Settle
Doc. No. P3274R0
Date: 2024-5-5
Audience: SG23, EWG, LEWG
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

A framework for Profiles development

1. Introduction

This document outlines a structured framework for developing and implementing safety profiles in ISO standard C++. It addresses the industry's urgent need for improved safety and the challenges of standardization.

Profiles (e.g., see P2687r0 , P2816R0 , and P3038R0) can deliver guarantees; that's how they differ from guidelines and from many tools for detecting errors. To deliver the strongest guarantees, such as complete type-and-resource safety, we need analysis that may not fit into current tool chains and may require sophistication that is not available at scale.

IMO, we can't wait for a standard offering standardized profiles. The industry – partly because of governmental pressures – needs to move faster for our efforts to demonstrate some forms of C++ safety to be credible. Only verified safe systems (for some reasonable definition of "safe") deployed at scale are credible. We need a framework for safety (i.e., Profiles) or we will soon have a mess of incompatible ways of enforcing incompatible versions of the notion of safety, compromising C++'s strengths in standardization and portability.

To act in a timely manner, we must start by picking "low-hanging fruit" based on industry needs, feasibility of implementation, and cost of introduction into existing code bases. That will give early improvements and gain us experience for further, more complete, guarantees.

A framework is essential to enable gradual adoption of safety measures, to support incremental improvements of guarantees and support tools, to become a tool for fighting the problems of technical debt affecting safety, and to ensure portability of guarantees across tool chains.

Better education would of course help, and is necessary whatever we do, but on its own it cannot offer guarantees. We need a comprehensive and tool supported approach based on analysis, libraries, and techniques to offer guarantees: In other words, Profiles. Similarly, benefits can be achieved without modifying source code (e.g., we definitely should offer range-checked **vector**, **string**, and **span**), but we have several decades of experience that says that bugs hide in complex, low-level code, even when we try our best to eliminate them. Any serious approach to address such technical debt must involve simplification of code (supported by better analysis and libraries offering guarantees). Again: Profiles.

There are applications, organizations, and subsystems that don't feel the need to make the changes necessary for verified guarantees and others that cannot manage to make such changes just now. For those, the possibility of optional and of gradual adoption is essential.

Naturally, we would like to have universal guarantees, that is, guarantees the holds for every part of a program. Unfortunately, that's impossible for most programs. We need to call subsystems and libraries that are not under our control, such as the operating system, a graphics system, a database, specialized hardware support, code compiled by different compilers, code written in different languages, etc. The best we can do is to minimize that where safety (for any definition of safety) is paramount and identify calls to such code.

Note that we are talking about guarantees in code, rather than the security of a complete system. For security, I worry most about subjects like spearfishing, hostile individuals placed in development organizations, and tool-chain insecurities. Such avenues of attack are beyond the scope of this work.

Note again (P3038R0) that profiles offer guarantees, not optional conformance. In theory, the required analysis could be done by human readers, but to scale, we need tool support, so such support is our focus here.

Importantly, the syntax and semantics of the code accepted by a set of Profiles is ISO C++. Profiles do not change semantics beyond possibly giving manageable meaning to UB (e.g., range checking).

Next steps:

- Work out details for individual Profiles (so that we don't have to chase through the older papers for detailed information) and document them in separate focused papers.
- Maintain this paper as documenting the overall design (with revisions).
- Encourage experimental and exploratory implementations.

See also §5 "Next Steps."

1.1 Verification

Please note that complete type-and-resource safety is impossible for arbitrary C++ programs, especially for programs indulging in complex pointer usage. This implies that any analysis must classify code fragments into three groups:

- Verified safe (for the definition of safety offered by the profile used).
- Verified unsafe.
- Verified possibly unsafe: Too complex to classify as safe or unsafe.

By "verified" I mean proven by some sort of static analysis (possibly by the compiler). Obviously, the third category is the troublesome one. It requires human intervention to explicitly accept code that isn't verified (mark some code fragments as "trusted") or (preferably) to rewrite the code to eliminate the problem.

One big issue for initial implementations is that static analysis won't be perfect, so that we have to choose between two evils:

- *False positives*: Should we restrict language usage, rejecting safe programs?
- *Uncaught errors*: Should we accept incomplete analysis, letting erroneous programs slip through occasionally (e.g., memory leaks or type errors)?

I am strongly in favor of the first choice but realize that in cases we may have to – for a while – regress to the second. Why? Imagine that we can eliminate 99% of a kind of error so that a programmer may only encounter an uncaught error of that kind once every half year. That's valuable, but programmers would forget to suspect that kind of errors, thus have seriously increased trouble finding them. They may even forget common techniques to avoid them. This will induce distrust of the tools. On the other hand, we would have eliminated 99% of the bugs. How can we manage this dilemma? There are several ways, but for profile directives in the code, we could simply use separate names, e.g., **[[Profiles::enable(ranges)]]** to request the proper guarantee for range checking, and **[[Profiles::enable(ranges, weak)]]** to request a weaker, temporary, and/or experimental version. The weaker guarantee that can still be much better than nothing and available much sooner.

Why 99%? Well, I could have said 90% and still made the point. What might be an example? Consider **concurrency**. Many, but not all, deadlocks can be handled by using **scoped_lock** and many, but not all, data races can be handled by passing **unique_ptr**s and containers without invalidating operations (e.g., **gsl::dyn_array** rather than **vector**) together with the simplest static anti-aliasing checks.

Static guarantees can, and should, be achieved through local static analysis. For logical reasons and compile-time performance, global static analysis (whole-program analysis) doesn't scale: Most C++ users do not have access to every line of code potentially executed, such as libraries, operating systems, and device drivers. Also, many systems employ dynamic linking). There are applications for which these problems and constraints don't apply, but to serve the vast majority of users, we must accept the limitation to local analysis.

We must exactly specify the analysis required. We cannot have the validation of a Profile depend on the cleverness of individual implementations and vary significantly among implementations. Naturally, in some cases, we might need to apply analysis that is beyond the ability of most analyzers for applications with specialized needs or to accept code that can be verified safe only after massive time spent in analysis (thus minimizing false positives caused by incomplete analysis). Tools for such analysis beyond what can be supported by most compilers can be introduced into tool chains where needed.

Obviously, we must minimize false positives and keep analysis cost reasonable (relative to needs) if we want our static analysis to be widely used.

One way to minimize false positives is to deploy the static analysis only when explicitly requested (through annotations or options), rather than unconditionally putting all code through the straitjacket imposed by the most rigorous needs. The latter would not be feasible: it wouldn't allow gradual adoption or scale to cope with the large systems for which guarantees are most needed. After asking for specific guarantees, a programmer would (hopefully) consider significantly far fewer of the flagged problems false positives. For code where a guarantee is not considered feasible, affordable, or valuable the checking would simply not be deployed (and the resulting code would not be used where verification is required).

Any guarantee can be suppressed. That's essential (e.g., for some low-level programming and for the implementation of some linked data structures), but we want to minimize that. Having large amounts of "trusted code" is a source of errors. Profiles offer unverified sections of code for that.

1.2 Annotations

My ideal is that guarantees are explicitly requested in the source code. That way the intent is obvious and (eventually) well-defined and portable. The suggested annotations use the ISO standard attribute syntax, e.g., **[[Profiles::enable(ranges)]]**. Earlier papers used syntax that wasn't 100% standards conforming. That was not a deliberate incompatibility nor a necessity, just a simple way of expressing the early ideas. One of the early next steps is to settle on an annotation syntax.

In places, there is a need for guarantees for code that cannot be modified (e.g., code owned or maintained by others) or needs to be compiled by compilers that have not yet been modified to recognize Profile directives in code. In such cases, compiler and/or build options need to be used.

The "contemporary C++" way of doing that would be to have profile options on module compiler directives. More traditional environments would have to use conventional, header-based TUs as the unit of profile enforcement. In such cases, unverified code would have to be placed in separate TUs.

Without a common framework, the options from different suppliers addressing common problems (e.g., range errors) will be significantly different, and code relying on one cannot be relied on to work on another or to offer the same guarantees. This serious problem can be addressed by the options implementing specific Profiles, just as if they had been specified in-code. When options are used, we need to restrict Profiles to be used on a per-module or per-file basis. Alternatively, we could define some macros to specify more restricted scopes (yuck!).

I use the attribute syntax, e.g., **[[Profiles::enable(ranges)]]** rather than a "proper language syntax using keywords" (possibly context dependent keywords), e.g., **profiles enable(ranges);** to ease experimentation and gradual introduction. This choice of syntax does not imply that enforcement is optional in implementations that support profiles. However, using the attribute syntax allows us to have a single code base for code that must be compiled with a variety of compilers, some pre-profiles. In particular, it allows us to support a likely important use case: First check your program with the best compiler for profile support, then port the code and compile with an older compiler. I prefer that to **#ifndef** hackery.

So, for each of the Profiles below, I propose that its validation can be triggered by an annotation or an option.

1.3 Initial profiles

"Meta questions" about the profiles listed below:

- Are these the right profiles?
- Which profiles might be missing?
- Which profiles should we start defining and implementing initially?

Part of the purpose of this framework for development is to allow people to work independently on different parts and to allow people to collaborate on specific parts. I consider this important pre-standardization. Another purpose is to facilitate discussion and to allow people to focus on their interests and expertise. Without a general framework, we will be left with a patchwork of incompatible and probably incomplete safety checks.

Given the current heavily published emphasis on "memory safety," It would be wise to pick the "low-hanging fruit" based on industry needs, feasibility of implementation, and cost of introduction into existing code bases. I point to Initialization (§3.7), Ranges (§3.4), and Pointers (§3.5) as likely initial targets. Personally, I'm focusing on the likely technically harder profiles: Algorithms (§3.6) and Invalidation (§3.9). I think there is also some low-hanging fruit in Algorithms (§3.3).

As ever, we must remember that the programming language is only a part of the total safety and security picture. Here, I say "the language" because I am concerned with C++, but of course most large systems are built using a variety of languages and tools. We need better tool chains (including compiler support) and better development processes, but improvements to the language and its foundational libraries are critical areas. For example, we need range-checked container access and checking casts. Such might by implicitly used by compilers to add safety guarantees to conventional code.

Also, we must remember "memory safety" is more than just range checking and nullptr guarantees, yet less than complete type-and-resource safety. Complete plain "safety" is either ill-defined or impossible for real-world code.

1.4 Profile specification format

Below, I discuss a number of potential profiles. For each, I use this format:

- ***Name***: the name of the desired profile (we may have to use a slightly different name for an initial/limited version).
- ***Definition***: a specification of the guarantees offered.
- ***Implications***: what we need to do to offer the guarantees.
- ***Initial version***: suggestions for an initial/limited version that can be implemented relatively simply in current tool chains.
- ***Observation***: an additional comment, if needed.
- ***Question***: a question about the design, if any.
- ***Detailed specification***: where to find the detailed specification of the guarantee and the tests required to enforce it if such has been constructed.

In all cases, I expect that description to evolve over time. However, it should allow us to focus on one problem at a time rather than being paralyzed by the total size of the project.

I imagine that each profile will eventually get a ***Detailed specification*** document or appendix, leaving later versions of this paper as an overview and rationale.

In many cases, further details of what I'm suggesting can be found in the papers P2687r0 , P2816R0 and P3038R0.

Note: when I say "range" I mean "a sequence of elements" rather than just the ranges from **std::ranges**.

2. Run-time checks

Some guarantees require run-time checks (e.g., range checks). We prefer static checks, but there has to be a mechanism for dealing with failed run-time checks. I use "triggers error" to mean: an error reporting mechanism will be invoked. This could result in an exception throw, termination, or even

something else, depending on system setup. If WG21 manages a consensus on contracts, that may be a possible mechanism. Maybe a more specialized facility would serve better.

For example, and only to illustrate the idea rather than a specific proposal, I used this for some applications (from https://github.com/BjarneStroustrup/flats/tree/main):

```
template <Error_handling action = default_error_action, class C>
constexpr void expect(C cond, Error_code x)              // C++17; a bit like assert()
{
 if constexpr (action == Error_handling::ignoring)
    return;
 else if constexpr (action == Error_handling::logging)  {
    if (!cond())
        std::cerr << "Flats error: " << int(x) << ' ' << error_code_name[int(x)] << '\n';
    return;
 }
 else if constexpr (action == Error_handling::testing)  {
    if (!cond()) {
       std::cerr << "Flats error: " << int(x) << ' ' << error_code_name[int(x)] << '\n';
       throw x;
    }
    return;
 }
 else if constexpr (action == Error_handling::throwing) {
    if (!cond())
       throw x;
    return;
 }
 else if constexpr (action == Error_handling::terminating) {
    if (!cond())
       std::terminate();
    return;
 }
 else
    static_assert(False<>, "bad error handling option");
}
```

For now, what is urgently needed is a notation for indicating a violation.

A required run-time check may not be eliminated. In particular, time-travel optimization based on UB that could eliminate run-time checks must be prevented. That could be very hard to achieve.

It must be possible to avoid stack overflow. This could be done by a hardware check. If it is possible for a stack to overflow into space occupied by data, it must be part of type safety (because it could scramble objects).

3. Profiles

Many of the profiles listed below are relatively small, logically separate units, rather than collections of guarantees combined for easy remembering and use. I think we need a name for such units. I suggest the term "Profile fragments" for sets of guarantees that are primarily intended for building other profiles than for direct use by users.

It is essential that users are presented with a manageable set of profiles, rather than a set of 50 or 100 individual checks (allowing 2^{50} or more alternatives). Only a few combinations of checks add up to manageable sets of guarantees.

Profiles must be designed to work in combination. In particular, we can't have two profiles that have contradictory requirements. Exceptions to profiles should be handled by unverified ("trusted") sections of code.

Different profiles will depend on the same parts. In a well-designed set of profiles, such fragments and profiles will constitute a DAG. For example, essentially all profiles will depend on Initialization and Type will depend on Ranges and Pointers.

3.1. Profile: Type

- ***Definition***: every object is used only in accordance with its definition.
- ***Implications***: This is a very strong guarantee that requires a combination of static and run-time checks. It will almost certainly be a union of simpler profiles, such as Ranges (§3.4), Invalidation (§3.9), Algorithms (§3.6), Casting (§3.8), RAII (§3.10), and Union (§3.11).
- ***Initial version***: Start with Initialization (§3.7), Pointers (§3.5), and Ranges (§3.4) and add other initial profiles as they become available.
- ***Observation***: To achieve complete type safety, we need to eliminate subscripting of raw pointers outside the implementation of abstractions such as **span** (because in general we don't have the information needed to range check built-in pointers), null-pointer dereferences, use of invalidated pointers, dangling pointers, and memory leaks (leading to resource exhaustion). That eliminates essentially all traditional C-style code, but not all that much code written following the recommendations post-C++11 (e.g., using algorithms, resource-management pointers, ranges, range-checked containers, RAII, and avoidance of casting).
- ***Observation***: We can have "full traditional C++" or complete type-and-resource safety, but not both at the same time.
- ***Question***: Is heap exhaustion a type error? Not if it is handled, as it can and must be in some applications where termination isn't an option.
- ***Question***: Is a resource leak a type error? It is because it can lead to unrecoverable constructor failures. Systematic use of RAII takes care of most potential resource leaks.

3.2. Profile: Arithmetic

- ***Definition***: Over and underflow trigger errors; narrowing conversions trigger errors.
- ***Implications***: we need the compiler to use checked arithmetic.
- ***initial version***: We might use some checked arithmetic libraries. Possibly such libraries could be used by code generators when the arithmetic profile is used.

- ***Observation***: we need **round()** and **truncate()** functions. Checked arithmetic libraries exist but are not standard.
- ***Observation***: Mixing **signed** and **unsigned** in expressions is a serious source of errors. In particular, the use of **unsigned** for standard-library sizes causes difficult problems.
- ***Questions***: should "Arithmetic" imply "Cast"? Suggested answer: yes; see §3.8 Casting.

3.3. Profile: Concurrency

- ***Definition***: no data races. No deadlocks. No races for external resources (e.g., for opening a file).
- ***Question***: should we also deal with priority inversion, delays caused by excess contention on a lock? Suggested initial answer: no.
- ***Observation***: The concurrency profile is currently the least mature of the suggested profiles. It has received essentially no work specifically related to profiles, but concurrency problems have received intensive scrutiny in other contexts (including the Core Guidelines and MISRA++) so I can offer a few suggestions for initial work:
    - ***Threads***: prefer **jthread** to **thread** to get fewer scope-related problems.
    - ***Dangling pointers***: consider a **jthread** a container and apply the usual rules for resource lifetime (RAII) and invalidation (§3.9).
    - ***Aliasing***: statically detect if a pointer is passed to another thread. For an initial version, that will require restrictions on pointer manipulation in non-trivial control flows. In general, not all aliasing can be detected statically, and we need to reject too complex code. Defining "too complex" is essential, or we will suffer portability problems because of compiler incompatibilities. See "Flow analysis" (§4).
    - ***Invalidation***: use **unique_ptr** and containers without invalidation (e.g., **gsl::dyn_array**) to pass information between threads.
    - ***Mutability***: Prefer to pass (and keep) pointers to **const**.
    - ***Synchronization***: use **scoped_lock** to lessen the chance of deadlock. Look into the possibility of statically detecting aliases in more than one thread to mutable data and enforce the use of synchronization on access through them. Use **unique_ptr** combined with protecting against aliasing across treads.

  We need to look at lock-free programming.

3.4. Profile: Ranges

- ***Definition***: out-of-range subscripting using **[]** triggers errors
- ***Implication***: we cannot accept subscripting of raw pointers.
- ***Implication***: Time-travel optimizations based on UB must be eliminated to ensure that the run-time checks are always executed.
- ***Initial version***: Ban subscripting of raw pointers, check **[]** for **vector**, **span**, views, and **string**.
- ***Question***: what about **std::array**?
- ***Question***: Should "Algorithms" be part of ranges? Suggested answer: yes.
- ***Question***: Should Pointers (§3.5) be part of Ranges (§3.4)? Suggested answer: no, or "partially"; we can get range checking of library-implemented subscripting much sooner than complete pointer guarantees.
- ***Observation***: Just checking every subscript operation will impose unacceptable run-time costs for many applications. Therefore, individual checks must be supported by checks of operations

being in-range (by a single check). This means that rang-for and range algorithms should be strongly encouraged. Intelligent compilation of many conventional (C-style) loops is also possible.

3.5. Profile: Pointers

- ***Definition***: every pointer points to an object or is the **nullptr**; every iterator points to an element or the end-of-range; every access through a pointer or iterator is not through the **nullptr** nor through a pointer to end-of range.
- ***Implication***: Eliminate pointer arithmetic outside trusted regions (e.g., outside **span** implementations).
- ***Implication***: Eliminate dangling pointers (See §3.9 Invalidation).
- ***Initial version***: require **not_null**, require use of range version of algorithms, require use of a range-checked **span,** range-**for**, or algorithms when using a range.
- ***Observation***: Whatever guarantees are offered for pointers must equally apply to objects that are not built-in pointers but identify objects. For example, lambda captures and **unique_ptr**; see P2687r0 §5.3.
- ***Observation***: Just checking every dereference operation will impose unacceptable run-time costs for many applications. Therefore, techniques for checking a pointer once for many uses must be supported. **not_null** is one way to go, smart compilation is another. Both should become widely used to contain the cost of this necessary guarantee.

3.6. Profile: Algorithms

- ***Definition***: No range errors from mis-specified ranges (e.g., pairs of iterators or pointer and size). No dereferences of invalid iterators. No dereference of iterators to one-past-the-end of a range.
- ***Implications***: Use ranges consistently, rather than pairs of iterators to identify sequences and single iterators to indicate output targets. Access to ranges must be range checked.
- ***Initial version***: Provide range versions of all standard library algorithms. Provide an **not_end(c,p)** check whether **p** is the one-past the end of **c** for every container (**c**) (that **is p!=end(c)**) to help static analyzers verify that iterators used as return values are checked before any dereference (P2687r0).
- ***Observation***: infinite ranges (e.g., some **ostream**s) and automatically growing ranges (e.g. writing to a **back_inserter**) do not require range checking (at this level of abstraction).
- ***Detailed specification***: For a very early start on such a specification, see the appendix.

3.7. Profile: Initialization

- ***Definition***: Every object is explicitly initialized (a default constructor is considered initialization). **Implication**: Constructors must (recursively) be checked to ensure that every member is initialized (once only).
- ***Initial version***: For the initial version, simply trust that a constructor properly initialize all members. For the final version: use static analysis to ensure that every member is initialized.
- ***Observation***: This guarantee is essential for every profile.
- ***Observation***: Guaranteed initialization severely cuts down UB.
- ***Observation***: Cleverer and less restrictive schemes to guarantee initialization are possible, but more complex to implement and for programmers to manage.

- ***Observation***: Avoiding explicit initialization though the implicit initialization is in itself error prone because a default value may be a valid but wrong (leading to wrong results), because not all user-defined types have a suitable default value, and because it can lead to double assignment of buffer values (considered unacceptable in some high-performance applications).
- ***Observation***: In performance critical applications, it is not feasible to initialize output buffers before use. This must be handled through suppression, an **uninitialized** annotation, and/or by specific uninitialized types.

3.8. Profile: Casting

- ***Definition***: No cast grants access to an object in a way that disagrees with its definition. No cast yields a result with a value that doesn't compare equal to its source (aka, "no narrowing").
- ***Initial version***: Ban all casts. Provide run-time-checked casts (e.g., gsl::**narrow<T>(x)**) to cope with potential narrowing and potential **signed**/**unsigned** problems.
- ***Observation***: **dynamic_cast** is safe but we must enforce checking of pointer results.
- ***Observation***: If you need an unchecked, unconditional cast, you must suppress this guarantee.
- ***Observation***: We need casts to convert from untypes ("raw") memory to typed objects.

3.9. Profile: Invalidation

- ***Definition***: No access through an invalidated pointer or iterator
- ***Initial version***: The compiler bans calls of non-**const** functions on a container when a pointer to an element of the container has been taken. Needs a **[[non-validating]]** attribute to avoid massive false positives. For the initial version, allow only straight-line code involving calls of functions on a container that may invalidate a pointer to one of its elements (P2687r0).
- ***Observation***: In its full generality, this requires serious static analysis involving both type analysis and flow analysis. Note that "pointer" here means anything that refers to an object and "container" refers to anything that can hold a value (P2687r0). In this context, a **jthread** is a container.

3.10. Profile: RAII

- ***Definition***: no resource leaks.
- ***Initial version***: Every resource that must be acquired and later released must be represented by a scoped object.
- ***Observation***: A resource owned by a static object lives forever unless explicitly released. Examples of resources: memory, locks, shaders.

3.11 Profile: Union

- ***Definition***: Every field of a union is used only as set
- ***Initial version***: Use a run-time checked variant (e.g., **variant**). Leave other uses to unverified code.
- ***Observation***: We need pattern matching. Then, the rule would be "don't use **union**s, use **match**."
- ***Observation***: there are important uses of unions that don't involve a tag. For example, versions of the short-string optimizations that uses the length of the string to select the implementation.

4. Flow analysis

The hardest practical problem with early Profiles implementations is likely to be the guarantees that require control flow analysis; e.g., see P3038R0 §8 ("dangling pointers"). Our implementations don't (all?) have general and fast static analysis that we can use. Consider

```
int* f(int x, int* p)
{
        // …
        if (x==0) {
                p = new int{9};
                // …
        }
        // … use *p …
        return p;
}
```

That's somewhat obscure code, but we have all seen worse. It may even be correct, but that assignment to **p** depends on the value of **x** and there is no guarantee that **p** isn't the **nullptr** upon entry to **f()**. In general, such problems could be buried deep in code. Two years ago, I found a logical equivalent, but much larger, example by applying Core Guidelines rules. I suggest (P3038R0 §8) that in an initial version problems requiring flow analysis be flagged except for simple linear code.

Instead, alternatives with safety implications could (in initial implementations) be restricted to a subset of C++ that a compiler can handle relatively easily and reasonably fast. For example:

```
int* f(int x, int* p)
{
        int* res = (x==0) ? new int{9} : p;
        // … use *p …
        return res;
}
```

The suggested pointer profile would ensure that **p** was checked for nullptr before use.

5. Next steps

To make the profiles approach credible to a wider audience, we need some of these profiles precisely specified, implemented, and tested. I suggest starting with:

- Settle the syntax for profiles (P3038R0).
- Settle on a syntax for indicating run-time violations (§2).
- Specify Initialization (§3.7) and decide what's feasible in the very short term
- Specify Ranges (§3.4) and decide what's feasible in the very short term
- Specify Casting (§3.8) and decide what's feasible in the very short term
- Devise a subset of Pointers (§3.5) that is feasible in the very short term (nullptr checking)

Separately, we need to work on the more complete validation issues:

- Document a detailed design for range-checked algorithms for Algorithms (§3.6)

- Document a detailed design for Invalidation (§3.6)

Crucially, once we have a syntax (and maybe even before), the work on the other suggested priorities can proceed in parallel. Obviously, I'd like to see work on all Profiles and on all aspects of Profiles. People will work on what interests them and are priorities for them and their organizations. The list above is just my suggested priorities.

I suggest some urgency. We cannot wait for a couple of years while polishing standards text and we need feedback from initial implementations and tests.

6. References

- B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. P2687R0. 2022-10-15.
- B. Stroustrup: Concrete suggestions for initial Profiles. P3038R0. 2023-12-16.
- B. Stroustrup, G. Dos Reis: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. P2816R0. 2023-02-16.

Further references can be found in those papers.

7. Acknowledgements

I received more constructive comments on the topic of Profiles than I could easily keep track of. Thanks to all who contributed to this note, notably Ilya Burylov, J-Daniel Garcia, Greg Law, Christof Meerwald , Gabriel Dos Reis, Jonathan Wakely, Andreas Weis , J.C. van Winkel, and Mike Wong.

Jonathan Wakely and Ville Voutilainen provided useful input to the range appendix.

I apologize for the length of this note. It was intended to be just a couple of pages outlining key ideas, but to clarify various points it grew to a full-blown paper.

Appendix: Algorithms

The description of Profiles in this paper is meant to be an overview, rather than a set of detailed specifications. I expect most individual Profile detailed specifications to be larger than this paper, involving a combination of static analysis, supporting library components, enforced coding guidelines, and possibly additional support tools. This appendix points to a couple of issues that need to be addressed for the Algorithms profile. The concrete examples are not standard-library quality, but examples to illustrate general ideas.

Rationale

To remain useful in programs requiring range safety guarantees, STL algorithms need to be range checked. Without that, people insisting on guarantees will have to abandon the STL. Range checking for STL algorithms is most easily achieved by using ranges explicitly.

Algorithms with explicit ranges

We can imagine compilers checking some examples of algorithms relying on the conventional pair-of-iterators style (e.g., **sort(p,q)** for iterators **p** and **q**) but such checking would be complex and lead to false positives. For example:

```
sort(v1.begin(),v2.end());          // almost certainly an error
sort(v1.end(),v1.begin());          // almost certainly an error
sort(v1.begin(),v1.begin()+n);      // Range error if v1.size() < n
sort(v.1.begin(),v1.end());         // OK
```

The first three examples could be caught by a static analyzer, but the third example would likely yield many false positives.

The Algorithms profile better insist on explicit use of ranges, e.g. **sort(v)**. Every STL algorithm should have a version that can be used exclusively with ranges, e.g., **copy(s,t**) for ranges **s** and **t**.

Ideally, we should write simple and safe code like this:

```
void sort(forward_range auto& r)
{
        vector v(r);        //  initialize v from r; deduce v's element type from r's
        sort(v);
        copy(v,r);          // copy into a range (potentially range checked)
}
```

But as we all know, that won't work. Without language changes relative to C++23, we can get close:

```
void sort(forward_range auto& r)
{
        vector v {from_range_t, r};       //  initialize v from r; deduce v's element type
        ranges::sort(v);
        copy(v,Output_range(r));          // copy into r (range checked)
}
```

I consider the need for **from_range**, **ranges::** , and **Output_range** ugly, distracting, a barrier to acceptance, and probably redundant, but that seems to be a minority opinion and it is not essential for ensuring range safety.

A checked output range class

The **Output_range** is a class from my personal code:

```
template<ranges::range R>
class Output_range {
public:
        using value_type = ranges::range_value_t<R>;
        using difference_type = int;

        Output_range(R r) : p{ r.begin() }, e{ r.end() }{}

        Output_range& operator++() { check_end(); ++p; return *this; }
        Output_range operator++(int) { check_end(); auto t{ *this }; ++p; return t; }

        value_type& operator*() const { check_end();  return *p; }
```

```
        bool operator==(const Output_range& a) const { return p == a.p; }
        bool operator!=(const Output_range& a) const { return p != a.p; }
private:
        void check_end() const { if (p == e) throw Overflow{}; }
        ranges::iterator_t<R> e;
        ranges::iterator_t<R> p;
};
```

It illustrates an idea, rather than being a concrete proposal. Note that **Output_range** is an STL iterator.

Implicit range checking

The Ranges profile requires that subscripting of **vector**, **span**, and the like using **[]** is implicitly range checked. So should ranges used by the Algorithms profile be. So **copy(s,t)** for a range **s** and an iterator or range **t** should mean **copy(s,Output_range(t))** using range-checking class **Output_range**. For initial experiments, explicit use of checking range classes would be obvious.

For many algorithms, run-time checking of individual accesses could be costly, possibly leading to the use of unverified implementation code. Wherever possible, that must be avoided:

- For output ranges, only fixed-sized targets need to be checked.
- For random-access ranges with a fixed range, it should be possible to avoid checking of many accesses after the initial check of the range. However, for many algorithms the heavy use of subscripting or iterator arithmetic, that's not going to be easy. In some cases, implementing using **span** will be a solution.

One-past-the end return values

Many algorithms return an iterator to an individual element or one-past-the-end. To be dereferenced, it must first be checked to see that it is not one-past-the-end. For each container, one-past-the-end is the logical equivalent to the **nullptr**. For **nullptr** we can easily check (**p!=nullptr**) or we can use **not_null(p)** or an equivalent. However, a returned iterator doesn't have a single, simple, and conventional way of checking. To simplify static analysis (and coding), we need one. For each container, **C**, we can define **bool C::not_at_end(C::iterator)** and/or **bool not_at_end(C&,C::iterator)**. This can be generalized to every range.