What is Safety?

The difference between “Language Safety” and “Grandma Safety”

Or

How technical words have meanings that matter to *you*.

Ryan McDougall (mcdougall.ryan@gmail.com)

Document #: P3578R0
Date: 2024-12-12
Project: Programming Language C++
Audience: SG23, SG21, EWG, WG21

Headliner

The single greatest barrier to making C++ safer is the meaning of the word “safety”. Failure to understand the precise meaning of decades-old established industrial jargon undermines our ability to address the problems facing C++, and defeats our own best intentions.

As contributors to a specification that relies on words of power with specific meanings, we should be precise. If we do not understand and use commonly held definitions of safety we are letting down the C++ community that is relying on us to get this right.

Note that this paper is mostly C++ agnostic – it offers a linguistic and ideological framework based on decades of existing practice. We speak C++-isms – but the industries that C++ serves do not.

Specifically we contrast the *synthesized* terms “Language Safety” and “Grandma Safety” as two opposite ends of a “safety spectrum”:

- **Language Safety** is the set of all things that the language designer and compiler can enforce – specifically, it's just the {Type, Memory, Lifetime, Thread, ...}-safeties that you already know.
- **Grandma Safety** is the set of things that ensure a program will not harm your Grandma. This is also known as *Functional Safety*.

These terms are not standard terms, but are meant to be illustrative. Functional Safety is the boring but correct term for “Grandma Safety”. Please use the former outside this paper. Because “Language Safety” is non-standard, use with caution. Some define “Language Safety” as “freedom from UB”, but that is C++ specific. UB is a consequence of Language Safety, not its definition.

1. Do not use “safe” or “safety” unhyphenated. Use the full, correct, industry recognized terms that precisely match what you mean.
2. Understand what Language Safety is, and how it’s different from Functional Safety.
3. Understand that Language Safety and Functional Safety are *both* required – though they are not required to be satisfied by the *same features*.
4. Contract checking (P2900 or otherwise) is *required* for Functional Safety, but benefits Language Safety as well. See C++26 Needs Contract Checking (approved by SG23).
5. If what you want is Language Safety, and you do not care about Functional Safety – please focus on improving Language Safety, and do not derail features that successfully address Functional Safety.

Tl;dr

In technical discussion and specification, it is meaningless to use the word “safety” unhyphenated. All of the following terms mean something different:

Spatial Safety	Temporal Safety	Type Safety	Bounds Safety
Memory Safety	Lifetime Safety	Thread Safety	Undefined Behavior
Functional Safety	Occupational Safety	Traffic/Road Safety	Industrial Safety
Correctness	Regulation	Security	etc…

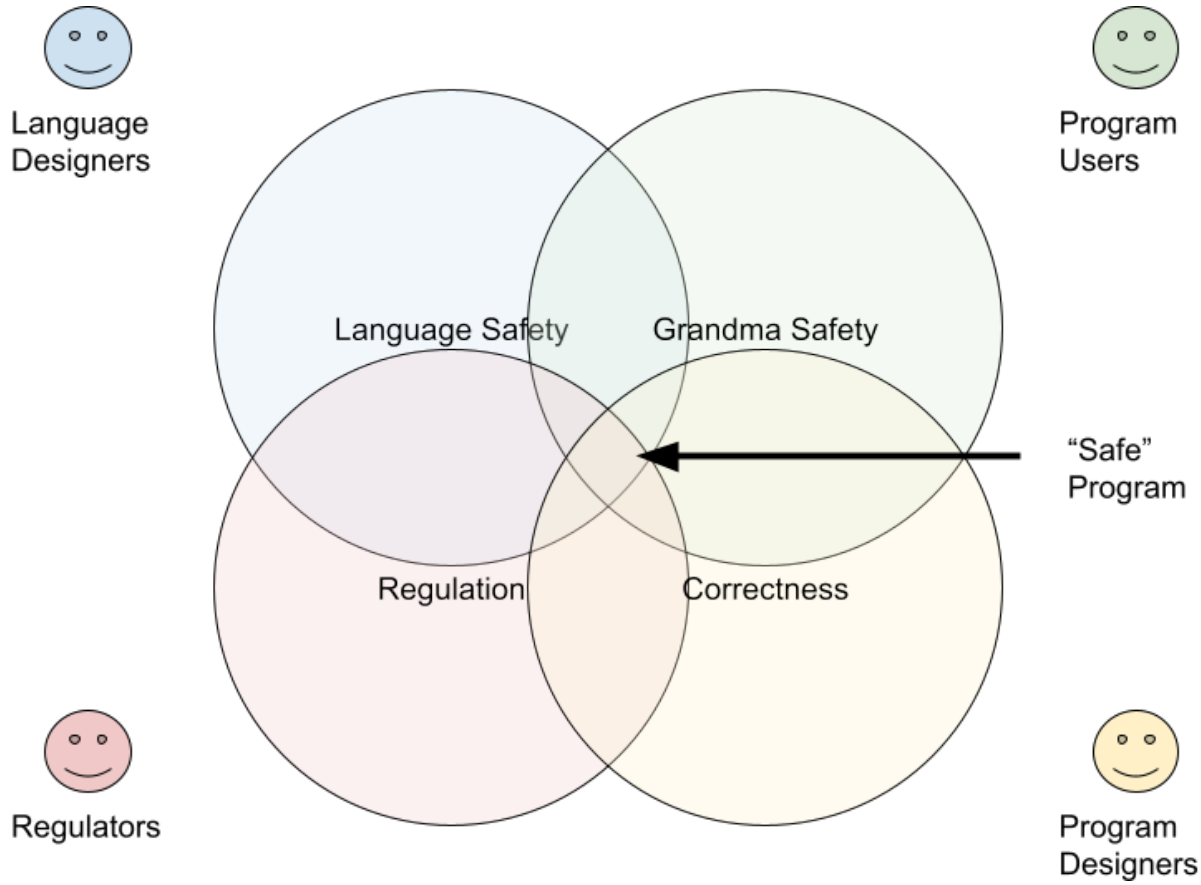
In order to have effective conversations that are not confused and contentious – with unstated assumptions and biases swirling and muddying our best intentions – **we must understand and use the correct specific terms**.

We define **Language Safety** to be the set of all of Type Safety, Memory Safety, Thread Safety, Lifetime Safety, Bounds Safety, Undefined Behavior, etc. – **all the things that the programming language is able to specify and enforce at compile time** (even if it means generating runtime code at compile time). Language Safety works differently in different languages (eg. C, C++, and Rust have different kinds of Language Safety).

We do not consider UB anywhere in this paper, because the existing standard terminology is C++ agnostic. UB is a largely C++-specific term, and does not necessarily apply to the many other programming languages used to make “safe” systems in practice. **UB is what happens in the gaps where Language Safety is not enforced**.

We define **Grandma Safety** (ie. *Functional Safety*) to contain the set of **all software behaviors that keep people, places, or things – ie. your Grandma – from harm**. These behaviors interplay with correctness, regulation, security, and occupational safety.

A feature proposed for C++ may (or may not) address either one differently, and **addressing one more than the other is not a failure of the feature**.



- A program can be "Language Safe" and still cause harm to Grandma.
- A program can be correct (ie. built to specification), and still (unintentionally) cause harm to Grandma.
- A program can be well regulated (ie. compliant), and still cause harm to Grandma.
- A program can be safe for Grandma – and still not be fully language safe, fully correct, or fully regulated.
- etc…

For many applications, the intersections in the diagram are empty – but for others they aren't. A battlefield drone is an example of a system correctly built to specification that is not functionally safe. A program could have UB and still be functionally safe, because in practice, the program is test, test, test, test-ed on actual hardware in The Real World. The purpose of the diagram is to justify why safety critical systems use precise words with specific meanings. See the section on Correctness for more.

A program that is "Grandma Safe" does not have behavior that could harm Grandma. The level of abstraction needed to describe "don't harm Grandma" is too far *above* what a language designer could know (and too far *below* what a regulator could know).

“Grandma Safety” happens when the specification matches The Real World Requirements, the software meets the Specification, and repeated cycles of Testing offers acceptable Assurance.

Contract checking (P2900 or otherwise) can be used to check that some code meets Language Safety:

```
CHECK(index < container.size());  // Check and enforce Memory-Safety.
```

or Grandma Safety:

```
CHECK(component.IsHealthy());  // Check and enforce Functional-Safety.
```

The former example could be rephrased in terms of other memory-safety techniques, whereas the latter *cannot* – the semantics of what “component is healthy” means are beyond what the language designer or compiler engineer could know.

In practice with real time systems, “component is healthy” often means “I was able to establish message based communication with the component, within a certain time frame, with low enough latency, no transmission loss or corruption, and the component able to report to me the values that I expected.”

If what you want is Language Safety, and you do not care about Grandma Safety – please focus on improving Language Safety, and do not derail features that successfully address Grandma Safety.

The Real World

Historically, most software has run on developer or cloud machines for applications where failure was not serious (ie. “move fast and break things”) – but increasingly software is being run on machines that run on Edge Computing devices in The Real World (eg. automobiles), and where breaking things has serious consequences.

Software is built by technologists, but **users expect software to do what it’s supposed to** – out in the everyday world, and does not hurt or betray us. **It’s not enough that any large system is technically sound**, the entire system taken as a whole, and running in a representative environment (ie. The Real World), must be **free from harm**.

The annoying thing about The Real World is how little we seem to understand it. As humans we have an internal model about how The Real World should work – but in practice our model more closely reflects our internal biases. The history of engineering safety is written in the blood of those harmed when an engineer’s model failed to match reality because of some unforeseen factor. Those bloody mistakes are regularly codified in regulation.

Assurance

Assurance is the process of building a probabilistic measure of how well the software meets its requirements in The Real World. For example, if I drive a prototype automobile 5km in Death Valley under 50C heat, then I'm 100% certain I can drive that prototype 5km in Death Valley under 50C heat at least once. I'm slightly less certain that I can drive the production automobile indefinitely in *any* desert conditions – *but it's a start*.

If I run a test as follows:

```
CHECK(f(5) == 42);
```

Then I am 100% sure that f(5) is 42. The more I'm able to expand the test suite, the more assured I can be of more and higher level properties.

It's possible to have a program with UB in it – yet still be Functionally Safe – because of Assurance: the program is run on specific hardware with specific compilers so many times, that – with a high level of probability – even relatively uncommon bugs will eventually manifest as test failures. For example, if there was UB in an automobile, and that automobile was driven for thousands of kilometers, then the UB is likely to eventually manifest and fail the test.

While it's always helpful when technology can automatically identify and exclude incorrect programs, **a program is only Assured when it has been test, test, test, test-ed in an environment that is representative of The Real World**.

The Safety Spectrum

Bottom-Up Safety

Computer scientists and language specialists most often see the word "safety" hyphenated in such terms as "Type-safety", "Memory-safety", "Thread-safety", "Bounds-safety", "Lifetime-safety", etc. It is quite natural that when we see unadorned word "safety" we translate that internally to "*-safety" – and "safety" in our minds is short-hand for "the set of all the common *-safety techniques common to modern programming languages". Since UB is "bad", and a number of these techniques prevent UB from happening, many of us think "safety is what keeps UB from happening".

The common theme among all these "*-safeties" is that they are programming language techniques. They are things we can design into a programming language, and have the compiler enforce somehow – either by refusing to compile the program, or by injecting runtime checks – or other implicit runtime code generation. In this paper we call this **"Language Safety" – the set of things the programming language has the power to enforce**.

What these “*-safeties” have in common is that they limit the kinds of programs that will run, specifically **certain kinds of incorrect programs cannot run**.

```
std::vector<int> v;
v[42] = 5;
```

Currently, in C++ this *will run* – but exhibit UB. If we made C++ memory-safe, this would either not compile – or fail to continue beyond where the UB was encountered.

However “Language Safety” *cannot* ensure that correct programs are the *only* programs that run.

```
std::vector<int> v(100);
v[42] = 5;
```

This program is no longer demonstrably incorrect due to UB, but is it fully *correct*? Only the program designer could know. The level of understanding a compiler would require in order to answer that question is orders of magnitude and years away from our foreseeable technology.

And even then, the program designer may not even be sure it’s correct: what if the specification is wrong, or doesn’t match reality? Assuming a complete and accurate list of software requirements exists, the software will still need to be test, test, test, test-ed until it passes Assurance.

Top-Down Safety

In The Real World, the unadorned word “safety” is assumed to have a different non-technical definition: freedom from physical injury to person, place or thing (the word “safe” comes from Latin salvus "uninjured, in good health"). **This is the definition most of the world uses**.

When software preserves the property “uninjured, in good health” for people, places, and things (ie. “freedom from harm”) – **we call this “Grandma Safety”.** In formal literature this is called Functional-safety (ie. does the software *function* as intended when in a safety-critical context in The Real World)?

In Language Safety the language designers and compiler work to enforce this -safety. Who enforces Grandma Safety?

1. The Government (on behalf of Society), when they enact Regulations.
2. The Program Designer, when they build complete Specifications from accurate Requirements.
3. The Programmer, when they write Correct code that meets Specifications.

4. The Test Engineer, then the test, test, test, test until it passes Assurance

Whereas Language Safety is a bottom-up technical constraint enforced by technical means, Functional Safety is top-down encoding of high level needs into software. We saw that compilers can only reject certain kinds of *incorrect* programs – they **cannot ensure programs are correct, because they do not understand** the high level human requirements for what it means to be “Grandma Safe”.

Correctness

A program is correct when it meets all of its Requirements. However in practice, Requirements may be missing or misinterpreted from the Specification. In a sufficiently complicated system, a program is only assured to be correct when it’s test, test, test, test-ed in a representative environment, and found to be conforming to all its Acceptance Criteria (ie. the test conditions that match the Requirements) – ie. passes Assurance.

It is *possible* to build Functionally Safe and correct software using assembly code running on a TRS-80 that can take human beings to the moon and back – *without any technical enforcement such as Language Safety* – all you need is to test, test, test, and test until it passes Assurance. However, final stage testing is really expensive, and it’s way cheaper to shift discovery of errors earlier in the development timeline.

Language Safety helps us avoid incorrect programs earlier, and makes development cheaper. Technically speaking, it’s *neither necessary nor sufficient for correctness* – but practically speaking, Language Safety is *necessary but not sufficient* for correctness.

Conversely, a program that correctly meets its Specification is not necessarily Functionally Safe either!

For example I could specify that dangerous behavior is expected or required (eg. a killer military drone), or I could have incorrectly specified the requirements in a way that leaves a gap for dangerous behavior. In Functional Safety, a Systems Engineer would use Failure Analysis to try and find all failure scenarios to ensure the requirements are never underspecified. In practice it is challenging to both

1. Find and specify all failure modes.
2. Ensure the software is fail safe under all specified failure modes.

Technically speaking, *correctness is necessary and sufficient* for Functional Safety, in practice, *correctness necessary but not sufficient* – that is, you can’t fully trust the specification – you’re still going to have to test, test, test, test a lot before you can be assured the software is functionally safe in The Real World.

The intuitive definition of "correctness" is the platonic ideal (ie. how god would have written the program). In this sense, if the Specification is wrong then the program cannot be "correct" – but in practice the platonic ideal is not helpful because humans build systems. There are different specialists responsible for Specification and Implementation. If there is a defect in the specification process, then the specialists involved in specification are responsible – namely the systems engineers.

Most modern technical disasters are due to a complex interplay of technical components that the designers did not foresee. In the Boeing 737 MAX disaster, the angle-of-attack sensor caused some automatic software to incorrectly nose-down, which the pilot struggled (and failed) to defeat. The engineers who built both systems did so according to specification – and as far as they were aware, their systems were "correct". There were many non-technical reasons for the Boeing 737 MAX disaster, but the proximate technical cause was failure of systems engineering to fully realize how the failure of the single angle-of-attack sensor could affect MCAS, with the pilot under stress.

Contract Checking

Is a Language Safety Feature

Language Safety is the set of things the language designers allow the compiler to enforce by stopping incorrect programs from continuing. Contract checking (P2900 or otherwise) injects programmer specified runtime checks at compile time. If a programmer specifies a contract, and the check determines the contract is in breach, then the program will terminate (among other options).

In a multi-MLoC safety critical codebase, a vast majority of (macro-based) contract checks check for memory-safety violations:

```
std::pair<T,U> get_ith_pair(const std::vector<T>& ts,
                            const std::vector<U>& us,
                            std::size_t i) {
    CHECK(i < ts.size())
    CHECK(i < us.size())
    return {ts[i], us[i]};
}
```

This kind of memory issue could be avoided by using higher level primitives and built-in type safety – for example ranges. This observation is correct, but misses an important reality: in practice software gets made first, and hardened sometime after. Adding checks is quicker than a potentially "viral" refactoring. If the previous code was already test, test, test, test-ed, until it passed Assurance, then refactoring has the potential to accidentally make the code less correct (ie. further diverged from specification).

Is Simple (not Easy) Language Safety

A bicycle is **simple** – it's made of a few easy to understand and fix parts, but can take effort to travel long distances. A car is **easy** – it's complicated and hard to fix, but takes very little effort to travel long distances.

Contract checking (P2900 or otherwise) allows adding ad-hoc predicate checks to any function, where a failed predicate results in the incorrect program being terminated. This is conceptually simple, follows decades of software best practices, can be added anywhere at any time, and is flexible enough to be introduced to any code base.

However applying memory-safety to a large project by means of contract checks would be an effortful, manual task. A programmer would have to inspect every memory-safety issue and manually describe it as a predicate. This is a lot of effort for a new code base, but it may be completely infeasible for an existing code base.

If you want Easy Language Safety, you probably want Profiles or P3100.

Is not Ideal Language Safety

Language Safety by means of manually adorned predicates takes a lot of effort to get the expected result. And programmers are fallible, so what if they miss something that a compiler could have caught?

If our concern is {Type, Memory, Lifetime, Thread}-safety, and we had a magic wand, we'd want the **compiler to do as much as possible to automatically check** all bounds and lifetimes – which probably looks something like rust-like extensions for C++.

Can be Turned Off

In all versions of contract checking as a feature, there are means to "turn it off" – ie. either not generate runtime checks, or not terminate when those runtime checks fail. If you mistake Contract Checking solely as a Language Safety feature, this ability may shock you – why would you ever want to make your code "less safe"??

The answer is that **Contract Checking is not intended to be a Language Safety feature**. Contract Checking is a Correctness feature, and because the programmer's understanding of what it means to be a correct program changes as requirements evolve, the contract statements themselves may become incorrect.

In practice, the phasing in or out of new requirements creates a period of instability during testing where the semantics of the code and the contract predicates are not in sync. These periods of testing are not intended to be seen by the end user. When the program has been

test, test, test, test-ed until it passes Assurance and is running in The Real World, only then is it Correct and can be deployed to end users.

Turning off contracts does not "turn off Correctness", because Correctness is only ever established during Assurance – ie. lots of representative tests. Contract checks are a tool to put the specification in the code where it can be checked during testing – no technological tool can force you to employ good tests as part of a comprehensive assurance process.

Is for Correctness

Contract checking (P2900 or otherwise) is a way to stop incorrect programs from continuing to run. As a side effect this can halt programs that are incorrect because their memory usage is unsafe, and as such could be used to manually implement memory safety.

This is a flexible way for enforcing ad hoc Language Safety – but it is *manual*, and as such takes a lot of effort – and neither the language designers, compiler engineers, or even library implementers can make any assurances about the program (from their limited perspective). **Contract checking *is* a tool for making our programs *more* Language Safe, but that is not the goal**.

The goal of contract checking (P2900 or otherwise) **is to encode and actively check the Specification, alongside the C++ that implements it**. Contract checking stops the program from continuing when it detects that the software no longer meets its specification in high level terms – that is, when it is incorrect – as the program designer intended. **Contract checking enforces Correctness beyond Language Safety**.

For example, the following check statement says "my program meets its specification when the index does not exceed the size of the container it indexes".

```
CHECK(index < container.size());  // Check and enforce Memory-Safety.
```

Arguably however, this is over specified. **In a memory-safe language you don't need to explicitly manually specify this.**

If you have an existing program that uses container indexing like this, you can incrementally and ad hoc improve its memory safety by adding such a check statement. That's *simple*, but not *easy*. If you wanted all your container indexing to be automatically memory-safe, there are easier techniques – such as ranges, profiles, or rust-like behavior.

However, the following statement says something like "my program meets specification when I am able to establish message based communication with the component, within a certain time frame, with low enough latency, no transmission loss or corruption, and the component able to report to me the values that I expected."

```
CHECK(component.IsHealthy());  // Check and enforce Functional-Safety.
```

There is *no* way for Language Safety to implicitly enforce this specification because the concepts are too high level for any current language or compiler to comprehend. **This kind of specification must be encoded manually by the programmer**.

But that's just, like, Your Opinion… man

"Language Safety" and "Grandma Safety" are words I made up to help drive a point.

The rest of the words – Requirements, Specification, Fault Analysis, Acceptance Criteria, Assurance, Testing, Functional Safety, {Type, Memory, Lifetime, Thread}-safety, etc. – **have all been industry standard terminology for decades** in safety critical industries like aerospace, defense, health, high energy, and automotive.

As the level of automation in our world increases, and we as a society interact more with automated physical devices – definitely healthcare devices, autonomous vehicles, and possibly even humanoid robotics – we as C++ designers and developers will only need to interact with these terms more often. **It is not in our own interest to make up new words or assign divergent meanings.** We owe our community to use the correct existing widely understood terminology.

Functional Safety is the correct term for software behaviors that do not harm Grandma, and C++ must address this need. Contract checking (P2900 or otherwise) is that feature.

If what you want is Language Safety, and you do not care about Grandma Safety – please focus on improving Language Safety, and do not derail features that successfully address Grandma Safety.