

Evolution of C++: Learning from other languages.

Bart Beumer (bart.beumer@stacker.nl)

2026-05-26

About the talk

- Available as PDF for download.
- Includes links to external sources of information.





About me

- Bart Beumer, 45yr
- Experienced C++ developer & trainer.
- Linux enthusiast.



Motivation

There are significant negative opinions regarding C++

- “It’s not safe! Memory problems are waiting!”
- “It’s too difficult”
- “My language is way more modern”



Goal

The goals of this talk:

- Show that shortcomings are being addressed.
- Inform you about some new features are added.
- Show that features are inspired by other languages.



NEWS: A C++26 compiler released!

GNU GCC 16.1 released April 30 ¹ ²

First compiler with significant C++26 support.

¹<https://isocpp.org/blog/2026/04/gcc-16.1>

²<https://gcc.gnu.org/gcc-16/>



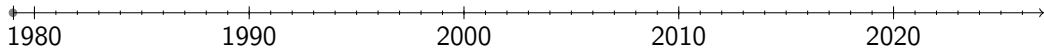
STACKER

Historical context



Historical context

C with classes



The world has changed a lot since the introduction of C++

Better computers

For comparison: A popular computer from 1988 ³:

- Intel 8088 16 bit computer @ 4.77Mhz
- 640kB RAM
- 20MB internal disk space.
- No hardware safety features



³Home computer museum Philips NMS9100



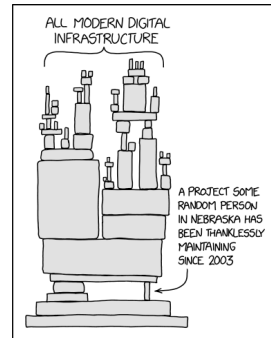
A connected world

A more connected world:

- 1995 dial up internet for consumers.
- 2000 always-on internet (ADSL) for consumers.
- 2007 First smartphones

Software development

- Larger software.
- More building blocks.
- Using external software found online.
- Developers expect more supplied functionality as a default.

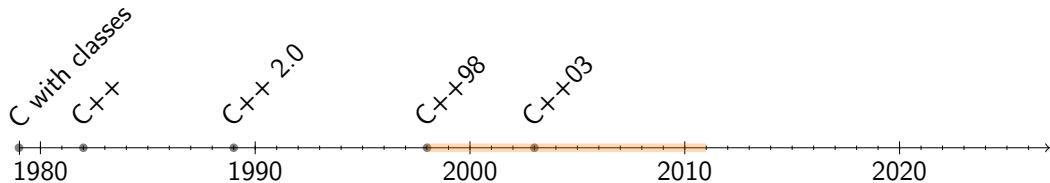


xkcd/2347



STACKER

C++ developing



Features added to C++ until C++98, long silence follows.

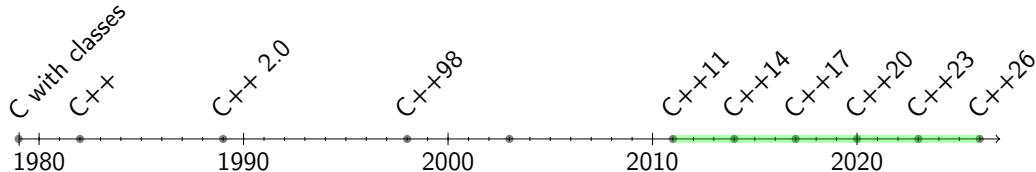


Other languages⁴

- Python (1991)
 - Java (1995)
 - C# (2000)
 - Go (2009)
 - Rust (2010)
 - ...
- Easier to use
 - Memory management (Garbage collection)
 - Improved memory safety
 - More (domain specific) functionality in the language by default.
 - Reflection
 - Package managers
 - ...

⁴Wikipedia, timeline programming languages

C++ development continues



- Steady stream of releases.
- New significant changes are being introduced.



Summary

The world of software has changed a lot since the beginnings of C++.

- better hardware
- more complex software
- more connected & exposed software

Other languages have started solving the problems of their time.

- easier to use
- safety features
- convenient usage of packages

We see C++ is evolving too.



STACKER

Evolutions



Topics

- Memory safety & robustness
- Concurrency
- Package managers



Topics

- Memory safety & robustness
Important for the future of C++. (Shown in detail)
 - Contracts (C++26).
 - Standard library hardening (C++26).
- Concurrency
- Package managers



Topics

- Memory safety & robustness

- Concurrency
 - Future enabler for asynchronous behavior. (Less detail)
 - Primitives (C++20).
 - Execution control library (C++26, not available).
- Package managers



Topics

- Memory safety & robustness

- Concurrency

- Package managers

Quick overview of useful feature we as developers want for C++



STACKER

Memory safety & robustness



STACKER

- C++ is not a memory safe language.
- Proven but limited mechanisms (`unique_ptr`, `shared_ptr`)
- Can opt-in to safer bounds-checking functions (`std::vector::at()`)
- Provide (core) guidelines instructing developer what (not) to do. ⁵

⁵C++ Core Guidelines, Resource management



STACKER

- C++ is not a memory safe language.
- Proven but limited mechanisms (`unique_ptr`, `shared_ptr`)
- Can opt-in to safer bounds-checking functions (`std::vector::at()`)
- Provide (core) guidelines instructing developer what (not) to do. ⁵

The developer is still responsible.

⁵C++ Core Guidelines, Resource management



This situation is becoming less acceptable ⁶.

CISA ⁷

... Second, companies should investigate memory safe programming languages. Most modern programming languages other than C/C++ are already memory safe.

White house ⁸

... Choosing to use memory safe programming languages at the outset, as recommended by the Cybersecurity and Infrastructure Security Agency's (CISA) Open-Source Software Security Roadmap is one example of developing software in a secure-by-design manner.

⁶(web archive) [whitehouse.gov](https://www.whitehouse.gov), Future software should be memory safe (2024)

⁷US Cyber Defence Agency, The urgent need for Memory Safety ... (2023)

⁸(web archive) [whitehouse.gov](https://www.whitehouse.gov), ONCD, A path towards secure and measurable software (2024)



C++26 significantly increases the ability to write safe and robust code.

Contracts

Contract assertions^{9 10 11}

Contract assertions allow the programmer to specify properties of the state of the program that are expected to hold at certain points during execution.

Prior Art: Eiffel (1986), D (2001), Ada

C and C++: the `assert` macro.

⁹cppreference, Contract assertions

¹⁰P2900R14 Contracts for C++

¹¹P2899R1 Contracts for C++ Rationale



Specify contracts at function declaration: ¹²

```
1 double sine(double num)
2     pre(!std::isnan(num))
3     pre(!std::isinf(num))
4     post(r : r >= -1.0 && r <= 1.0);
```

```
1 double sine(double num)
2 {
3     return std::sin(num);
4 }
```

¹²cppreference Function contract specifiers



Can add contract assertions in a function to verify an internal condition. ¹³

```
1 constexpr void do_something()
2 {
3     int a = do_your_magic();
4     contract_assert(a > 0);
5 }
```

¹³cppreference contract_assert



Use cases:¹⁴

- Documentation
- Runtime checking
- Static analysis.
- Optimization.
- Verification.



STACKER

Use cases:¹⁴

- Documentation
Express contract in code, useful for humans & tooling.
- Runtime checking
A safety mechanism during program execution, identifying defects.
- Static analysis.
A contract is C++ code which tools can interpret and use in analysis.
- Optimization.
Program could be optimized using contracts as assumptions to be true.
- Verification.
Can be used to verify and prove correctness of a piece of software.



Examples on what to check:

- ... a function argument pointer not being `nullptr`
- ... `std::list<T>::front()` is not called on an `empty()` container.
- ... provided divisor is not zero.
- ... ensure `operator[]` called using a valid index.
- ... supplied (combination of) function arguments is valid.

Evaluation semantics

Configure usage and behavior of contracts.

Evaluation semantics	Is checking	Is terminating
ignore		
observe	y	
enforce	y	y
quick-enforce	y	y

Semantics are chosen at compile time.

Handling contract violations

```
1 void handle_contract_violation(std::contracts::contract_violation);
```

`std::contracts::contract_violation`

Defined in header `<contracts>`

```
class contract_violation; (since C++26)
```

The class `std::contracts::contract_violation` defines the type of objects used to represent a contract violation that has been detected during the evaluation of a [contract assertion](#) with a particular evaluation semantic.

Objects of this type can only be created by the implementation when the [contract-violation handler](#) needs to be invoked. Users cannot create such objects directly.

Member functions

<code>(constructor)</code> <small>[deleted]</small>	<code>contract_violation</code> objects cannot be constructed by users <small>(public member function)</small>
<code>operator=</code> <small>[deleted]</small>	<code>contract_violation</code> objects cannot be assigned <small>(public member function)</small>
<code>(destructor)</code> <small>[possibly virtual]</small>	destructs the <code>contract_violation</code> object <small>(public member function)</small>

General contract-violation information

<code>kind</code>	returns the kind of the contract assertion violated <small>(public member function)</small>
<code>semantic</code>	returns the evaluation semantic when the contract violation occurs <small>(public member function)</small>
<code>is_terminating</code>	returns whether the evaluation semantic is terminating <small>(public member function)</small>
<code>detection_mode</code>	returns the reason that causes the contract violation <small>(public member function)</small>

Miscellaneous contract-violation information

<code>comment</code>	returns the explanatory string about the contract violation <small>(public member function)</small>
<code>location</code>	returns a <code>std::source_location</code> indicating the location of the contract violation <small>(public member function)</small>

Example application

Available on godbolt¹⁵

```
1 double sine(double num)
2     pre(!std::isnan(num))
3 {
4     return std::sin(num);
5 }
6
7 int main()
8 {
9     std::cout << " - " << sine(1.57) << std::endl;
10    std::cout << " - " << sine(std::nan("")) << std::endl;
11    std::cout << "end" << std::endl;
12 }
```

¹⁵<https://godbolt.org/z/MEaMsr67E>

Example output, contracts enforced

```
-std=c++26 -fcontracts -fcontract-evaluation-semantic=enforce
```

```
Program returned: 139
```

```
Program stdout
```

```
- 1  
-
```

```
Program stderr
```

```
contract violation in function double sine(double) at /app/example.cpp:5: !std::is  
[assertion_kind: pre, semantic: enforce, mode: predicate_false, terminating: yes]  
terminate called without an active exception  
Program terminated with signal: SIGSEGV
```

Example output, contracts observed

```
-std=c++26 -fcontracts -fcontract-evaluation-semantic=observe
```

```
Program returned: 0
```

```
Program stdout
```

```
- 1  
- nan
```

```
end
```

```
Program stderr
```

```
contract violation in function double sine(double) at /app/example.cpp:5: !std::is  
[assertion_kind: pre, semantic: observe, mode: predicate_false, terminating: no]
```



Contracts summarized

- Very useful for runtime checks.
- Providing clearer program terminations.

Standard library hardening

ISO C++ P3471R4¹⁶

While it is important to explore ways to make new code safer, we believe that the highest priority to deliver immediate real-world value should be to make existing code safer with minimal or no effort

DG Opinion on Safety for ISO C++¹⁷

... many have noted that safety critical applications have proliferated even more than ever, putting more strain on programming languages to be safe. This trend seems certain to accelerate now and in the future.

¹⁶[open-std.org, P3471R4](https://open-std.org/JTC1/SC22/WG21/P3471R4), Standard library hardening

¹⁷[open-std.org, P2759R0](https://open-std.org/JTC1/SC22/WG21/P2759R0), DG Opinion on safety for ISO C++



Improves the robustness of the standard library by turning some instances of undefined behavior into guaranteed termination of the program.

- Checks accessing an element is not out of bounds.
- Checks that ranges (iteration pair) represent a valid range.
- Checks that the pointer being dereferenced is not null.
- ...



Broken application¹⁸ example

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> empty;
7      empty.reserve(20);
8      std::cout << "printing a number:" << empty[22] << std::endl;
9  }
```

¹⁸<https://godbolt.org/z/eq41zeEqf>

- └ Memory safety & robustness
- └ Standard library hardening



Broken application, no checks

```
GCC 16.1: -std=c++26 -O3
```

```
Program returned: 0
```

```
Program stdout
```

```
printing a number:4113
```



Broken application, hardened

GCC 16.1: -std=c++26 -O3 -fhardened

Program returned: 139

Program stderr

```
/cefs/38/383ad2f84cbd57a52fd68bbe_consolidated/compilers_c++_x86_gcc_16.1.0/include/c++/16.1.0/bits/stl_vector.h:1253:
constexpr std::vector< <template-parameter-1-1>, <template-parameter-1-2> >::reference std::vector< <template-parameter-1-1>,
<template-parameter-1-2> >::operator[](size_type) [with _Tp = int; _Alloc = std::allocator<int>;
reference = int&; size_type = long unsigned int]: Assertion '._n < this->size()' failed.
Program terminated with signal: SIGSEGV
```



Broken application, older compiler

Older compilers might also perform some checks.

GCC 15.2: `-std=c++23 -O0`

```
Program returned: 139
```

```
Program stderr
```

```
/cefs/22/22e6cdc013c8541ce3d1548e_consolidated/compilers_c++_x86_gcc_15.2.0/include/c++/15.2.0/bits/stl_vector.h:1263:
constexpr std::vector< <template-parameter-1-1>, <template-parameter-1-2> >::reference std::vector< <template-parameter-1-1>,
<template-parameter-1-2> >::operator[](size_type) [with _Tp = int; _Alloc = std::allocator<int>;
reference = int&; size_type = long unsigned int]: Assertion '__n < this->size()' failed.
Program terminated with signal: SIGSEGV
```

cppreference.com

Create account

Search cppreference.com

Page [Discussion](#)Standard revision: [Diff](#) ▾

Read

[View source](#)[View history](#)[C++](#) [Containers library](#) [std::vector](#)

std::vector<T,Allocator>::operator[]

reference **operator[]**(size_type pos);

(1) (constexpr since C++20)

const_reference **operator[]**(size_type pos) **const**;

(2) (constexpr since C++20)

Returns a reference to the element at specified location `pos`. No bounds checking is performed, unless the implementation is hardened (since C++26).

If `pos < size()` is `false`, the behavior is undefined. (until C++26)

If `pos < size()` is `false`:

- If the implementation is hardened, a contract violation occurs. (since C++26)
- If the implementation is not hardened, the behavior is undefined.

Parameters

Standard library hardening

An implementation can be a *hardened implementation*, whether the implementation is hardened is implementation-defined.

Some standard library functions (and function templates) have *hardened precondition*. When such a function is invoked:

- If the implementation is hardened, prior to any other observable side effects of the function, *contract assertions* whose predicates are as described in the hardened precondition are evaluated with a terminating semantic.
- If the implementation is not hardened, when a hardened precondition is violated, the behavior is undefined.

Functions with hardened preconditions

Green - all overloads have hardened preconditions

Yellow - some overloads have hardened preconditions

Category	Class	Sequence containers					Container view		
		array	vector	inplace_vector	deque	list	forward_list	span	mds
Creation	constructor							span	mds
	static helper								
	operator*								
	operator->								
Element access	operator[]	operator[]	operator[]	operator[]	operator[]			operator[]	opera
	front	front	front	front	front	front	front	front	
	back	back	back	back	back	back		back	
	error								
Subviews	first							first	
	last							last	
	subspan							subspan	
	operator=								
Modifiers	operator+=								
	operator-=								
	operator++								
	pop_front				pop_front	pop_front	pop_front		
Non-member	pop_back		pop_back	pop_back	pop_back	pop_back			
	remove_prefix								
	remove_suffix								
	operator==								
Non-member	operator-								
	iter_move								
	iter_swap								

(since C++26)

- └ Memory safety & robustness
- └ Standard library hardening



“But... All these checks will kill my performance!”



Google chrome use case

Google uses hardened C++ and published their experience applying it to Chrome. ¹⁹

- Very low performance impact (0.3% impact claimed)
- Preventing exploits thwarting exploits, uncovering bugs
- 30% reduction in segmentation faults
- Crashes are easier to analyze/find since they are caught before causing undefined behavior.

¹⁹Google blog, Retrofitting spatial safety to hundreds of millions of lines of C++



- Clang 20 and up using libc++^a
- GCC 15 using libc++^b

GCC16 also offers a hardened standard library!

^a<https://libcxx.llvm.org/UserDocumentation.html#id3>

^b<https://releases.llvm.org/5.0.0/projects/libcxx/docs/UsingLibcxx.html#id4>

Name	Members	Functions	Iterators (ABI-d)
span	✓		STACKER ✓
string_view	✓		✓
array	✓		✗
vector	✓		✓ (see note)
string	✓		✓ (see note)
list	✓		✗
forward_list	✓		✗
deque	✓		✗
map	✗		✗
set	✗		✗
multimap	✗		✗
multiset	✗		✗
unordered_map	Partial		Partial
unordered_set	Partial		Partial
unordered_multimap	Partial		Partial
unordered_multiset	Partial		Partial
mdspan	✓		✗
optional	✓		N/A
function	✗		N/A
variant	N/A		N/A
any	N/A		N/A
expected	✓		N/A
valarray	Partial		N/A
bitset	✓		N/A



Hardening summarized

- Make new and existing code more robust.
- Failures result in easier to analyze program termination.
- Less segmentation faults to analyze.



Safety profiles

- Very much under development, earliest C++29²⁰ ²¹
- A profile is a set of guarantees (type safety, absence of resource leaks, and range errors, ...)
- Can opt in/out
- Could be implemented by banning features that could compromise the guarantees.

²⁰open-std.org, P3704R0, What about profiles?

²¹P4186R0 Proposed plan for profiles



Conceptual example: suppressed a profile

```
1 void f(int i)
2 {
3     (double*)&i; // error: type-unsafe
4     [[profiles::suppress(std::type)]]
5     (double*)&i; // ok
6 }
```

- Too early to dive deep.
- Worth tracking progress.



Conclusion on safety & robustness

- A lot of improvements have been made.
- Expect more improvements in the future.
- The subject has priority; seen as something needed stay relevant.



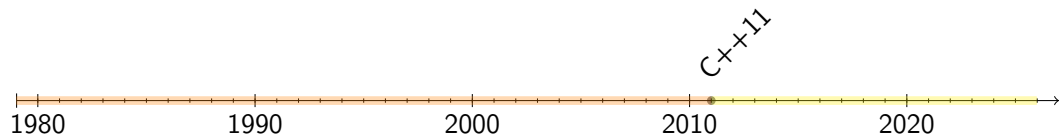
STACKER

Concurrency



Concurrency primitives

Low level building blocks for parallel execution:
thread, mutex, semaphore, promise/future, ...



- Pre C++11 no standardisation, different implementations existed.
- Part of the standard since C++11.



Some significant improvements and additions have been made in this area: ²²

- Joining thread (C++20)
- Semaphores (since C++20)
- Latches and Barriers (since C++20)

²²<https://en.cppreference.com/cpp/thread>

jthread example 1/3

A small example:²³

```
1 #include <chrono>
2 #include <iostream>
3 #include <thread>
4
5 using namespace std::chrono_literals;
```

²³Godbolt example



jthread example 2/3

```
6 void keep_doing_work(std::stop_token st);
7
8 int main()
9 {
10     {
11         std::jthread jt{keep_doing_work};
12         std::cout << "Zzzzz" << std::endl;
13         std::this_thread::sleep_for(3s);
14         std::cout << "Awake!" << std::endl;
15     }
16     std::cout << "No more jthread" << std::endl;
17 }
```



jthread example 3/3

```
18 void keep_doing_work(std::stop_token st)
19 {
20     while(!st.stop_requested())
21     {
22         std::cout << "... working" << std::endl;
23         std::this_thread::sleep_for(1s);
24     }
25     std::cout << "... done working" << std::endl;
26 }
```



Beyond primitives

Execution control library ²⁴ ²⁵ ²⁶

The Execution control library provides a framework for managing asynchronous execution on generic execution resources.

²⁴<https://en.cppreference.com/cpp/execution>

²⁵P2300R10

²⁶P0443R14



Concepts

- Scheduler
- Sender
- Receiver
- Operation State



Concepts

- Scheduler
Handle to an execution context.
- Sender
A description of asynchronous work to be sent for execution.
- Receiver
A generalized callback that consumes or “receives” the asynchronous results.
- Operation State
State needed by asynchronous operation



An alternative Hello World: ²⁷

```
1 using namespace std::execution;
2
3 scheduler sch = get_system_scheduler(); // P2079R3
4
5 sender s = schedule(sch)
6     | then([] {
7         std::print("Hello, from a different thread");
8     });
9
10 // submit and wait for completion
11 auto result = std::this_thread::sync_wait(s).value();
```

²⁷<https://accu.org/journals/overload/32/184/teodorescu/>



Follow-up: Async Networking & File I/O

Asynchronous execution. Not asynchronous I/O.

Timeline unknown.



STACKER

Package manager



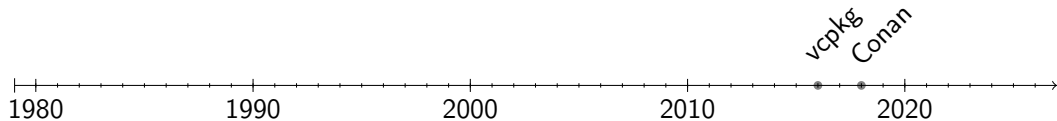
Package manager

(Relatively) Modern C++ tooling allows us to declare dependencies.

- Automatic retrieval of dependencies.
- Ensure package available at build.
- Less manual steps, easier to reproduce build.



STACKER



- vcpkg²⁸
Microsoft, free usage
- Conan²⁹
JFrog, open source

²⁸<https://vcpkg.io/en/>

²⁹<https://conan.io/>



STACKER

vcpkg

vcpkg.json

```
1 {  
2   "dependencies": [  
3     "cxxopts",  
4     "fmt",  
5     "range-v3"  
6   ]  
7 }
```



STACKER

Conan

An example of a conanfile.txt

```
1 [requires]
2   boost/1.84.0
3   gtest/1.14.0
4
5 [generators]
6   CMakeDeps
7   CMakeToolchain
```

More complex dependencies can be specified using a Python script.



STACKER

Conan usage

Relatively easy to use.

1

```
conan install . -of build -s compiler.cppstd=20 -s build_type=Debug --build=*
```

- Download & build.
- Handles regular dependencies & tool dependencies.
- Takes platform into account (x86, arm)
- Can be used in cross-platform environments.



STACKER

Conclusion



Summarize

- Memory safety features are becoming easily available.
- Features are added, more to follow
- The ecosystem is evolving, adding features.

The last slide

Questions?

