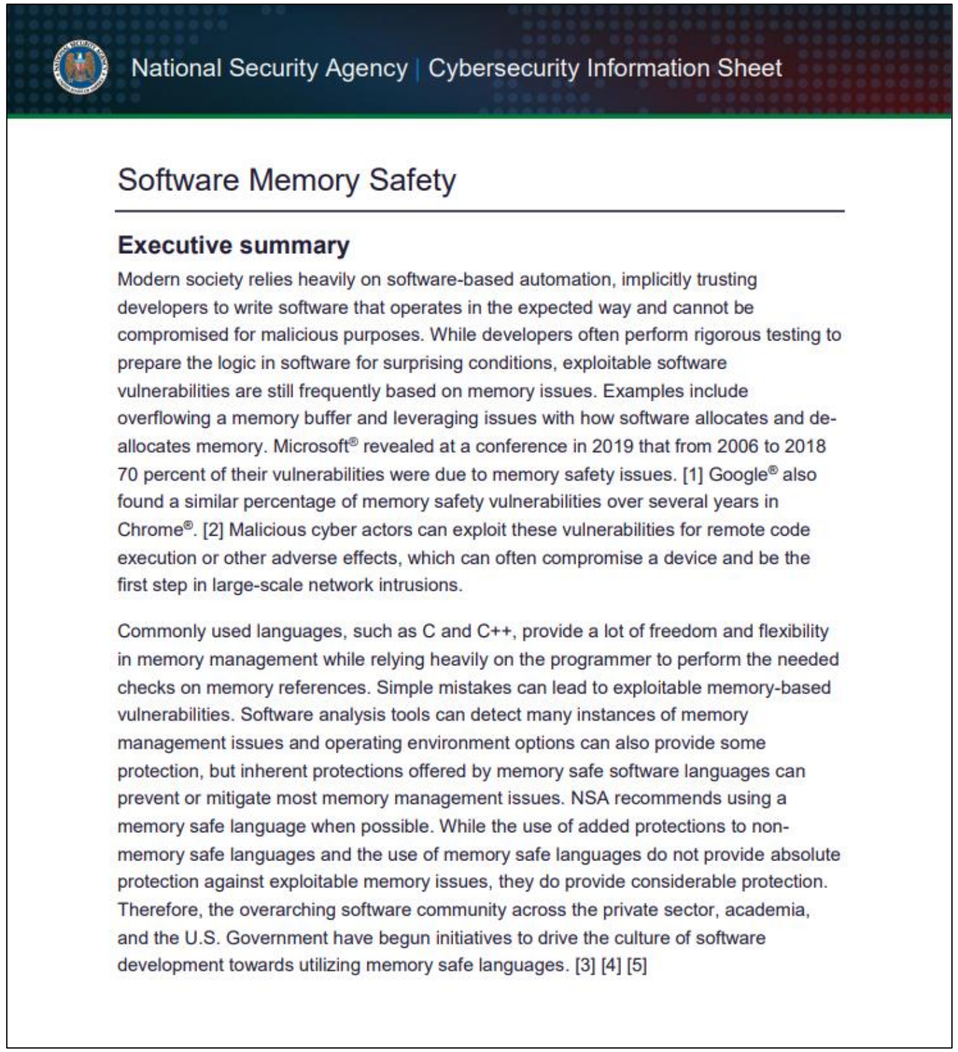
Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Предпосылки

- NISTIR 8397 “Guidelines on Minimum Standards for Developer Verification of Software” (Oct 2021)
- NSA “Software Memory Safety” (Nov 2022)
  - …
  - Pure Virtual Cast: “Где C++, а где безопасность, и при чем тут авиация?”
  - Cpp Cast: “Safety Critical C++”
  - …



National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

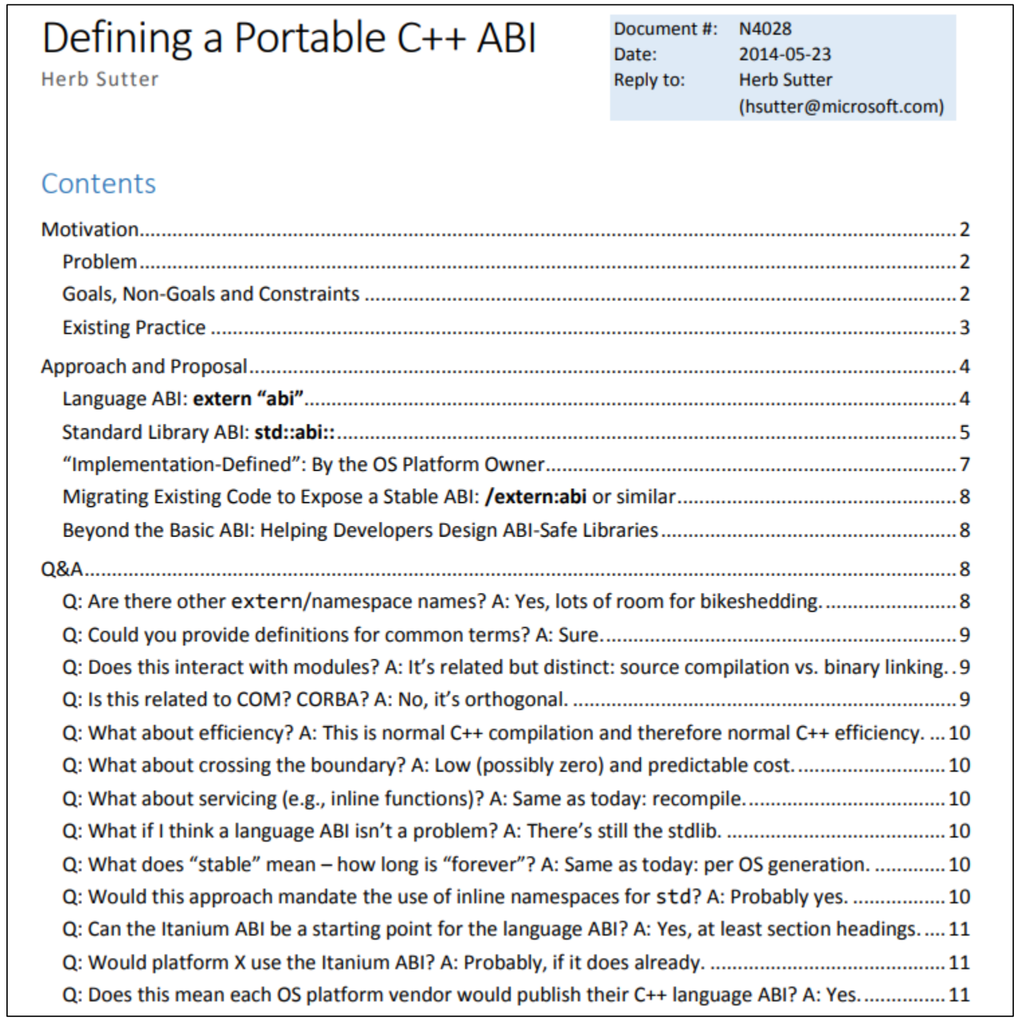
Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Предпосылки

- N4028, P1863 (P2028)
- NISTIR 8397 “Guidelines on Minimum Standards for Developer Verification of Software” (Oct 2021)
- NSA “Software Memory Safety” (Nov 2022)



Defining a Portable C++ ABI

Herb Sutter

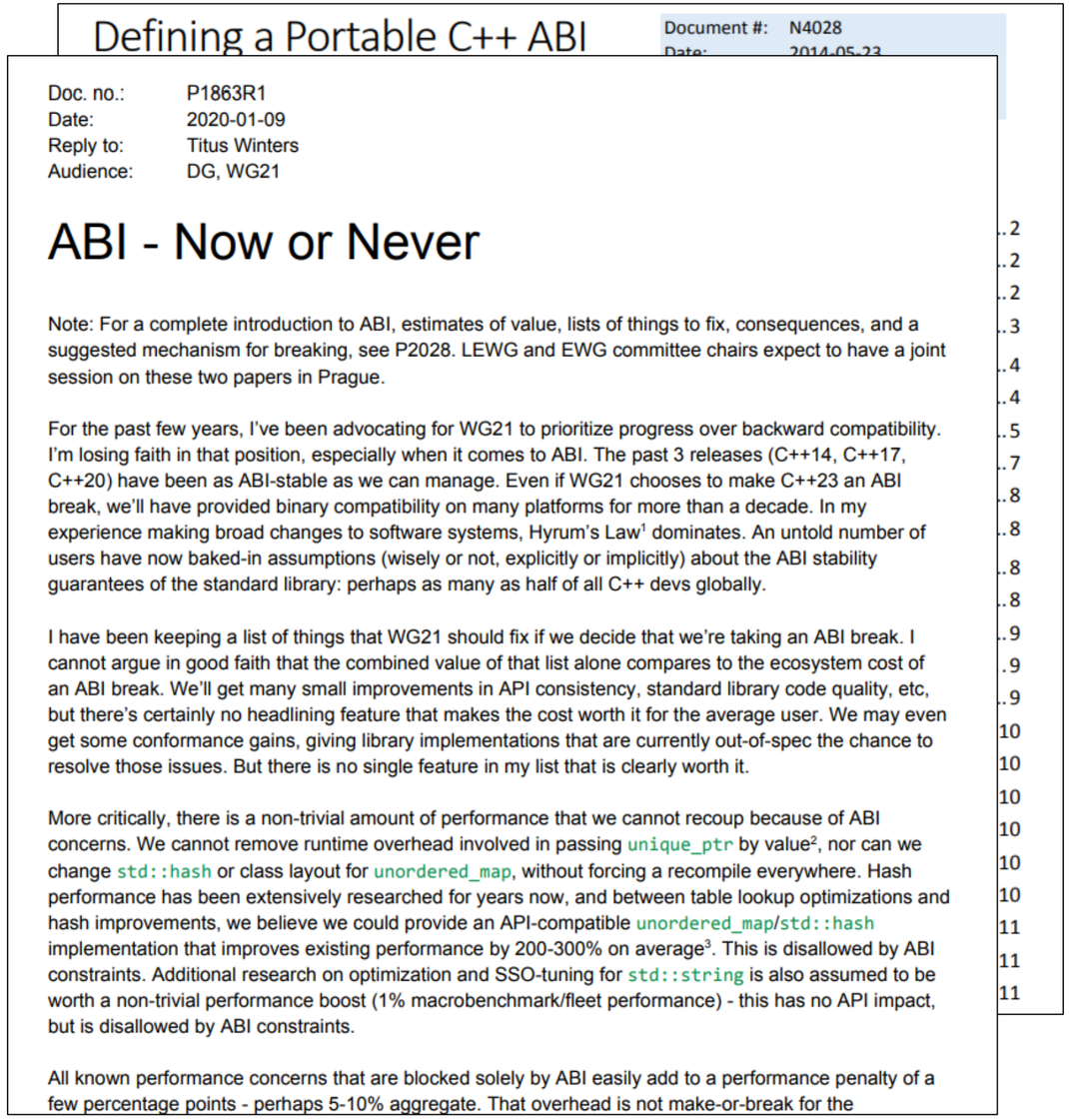
Document #: N4028
Date: 2014-05-23
Reply to: Herb Sutter (hsutter@microsoft.com)

Contents

https://isocpp.org/files/papers/n4028.pdf

Предпосылки

- N4028, P1863 (P2028)
- NISTIR 8397 “Guidelines on Minimum Standards for Developer Verification of Software” (Oct 2021)
- NSA “Software Memory Safety” (Nov 2022)



Defining a Portable C++ ABI

Document #: N4028
Date: 2014-05-23

Doc. no.: P1863R1
Date: 2020-01-09
Reply to: Titus Winters
Audience: DG, WG21

ABI - Now or Never

Note: For a complete introduction to ABI, estimates of value, lists of things to fix, consequences, and a suggested mechanism for breaking, see P2028. LEWG and EWG committee chairs expect to have a joint session on these two papers in Prague.

For the past few years, I’ve been advocating for WG21 to prioritize progress over backward compatibility. I’m losing faith in that position, especially when it comes to ABI. The past 3 releases (C++14, C++17, C++20) have been as ABI-stable as we can manage. Even if WG21 chooses to make C++23 an ABI break, we’ll have provided binary compatibility on many platforms for more than a decade. In my experience making broad changes to software systems, Hyrum’s Law[1] dominates. An untold number of users have now baked-in assumptions (wisely or not, explicitly or implicitly) about the ABI stability guarantees of the standard library: perhaps as many as half of all C++ devs globally.

I have been keeping a list of things that WG21 should fix if we decide that we’re taking an ABI break. I cannot argue in good faith that the combined value of that list alone compares to the ecosystem cost of an ABI break. We’ll get many small improvements in API consistency, standard library code quality, etc, but there’s certainly no headlining feature that makes the cost worth it for the average user. We may even get some conformance gains, giving library implementations that are currently out-of-spec the chance to resolve those issues. But there is no single feature in my list that is clearly worth it.

More critically, there is a non-trivial amount of performance that we cannot recoup because of ABI concerns. We cannot remove runtime overhead involved in passing `unique_ptr` by value[2], nor can we change `std::hash` or class layout for `unordered_map`, without forcing a recompile everywhere. Hash performance has been extensively researched for years now, and between table lookup optimizations and hash improvements, we believe we could provide an API-compatible `unordered_map`/`std::hash` implementation that improves existing performance by 200-300% on average[3]. This is disallowed by ABI constraints. Additional research on optimization and SSO-tuning for `std::string` is also assumed to be worth a non-trivial performance boost (1% macrobenchmark/fleet performance) - this has no API impact, but is disallowed by ABI constraints.

All known performance concerns that are blocked solely by ABI easily add to a performance penalty of a few percentage points - perhaps 5-10% aggregate. That overhead is not make-or-break for the

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1863r1.pdf

Предпосылки и митигация?

- P2739R0 B. Stroustrup:
  “A call to action: Think seriously about "safety"; THEN do something sensible about it” (Dec 2022)
- P2687R0 B. Stroustrup, Gabriel Dos Reis: “Design Alternatives for Type-and-Resource Safe C++” (Oct 2022) (etc)

Doc. no. P2739R0
Date: 2022-12-6
Project: Programming Language C++
Audience: All
Reply to: Bjarne Stroustrup (Bjarne@stroustrup.com)

A call to action:
Think seriously about “safety”;
then do something sensible about it

Bjarne Stroustrup
Columbia University
www.stroustrup.com

Consider this from the NSA: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF :

> the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]
>
> …
>
> NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®.

That specifically and explicitly excludes C and C++ as unsafe. As is far too common, it lumps C and C++ into the single category C/C++, ignoring 30+ years of progress. Unfortunately, much C++ use is also stuck in the distant past, ignoring improvements, including ways of dramatically improving safety.

Now, if I considered any of those “safe” languages superior to C++ for the range of uses I care about, I wouldn’t consider the fading out of C/C++ as a bad thing, but that’s not the case. Also, as described, “safe” is limited to memory safety, leaving out on the order of a dozen other ways that a language could (and will) be used to violate some form of safety and security.

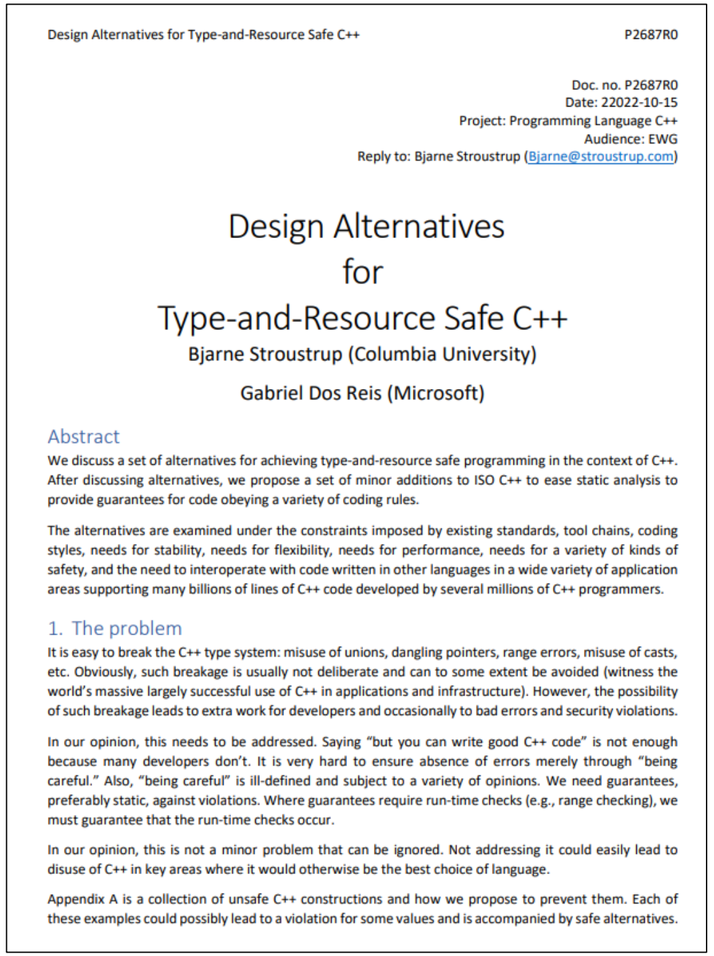
Now, I can’t say that I am surprised. After all, I have worked for decades to make it possible to write better, safer, and more efficient C++. In particular, the work on the C++ Core Guidelines specifically aims at delivering statically guaranteed type-safe and resource-safe C++ for people who need that without disrupting code bases that can manage without such strong guarantees or introducing additional tool chains. For example, the Microsoft Visual Studio analyzer and its memory-safety profile deliver much of the CG support today and any good static analyzer (e.g., Clang tidy, that has some CG support) could be

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2739r0.pdf

Предпосылки и митигация?

The latter pins down, safety violations:

- Logic errors;
- Resource leaks;
- Concurrency errors;
- Memory corruption;
- Type errors;
- Overflows and anticipated conversions;
- Timing errors;
- Termination errors;



Design Alternatives for Type-and-Resource Safe C++ P2687R0

Doc. no. P2687R0
Date: 22022-10-15
Project: Programming Language C++
Audience: EWG
Reply to: Bjarne Stroustrup (Bjarne@stroustrup.com)

Design Alternatives
for
Type-and-Resource Safe C++

Bjarne Stroustrup (Columbia University)

Gabriel Dos Reis (Microsoft)

Abstract

We discuss a set of alternatives for achieving type-and-resource safe programming in the context of C++. After discussing alternatives, we propose a set of minor additions to ISO C++ to ease static analysis to provide guarantees for code obeying a variety of coding rules.

The alternatives are examined under the constraints imposed by existing standards, tool chains, coding styles, needs for stability, needs for flexibility, needs for performance, needs for a variety of kinds of safety, and the need to interoperate with code written in other languages in a wide variety of application areas supporting many billions of lines of C++ code developed by several millions of C++ programmers.

1. The problem

It is easy to break the C++ type system: misuse of unions, dangling pointers, range errors, misuse of casts, etc. Obviously, such breakage is usually not deliberate and can to some extent be avoided (witness the world's massive largely successful use of C++ in applications and infrastructure). However, the possibility of such breakage leads to extra work for developers and occasionally to bad errors and security violations.

In our opinion, this needs to be addressed. Saying "but you can write good C++ code" is not enough because many developers don't. It is very hard to ensure absence of errors merely through "being careful." Also, "being careful" is ill-defined and subject to a variety of opinions. We need guarantees, preferably static, against violations. Where guarantees require run-time checks (e.g., range checking), we must guarantee that the run-time checks occur.

In our opinion, this is not a minor problem that can be ignored. Not addressing it could easily lead to disuse of C++ in key areas where it would otherwise be the best choice of language.

Appendix A is a collection of unsafe C++ constructions and how we propose to prevent them. Each of these examples could possibly lead to a violation for some values and is accompanied by safe alternatives.

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2687r0.pdf

Преемник: Carbon

Базовый синтаксис

```
package Sorting api;

fn Partition[T:! Comparable & Movable](s: Slice(T)) -> i64 {
     var i: i64 = -1;
     for( e: T in s) {
          if (e <= e.Last()) {
               ++i;
               Swap(&s[i], &e);
          }
     }
     return i;
}

fn QuickSort[T:! Comparable & Movable](s: Slice(T)) {
     if (s.Size() <= 1) {
          return;
     }
     let p: i64 = Partition(s);
     QuickSort(s[:p - 1]);
     QuickSort(s[p+1:]);
}
```

https://github.com/carbon-language/carbon-lang/blob/trunk/docs/images/snippets.md#quicksort

Преемник: Carbon

Базовый синтаксис

```
package Sorting api;

fn Partition[T:! Comparable & Movable](s: Slice(T)) -> i64 {
     var i: i64 = -1;
     for( e: T in s) {
          if (e <= e.Last()) {
               ++i;
               Swap(&s[i], &e);
          }
     }
     return i;
}

fn QuickSort[T:! Comparable & Movable](s: Slice(T)) {
     if (s.Size() <= 1) {
          return;
     }
     let p: i64 = Partition(s);
     QuickSort(s[:p - 1]);
     QuickSort(s[p+1:]);
}
```

```
interface Comparable {
// `Less` is an associated method.
     fn Less[self: Self](rhs: Self) -> bool;
}
…
```

https://github.com/carbon-language/carbon-lang/blob/trunk/docs/images/snippets.md#quicksort

Преемник: Carbon

Смешанный синтаксис

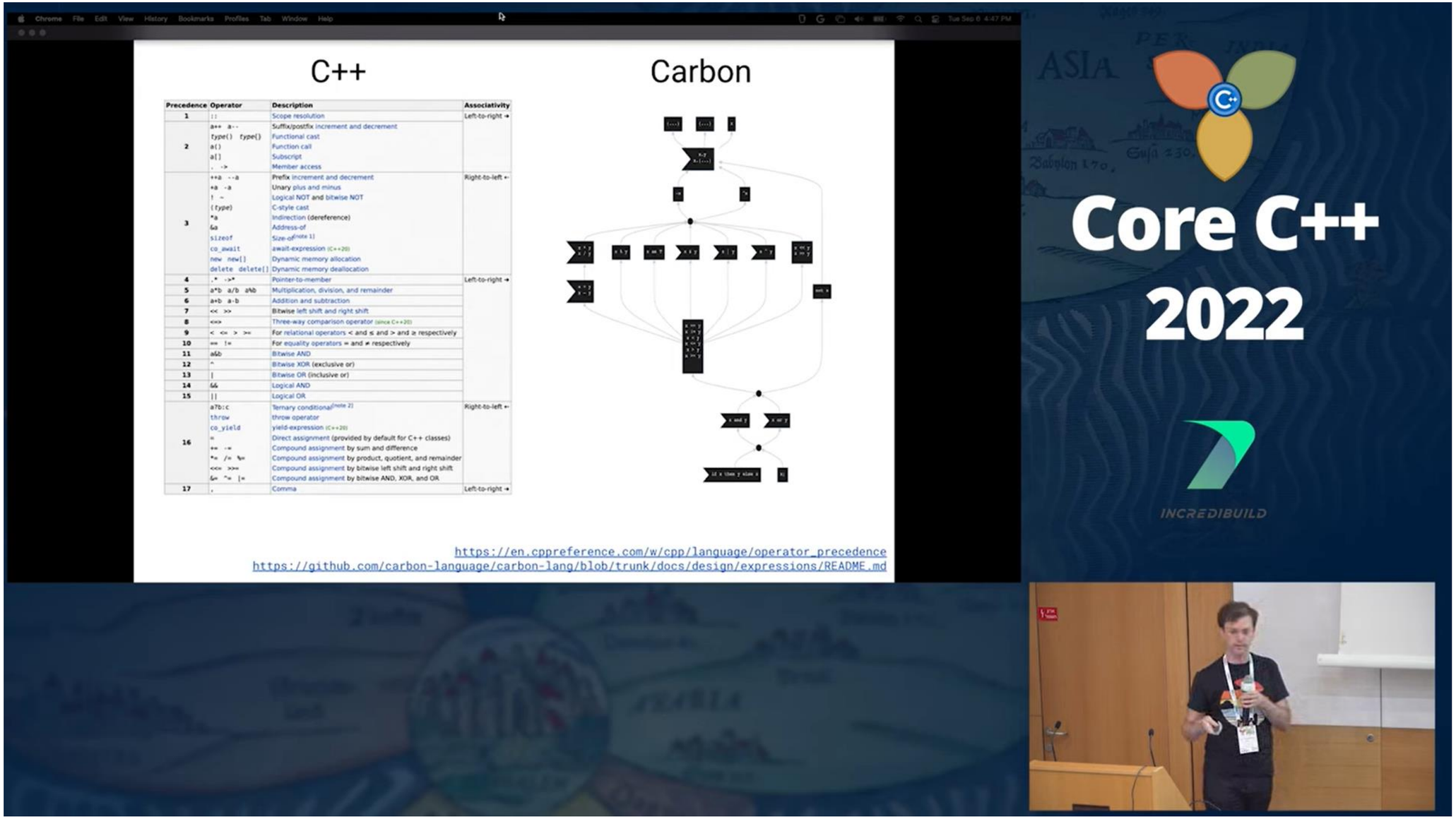
```
// C++ code used in both Carbon and C++:
struct Circle {
     float r;
};

// Carbon exposing a function for C++:
package Geometry api;
import Cpp library "circle.h";
import Math;

fn PrintTotalArea(circles: Slice(Cpp.Circle)) {
     var area: f32 = 0;
     for (c: Cpp.Circle in circles) {
          area += Math.Pi * c.r * c.r;
     }
     Print("Total area: {0}", area);
}
```

…

…

```
// C++ calling Carbon:
#include <vector>
#include "circle.h"
#include "geometry.carbon.h"
auto main(int argc, char** argv) -> int
{
     std::vector<Circle> circles =
{{1.0},   {2.0}};
     // Carbon's `Slice` supports
implicit construction from
`std::vector`, // similar to
`std::span`.
     Geometry::PrintTotalArea(circles);
     return 0;
}
```


https://github.com/carbon-language/carbon-lang/blob/trunk/docs/images/snippets.md#mixed



Carbon Language: Syntax and trade-offs: https://www.youtube.com/watch?v=9Y2ivB8VaIs by Jon Ross-Perkins

Преемник: Carbon

Language Goals

We are designing Carbon to support:

- Performance-critical software
- Software and language evolution
- Code that is easy to read, understand, and write
- Practical safety and testing mechanisms
- Fast and scalable development
- Modern OS platforms, hardware architectures, and environments
- Interoperability with and migration from existing C++ code

While many languages share subsets of these goals, what distinguishes Carbon is their combination.

We also have explicit *non-goals* for Carbon, notably including:

- A stable application binary interface (ABI) for the entire language and library
- Perfect backwards or forwards compatibility

https://github.com/carbon-language/carbon-lang

Преемник: Carbon

Roadmap

Table of contents

https://github.com/carbon-language/carbon-lang/blob/trunk/docs/project/roadmap.md

Преемник: Carbon

Текущий статус:

- toolchain (unstable) ;

Goals

The toolchain represents the production portion of Carbon. At a high level, the toolchain's top priorities are:

- Correctness.
- Quality of generated code, including performance.
- Compilation performance.
- Quality of diagnostics for incorrect or questionable code.

TODO: Add an expanded document that details the goals and priorities and link to it here.

High-level architecture

The typical compilation flow of data is:

1. Load the file into a SourceBuffer.
2. Lex a SourceBuffer into a TokenizedBuffer.
3. Parse a TokenizedBuffer into a ParseTree.
4. Transform a ParseTree into a SemanticsIR.
5. Lower the SemanticsIR to an LLVM Module.

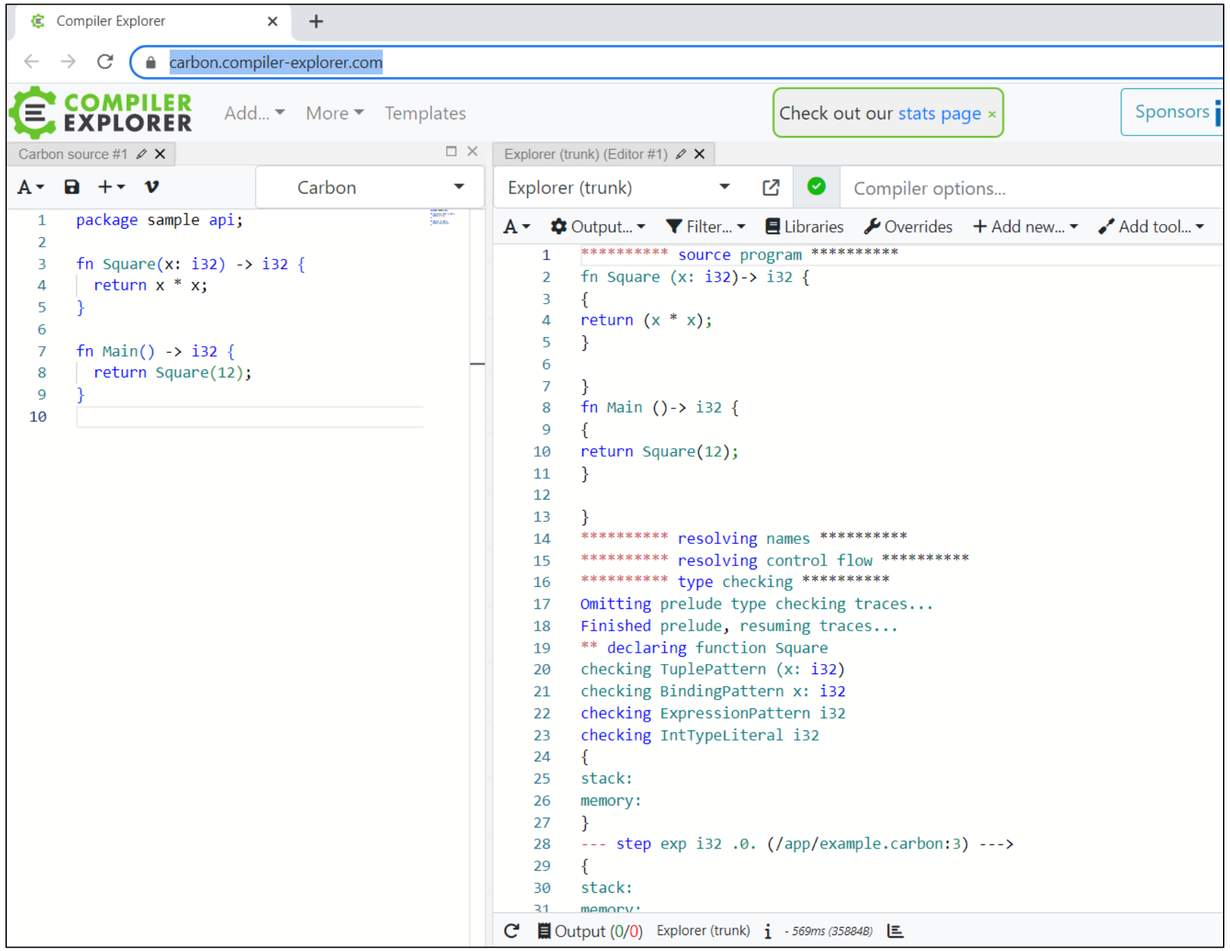
This flow is still incomplete: code generation, using LLVM, is still required.

https://docs.google.com/document/d/1RRYMm42osyqhI2LyjrjockYCutQ5dOf8Abu50kTrkX0/edit?pli=1&resourcekey=0-kHyqOESbOHmzZphUbtLrTw#heading=h.n1z6sen0ztt3

Преемник: Carbon

Текущий статус:

- toolchain (unstable) ;
- explorer (“…is an interpreter rather than a compiler…”) ;



https://carbon.compiler-explorer.com/

Другой преемник: Cpp2 (cppfront)

https://www.youtube.com/watch?v=ELeZAKCN4tY

Другой преемник: Cpp2 (cppfront)

```
#include <iostream>
#include <string>
name: () -> std::string = {
    s: std::string = "world";
    decorate(s);
    return s;
}
decorate: (inout s: std::string) = {
    s = "[" + s + "]";
}
auto main() -> int {
    // name();
    std::cout << "Hello " << name() << "\n";
}
```

> “within C++, there is a much smaller and cleaner language struggling to get out”
>
> Bjarne Stroustrup, ***The Design and Evolution of C++***

https://github.com/hsutter/cppfront/blob/main/regression-tests/mixed-hello.cpp2

Другой преемник: Cpp2 (cppfront)

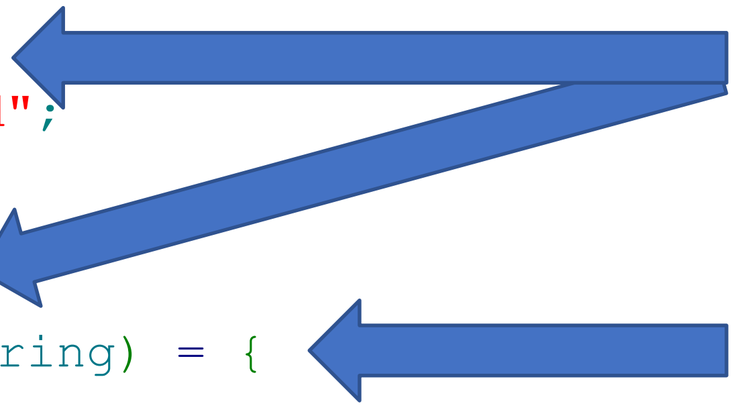
```
#include <iostream>
#include <string>
name: () -> std::string = {
    s: std::string = "world";
    decorate(s);
    return s;
}
decorate: (inout s: std::string) = {
    s = "[" + s + "]";
}
auto main() -> int {
    // name();
    std::cout << "Hello " << name() << "\n";
}
```

name : type = value

https://github.com/hsutter/cppfront/blob/main/regression-tests/mixed-hello.cpp2

Другой преемник: Cpp2 (cppfront)

```
#include <iostream>
#include <string>
name: () -> std::string = {
    s: std::string = "world";
    decorate(s);
    return s;
}
decorate: (inout s: std::string) = {
    s = "[" + s + "]";
}
auto main() -> int {
    // name();
    std::cout << "Hello " << name() << "\n";
}
```



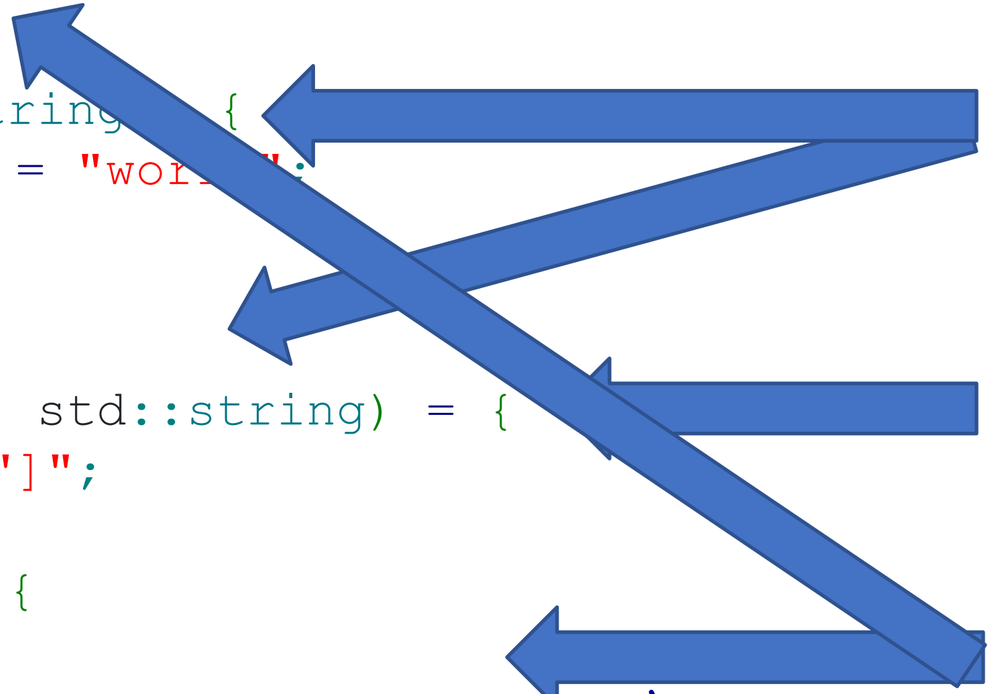
смешанный синтаксис

name : type = value

https://github.com/hsutter/cppfront/blob/main/regression-tests/mixed-hello.cpp2

Другой преемник: Cpp2 (cppfront)

```
#include <iostream>
#include <string>
name: () -> std::string = {
    s: std::string = "world";
    decorate(s);
    return s;
}
decorate: (inout s: std::string) = {
    s = "[" + s + "]";
}
auto main() -> int {
    // name();
    std::cout << "Hello " << name() << "\n";
}
```



смешанный синтаксис

name : type = value

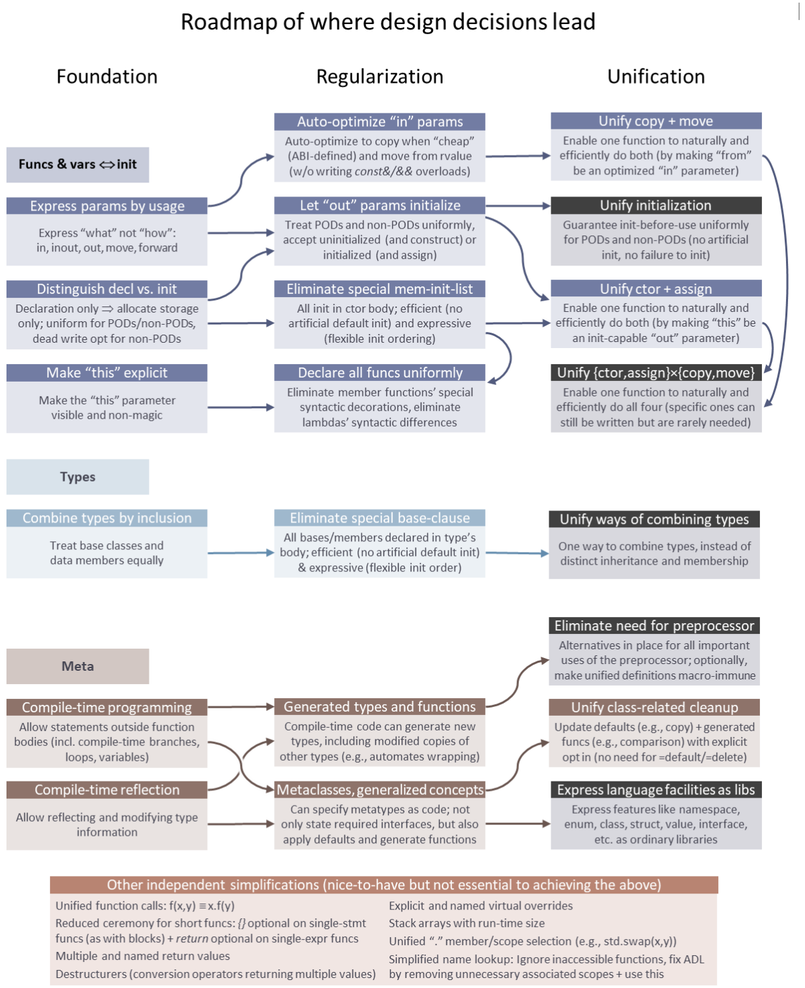
pure C++ syntax

https://github.com/hsutter/cppfront/blob/main/regression-tests/mixed-hello.cpp2

Другой преемник: Cpp2 (cppfront)

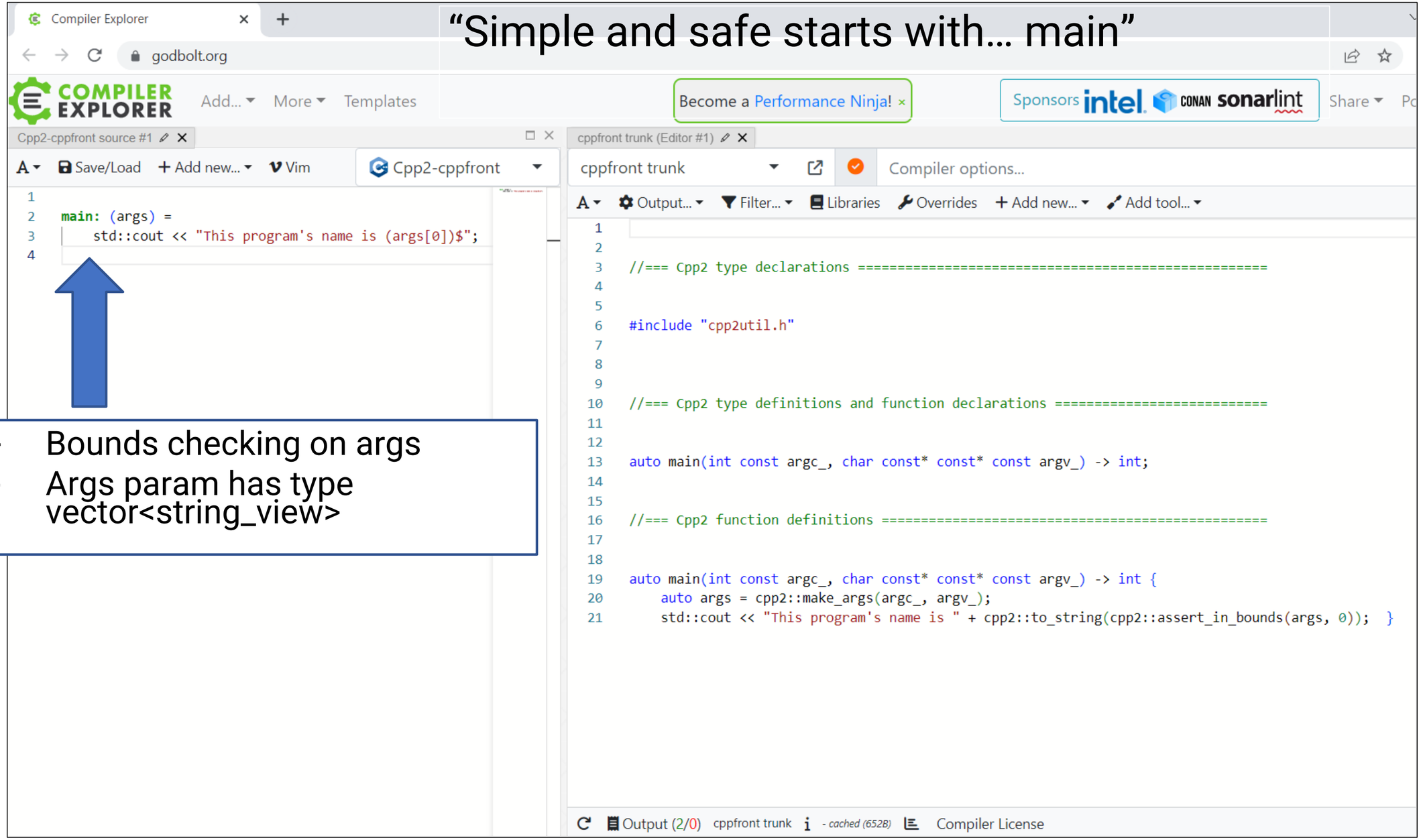
Основные цели альтернативного синтаксиса:

- Исправление поведения по умолчанию ([nodiscard] etc)
- Использование современного C++ (=начиная с C++20);
- Отказ от небезопасных конструкций (union, адресная арифметика etc) ;
- Безопасность типов и памяти по умолчанию;
- Уменьшение сложности использования языка;



https://github.com/hsutter/cppfront#epilog-2016-roadmap-diagram

“Simple and safe starts with… main”

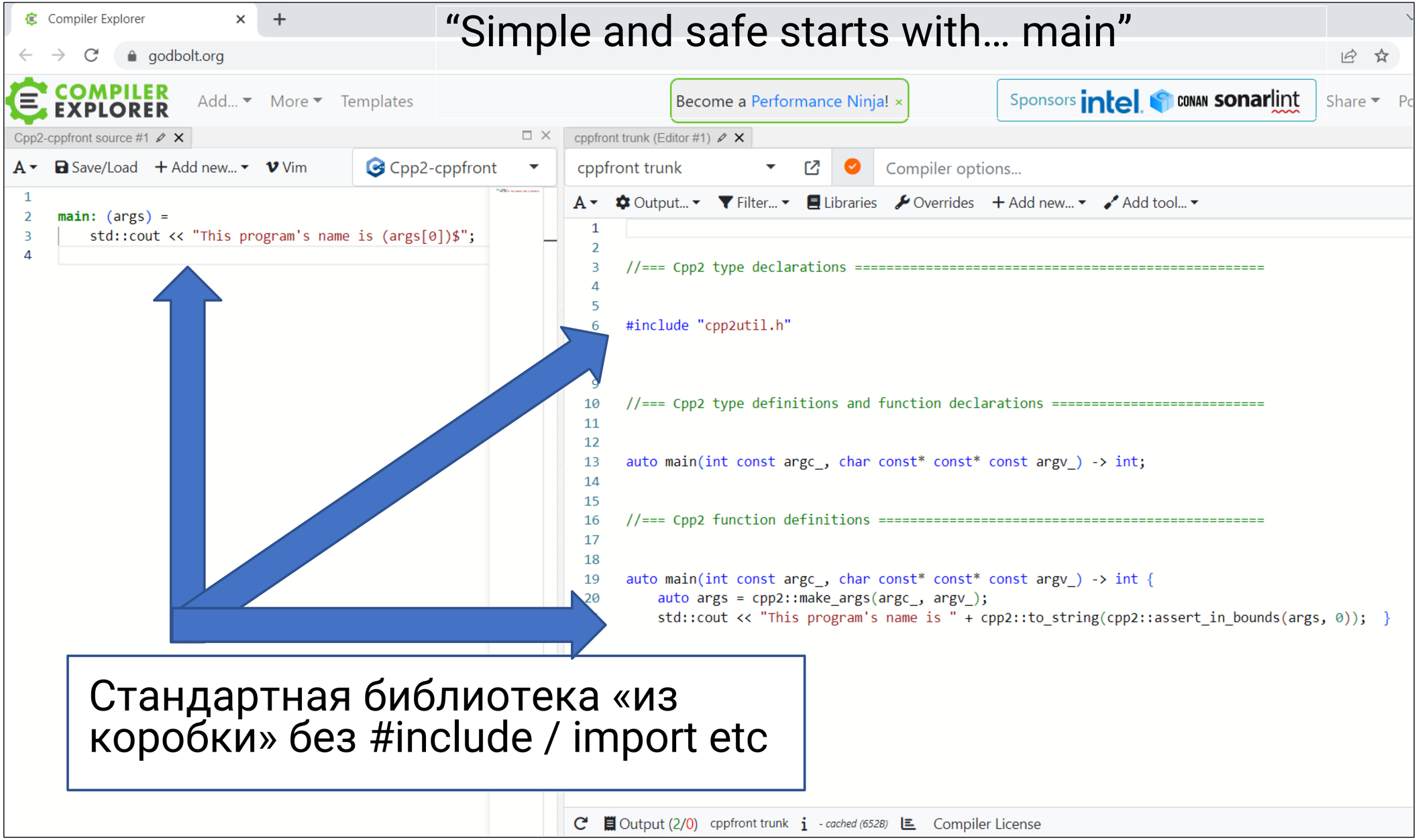


- Bounds checking on args
- Args param has type vector<string_view>

https://godbolt.org/

https://herbsutter.com/

“Simple and safe starts with… main”



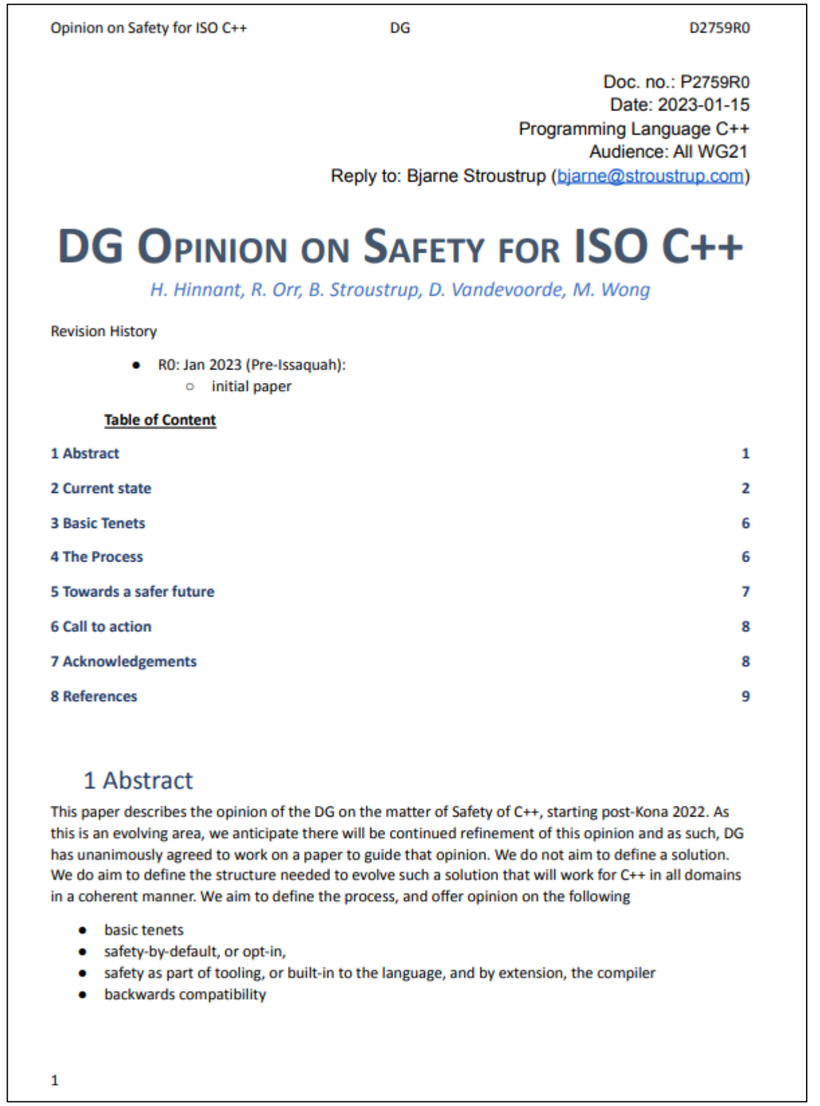
Стандартная библиотека «из коробки» без #include / import etc

https://godbolt.org/

https://herbsutter.com/

И снова про C++

- P2759 “DG opinion on safety for ISO C++”
  - Static, dynamic code analysis remains (wow!)
  - Safety’s (and its understanding) is evolving too
  - Backwards compatibility!
  - Profiles!



Opinion on Safety for ISO C++ DG D2759R0

Doc. no.: P2759R0
Date: 2023-01-15
Programming Language C++
Audience: All WG21
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

DG Opinion on Safety for ISO C++

H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong

Revision History

- R0: Jan 2023 (Pre-Issaquah):
  - initial paper

Table of Content

1 Abstract

This paper describes the opinion of the DG on the matter of Safety of C++, starting post-Kona 2022. As this is an evolving area, we anticipate there will be continued refinement of this opinion and as such, DG has unanimously agreed to work on a paper to guide that opinion. We do not aim to define a solution. We do aim to define the structure needed to evolve such a solution that will work for C++ in all domains in a coherent manner. We aim to define the process, and offer opinion on the following

- basic tenets
- safety-by-default, or opt-in,
- safety as part of tooling, or built-in to the language, and by extension, the compiler
- backwards compatibility

1

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2759r0.pdf

Val

The Val Programming Language

Language tour

Roadmap

Community discussion

Hosted on GitHub Pages — Theme by orderedlist

Val is a research programming language to explore the concepts of mutable value semantics and generic programming for high-level systems programming.

Val aims to be:

- **Fast by definition**: Val is compiled ahead-of-time to machine code and relies on its type system to support in-place mutation and avoid unnecessary memory allocations. Val avoids hidden costs such as implicit copies and therefore avoids heavy dependence on an optimizer for basic performance.
- **Safe by default**: Val's foundation of mutable value semantics ensures that ordinary code is memory safe, typesafe, and data-race-free. By explicit, auditable opt-in, programmers can use unsafe constructs for performance where necessary, and can build safe constructs using unsafe ones.
- **Simple**: Val borrows heavily from Swift which has demonstrated a user-friendly approach to generic programming and deep support for value semantics. Val's programming model strengthens and extends this support, while de-emphasizing reference semantics and avoiding the complexities that result from trying to make it statically safe (e.g., memory regions, lifetime annotations, etc.).

The **language tour** gives an overview of Val's features. The language specification and IR specification (work in progress) provides detailed information about Val's syntax and semantics.

Val is under active development and is not ready to be used yet. The code of the compiler is open source and hosted on GitHub. The current status of the project is described on our roadmap page.

https://www.val-lang.dev/

Mutable Value Semantics

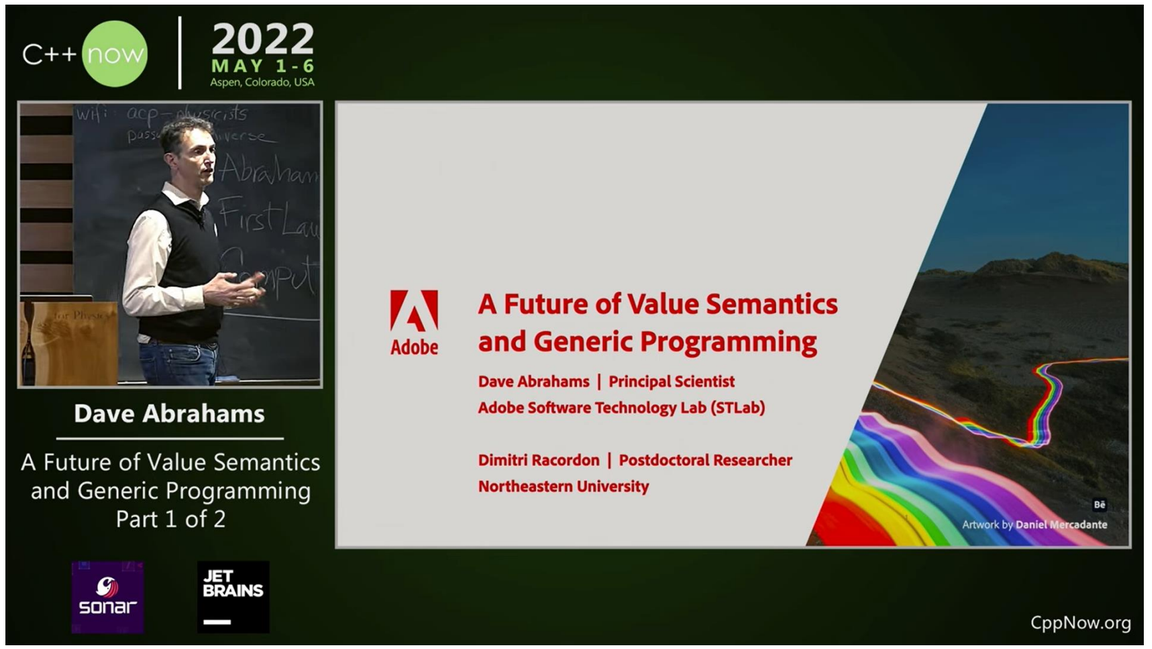
```
vector<int> v = {3, 2, 1};

auto broken_less(int left, int right) -> bool {
    v.push_back(v.front()); // !
    return left < right;
}

void test() {
    sort(v.begin(), v.end(), broken_less);
}
```

Mutable Value Semantics

```
vector<int> v = {3, 2, 1};

auto broken_less(int left, int right) -> bool {
    v.push_back(v.front()); // ! modification requires exclusive access! Law of Exclusivity (LoE)
    return left < right;
}

void test() {
    //sort(v.begin(), v.end(), broken_less);
    sort(&v, broken_less);
}
```

Mutable Value Semantics

```
swap(&x[i], &x[j]);
// ^LoE violation
-> x.swapAt(i, j);

// i = ++i + i++;
// ^also LoE violation
```



https://www.youtube.com/watch?v=4Ri8bly-dJs

Val

The Val Programming Language

Language tour

Roadmap

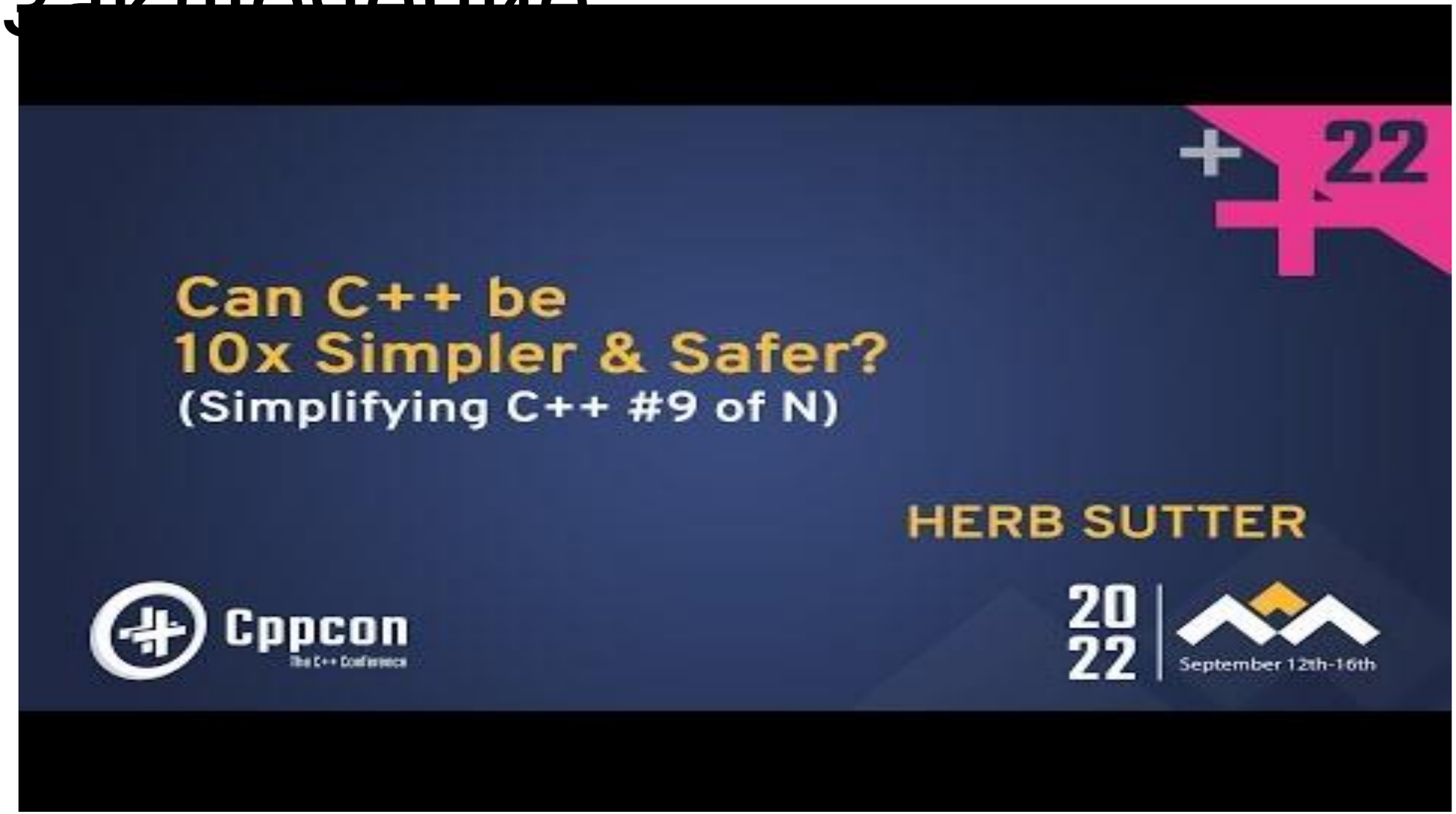
Community discussion

Roadmap: (shortened)

- 2023 – Q1/Q2
  - Deliver an alpha version of Val
  - Validate the language’s design
  - Access its usability
- 2023 – Q3/Q4
  - Complete the design of the language
  - Implement a standard library
  - Write and publish a specification

https://www.val-lang.dev/pages/implementation-status.html

В заключение



https://www.youtube.com/watch?v=ELeZAKCN4tY&t=86s

Спасибо