Doc. No. P3038R0
Date: 2023-12-16
Audience: SG23
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

Concrete suggestions for initial Profiles

Bjarne Stroustrup

Abstract

This is a set of suggestions for making the idea of “profiles” more concrete. It stops short of being a standards proposal. For that, it needs more work.

The general idea is to allow people to specify in code a set of guarantees to be provided. If you violate a profile, your code will not run. Profiles offer guarantees, not optional conformance. That’s a key difference between profiles and guidelines.

There must be several profiles because needs differ.

My personal focus is the **type_safety** profile that guarantees that every object is used (only) according to its definition. Type safety implies what people usually call memory safety and resource safety. Lesser guarantees can be easier to implement, yet still be immensely valuable to many, e.g., the **ranges** profile. Some safety requirements, such as the **arithmetic** profile, need guarantees beyond the type safety offered by C++ types.

The result of a profile is ISO standard C++ code; there are no changes in semantics, though typically the meaning of UB must be restricted (as is allowed by the definition of UB). If no profile is requested, ISO C++ – warts and all – is available unchanged.

1. Overview

This paper elaborates on the ideas presented on P2687R0 and my CppCon Keynote. There is nothing dramatically new here. It is still a design and a set of suggestions. It needs work on many practical details and implementation. That’s not a task for a single (part time) person.

Relatively new: discussion of

- Global properties (that can’t be guaranteed in a single translation unit)
- Response to run-time violations
- “lite” versions of profiles (that can be implemented simply in existing tool chains)
- UB is briefly addressed

Unless otherwise stated, I’m talking about the **type_safety** profile.

The rules suggested are inspired by the Core Guidelines project where they have been used without enforcement or with limited enforcement for years and at scale. For profiles, I suggest 100% enforceable rules, preferably compile-time enforceable rules without too high a burden for the compilers.

2. Global properties

A profile is enforced for a given section of code, but some rules we'd like enforced for a complete program. That's what compiler and build options are for. However, to provide a guarantee, the property must be stated in the code somewhere. It is then the compiler's job to communicate such a request to the linker and/or build system. For example, implementation of the **ranges** profile most likely includes the consistent use of range-checked **vector**, **string**, **array**, **span**, and **string_view** libraries.

To allow gradual adoption and composition of programs out of components with different requirements, not all properties can be global.

3. "lite" versions of profiles

Providing complete and fully general guarantees with minimal impact on programming style can be difficult in existing tool chains. It makes sense to implement incomplete solutions that catch the most common problems, such as lack of range checking (§10) and invalidation (§9). The suggestion here is to offer complete guarantees at the cost of requiring more restrictions on the code we can write under a profile than we would ideally want. Later, we can relax the restrictions on code as long as the guarantees are preserved.

Maybe "lite" is not the best name for this. Alternatives include "initial", "limited", and "experimental." It is important, though, that our initial versions do not appropriate the names of the ideal final versions.

4. Syntax – In-code directives

We need a way of requesting guarantees in code. Compiler and build option are hard to define portably, and hard to consistently enforce. Also, when reading a piece of code, we need to be able to know what guarantees it requires and/or offers. That holds for both human readers and compilers.

I use the **[[…]]** syntax because it stands out nicely in code and is implemented. I suggest:

- **[[enforce(P)]]** a module or a scope must enforce **P**
    - E.g., **namespace N [[enforce(p)]] { … }**
    - E.g., **export module DataType.Array [[enforce(memory_safety)]];**
- **[[enable(P)]]** Enforce **P** for uses of an imported module
    - E.g., **import M [[enable(P)]];**
- **[[suppress(P)]]** for a module or a scope
    - E.g., to use unverified/trusted code to implement abstractions needed for **P**.
    - E.g., **import old [[suppress(type_safety)]];**

- **[[profile(P) = enforce(P1,P2)]]**

Where **P**, **P1**, and **P2** are names of standard profiles or of strings naming profiles. For example, **[[provide(std::ranges)]]** and **[[provide("my-very-own-profile")]]**.

Clearly this specification needs work. In particular, it might be a good idea to use the already standard annotation syntax at least initially.

Using **suppress** to limit verification is more precise and more flexible than just a binary trusted vs. untrusted choice. This is important.

These directives are local to code in a single module or scope (i.e., enforced locally). Their names must be global.

Initially, we could specify enforcement for modules only and suppression for scopes only.

5. Initialization

I suggest the simplest and cheapest solution to the problem of missing initialization:

- *Unless marked **[[uninitialized]],** every object must be initialized explicitly or have a default constructor.*

There are clever (and safe) ways to ensure that an object is not read before written to, but they require compiler work (flow analysis and turning assignments into initializations) and aren't obvious to users.

Implicit initialization (e.g., zero-initialization of all objects that are not otherwise initialized) incurs a cost as we may need to reassign an object to its correct value after zero-initialization. Also, relying on zero-initialization when zero isn't a good initial value is a known source of errors.

Instead of the **[[uninitialized]]** annotation we might prefer a "value" to be used like an initializer for suppressing" uninitialized" compile-time errors, e.g., **char v[100'000] = uninitialized;** Either way, some annotation to suppress initializing large buffers is needed in several application areas.

6. Built-in pointers

The suggested rules for built-in pointers are simple and very restrictive:

- *A built-in pointer points to a single element or is the **nullptr**.*
- *Unless marked **owner**, a built-in pointer is not an owner.*
- *Don't access through a built-in pointer without checking for **nullptr***

An **owner** is an pointer responsible for destroying an object pointed to (§7).

The first rule eliminates pointer subscripting: use **vector**, **span**, and the like where subscripting can be run-time checked.

The second rule eliminates **new** and **delete** in general code: use **vector**, **make_unique**, and the like for which RAII can make sure that **new**s and **delete**s match.

The third rule is most easily enforced by insisting on use of **not_null**. Consider:

```
void g(int*  pp)                // traditional interface
{
        If (!p) error("nullptr");
        *p = 7;
}
```

Here, it is not explicit whether **nullptr** is a valid argument or if the test is necessary. Maybe the test is there just in case users didn't read the manual. However, we can be explicit:

```
void f(not_null<int*> p)        // checked interface
{
        *p = 7;
}

void g(int*  pp)                // traditional interface
{
        not_null<int*> p = pp;
        *p = 7;
}
```

It would be relatively simple to accept **nullptr** checks immediately before use. Accepting checks distant from use, would in general require flow analysis.

The reason the pointers are not simply banned is that they are extremely common, very useful, and their use can be restricted to type-safe uses. Built-in pointers are extensively used in interfaces so it would be infeasible to simply replace them with "smart" pointers.

Nested calls with a **null_ptr** argument could be optimized to incur the cost of a check only at the initial call.

7. Ownership

An owner is an object responsible for destroying an object pointed to. See P2687 "§5.2 Ownership." Prefer to use ownership abstractions, such as **vector** and **unique_ptr**. Such use leads to simple, efficient, and correct code. In addition, we need a low-level mechanism to handle interactions with code presented by C-style pointer interfaces. For example:

```
void f(int* p1, owner<int*> p2)
{
        delete p1;      // error: delete of built-in pointer
        delete p2;      // it would be an error not to delete p2
}

void g(int* p1, owner<int*> p2)
{
        p1=p2;  // OK: p1 is not an owner but points to the same object as p2
        p2=p1;  // error: p2 was not deleted before being assigned to
}
```

The basic rules are:

- A non-**owner** cannot be **delete**d.
- An **owner** must be **delete**d or passed to another **owner**.
- An **owner** can be copied to a non-**owner**, but that copy is not an **owner**.
- The result of a non-placement **new** is an **owner**.

Clearly using a **unique_ptr** would be easier.

These rules cannot be enforced in general code and could require complex flow analysis even in tractable cases. One approach is to use of **owner** for a restricted subset of C++ as described in §8. From the code-generation point of view, **owner** is just and alias. This is important to maintain ABI stability. However, **owner** must be known to the compiler (and/or other static analysis tools) for the rules of ownership to be enforced; it cannot be just an ordinary alias.

Another use of **owner** would be in the implementation of ownership abstraction, minimizing the need for suppressing profiles to be able to implement ownership abstractions.

Use of annotations does not scale; **owner** is a low-level mechanism to handle cases where "proper ownership abstractions" cannot be used.

One way of avoiding flow analysis would be to offer an **owner** class along these lines:

```
template<class P>
concept Ptr_to_class = is_class_v<remove_pointer<P>>;

template<class P> class owner {
public:
        owner(P pp)  = delete;
        owner(owner& pp) : p{ pp.p } { pp.p = nullptr;  } // transfer

        void operator=(P) = delete;
        void operator=(owner& pp) { delete p; p = pp.p; pp.p = nullptr; }  // transfer

        ~owner() { delete p; }

        operator P () { return p; }
        auto operator*() { if (p == nullptr) error();  return *p; }
        auto operator->() requires Ptr_to_class<P> { if (p == nullptr) error();  return p;  }
private:
        P p;
};
```

Note the deleted constructor. We would still have to ensure that the result of a non-placement **new** became an owner where a profile required it.

8. Dangling pointers

The rule for preventing dangling pointers is simple, though unenforceable in arbitrary code:

- *a pointer points to an object or is the **nullptr**.*

Here, “object” means an object that has been initialized and not destroyed. In this context, I think of every type as having a destructor, so this rule applies to all types, incl. built-in types.

Here, “pointer” means anything that directly refers to an object, such as a pointer, a reference, a lambda capture, smart pointer, or an iterator.

For example:

```
void f(X* p)
{
        auto p2 = new X{};
        // …
        delete p2;      // OK: q is an owner
        delete p;       // error: delete non-owner
}
```

This is dealt with (statically) by prohibiting **delete** of raw pointers that has not been declared **owner** (see §7). Better:

```
void f2(owner<X*> p)
{
        auto p2 = new X{};
        auto p3 = make_unique<X>();
        // …
} // errors: owner p not deleted and p2 not deleted
```

Obviously, most examples are not that simple. Consider:

```
int* glob = nullptr;

void g(int* p)
{
        glob = p;               // error: trying to store away pointer to unknown lifetime
}
```

Here, **glob=p** is trying to store a pointer to an object of unknown lifetime in an enclosing scope. That could give rise to a dangling pointer and is thus banned. That rule is conservative, but relatively simple and can be enforced statically.

```
void f(int* p)
{
        int x = 4;
        g(&x);          // OK
        g(p);           // OK
}
```

These calls are OK. It’s only **g()**’s attempt to store away the pointer that is bad

The rule is that You can pass a pointer out of a function only if it cannot point to an object that goes out of scope upon function exit:

```
int* f2(int* p)
{
        int x = 4;
        return &x;              // error: return pointer to local
        return p;               // OK: p was valid upon entry and we didn't invalidate it
        return new int{7};      // OK, but will lead to a leak
        return &glob;           // OK pointer to static object
        throw p;                // error: could pass p out of the scope of *p
}
```

We received **p** as an argument, so we can pass it to a called function.

Returning **new int{7}** is prevented by the ownership rule (§6 and §7), rather than the scope rule.

Many functions, including standard-library functions – take arguments of pointer types (e.g., iterators), so it is essential to allow pointers to be passed as function arguments (and into nested scopes). For example:

```
void g(span<int> s);

void  f()
{
        array<int,1024> buf;
        f(buf);                 // passes span<int,1024>{buf}
}
```

We can build and use containers of pointers but only return them if the pointers point to objects that will not go out of scope upon return:

```
void g(vector<int*>& v);

vector<int*> f(int* p)
{
        int x = 7;
        vector<int*> v = { &x, p, new int{7} };    // OK
        g(v);           // OK
        return v;       // error: v contains element &x that will go out of scope
}
```

The real problem when enforcing the fundamental rule is that determining which pointer is returned is in general control-flow dependent:

```
Int* f2(int* p, int v)
{
        int x = 4;
        // …
        return (v==7) ? &x : p;   // potential error
}
```

For arbitrary control structures (in the … part of the code) that **return** statement cannot be proven safe and must be rejected. The problem is that specifying a level of flow analysis for every compiler to follow is hard. We cannot have that level implementation defined because that would damage portability of code using that profile.

One possible answer (for now) would be to say "We can return a pointer that"

1. Was passed in as an argument.
2. Is initialized to point to a static object.
3. Is initialized to point to an object created by **new** (and not **deleted**).
4. Is initialized to a member of something pointed to by 1, 2, or 3.

Here, I consider returning a value from a function a form of initialization (of the return value), so this is an example of 4:

```
int* f(span<int&> s, vector<int*>& v)
{
        return v[3];       // OK
        return s[3];       // OK
        auto p = v[2];     // OK
        auto q = s[4];     // OK
        return (v[2]<s[4]) ? p : q;          // OK

}
```

This rule is very restrictive but requires no flow-analysis, just a need to keep track of which pointer variables have been initialized to a valid pointer. Questions with my suggested answers in parentheses:

- Any logical problems with this? (no)
- Any fundamental implementation problems with this? (no)
- Is this sufficient to cope with a large class of real-world problems? (yes)
- Can this be relaxed to gain significant benefits without placing a major burden on implementers? (I don't know)

My guess is that the answer will eventually be: Yes, we can gain important benefits from a relaxation, but it will be a significant burden to some implementers, and it will be hard to define exactly how clever an analyzer must be. Thus, I suggest we postpone such relaxation.

Note that passing arguments by-value and returning by-value isn't a problem unless those objects have pointer members, in which case they count as pointers (P2687 "§5.3.3. Multiple pointers"):

- Classes with pointer members
- Lambdas (they are classes, and remember capture-by-reference)
- **Jthread**s (they are classes in a defined scope)
- **unique_ptr**s and **shared_ptr**s (they are classes with a single logical pointer member)

- Pointers to pointers
- References to pointers
- Arrays of pointers

9. Invalidation

There is little I can add to the discussion in P2687 "§5.3.4". A container is invalidated when a pointer in its implementation points to an element that may have reallocated or **delete**d its elements. For example

```
void f(vector<int>& vi)
{
        vi.push_back(9);        // may relocate vi's elements
}

void g()
{
        vector<int> vi { 1,2 };
        auto p = vi.begin();    // point to first element of vi
        f(vi);
        *p = 7;                 // likely disaster (but we won't get here)
}
```

For a static safety guarantee, we need to reject **return**s that can lead to a violation. Thus, this **f()** must be considered invalidating and **g()** must be rejected.

By default, all functions that take a container by non-**const** reference are considered invalidating and all functions that take a container by **const** reference non-invalidating. This is simple and implemented in the VS static analyzer. We need a **[[not_invalidating]]** to annotate functions that potentially could invalidate, but don't (or we get a lot of false positives). A **[[not_invalidating]]** annotation can (and must) be verified when compiling the definition of a supposedly **[[not_invalidating]]** function. Thus, a faulty **[[not_invalidating]]** annotation will be a be an error rather than a potential loophole in the rules.

10. Range checking

Range checking a traditional pointer is in general impossible, so we ban subscripting of pointers. Instead, use **vector**, **span**, **string**, **string_view**, and the like and range check their subscript operators.

Pointer arithmetic is also banned, so the implementation of checked subscript operators must in general be done in unverified code.

Range checking, **ranges**, should be a global property (§2).

11. STL

Here, I use "STL" to refer to the standard library's iterator, ranges, containers, and algorithm framework. Iterators are kinds of pointers and thus require checking. I suggest:

- *Use range versions of algorithms.*
- *Provide versions of algorithms that use a range-checked output iterator.*

- *Use **at_end()** to check iterators that might be one-beyond a range*

See P2687, §6.3. Iterators and Ranges.

The general implementation of this can be difficult but it is open to a "lite" version that is easily added to a compiler and can catch most violations. For example:

```
void test(vector<string>& v1, vector<string>& v2)
{
        copy(v1,v2);
}
```

By taking the concepts of **copy()** into account, we can use **an input_range** for **v1** and an **output_range** for **v2**.

This simplifies the use of the STL algorithms while making them significantly safer. In this direction lies complete type safety because we preserve sufficient information to do complete checking even in the presence of iterators.

12. Other profiles

We need some standard profiles, notably:

- **type_safety** (no type-or-resource violations)
- **ranges** (no pointer arithmetic; **span** and **vector** range **throw** or terminate on violations)
- **arithmetic** (no overflow, no narrowing conversions, no implicit signed/unsigned conversions)

For **arithmetic**, I see no alternative but to ban implicit signed-unsigned conversions and the wraparound allowed by the standard's modular arithmetic. These are significant bug sources.

For a longer list, see P2687 or my CppCon keynote. Obviously desirable examples include

- **concurrency** (absence of data races and race conditions, and more)
- **hard_real_time** (no free store allocation after startup, and more, such as requiring some specific library support)

However, those are probably best left out of the initial set because they need more thought and experimentation. First, let's get the framework in place so that people can work on their favorite topic without being overly dependent on the work of others.

The set of profiles must be open because not all guarantees desirable to some community can be provided as standard profiles.

For previous work, see the Core Guidelines profiles (In.force: Enforcement).

13. Response to violation

When violating a profile rule that can be checked at compile time, we naturally get a compile-time error. However, not all rules can be caught at compile time. Notably range errors and **nullptr** dereference. Different people have different needs and different opinions on how to respond to a violation. The multi-year discussion about the similar problem in the context of contracts show have contentious this can be.

We need at least to be able to choose between termination and throwing an exception: some systems cannot accept termination and some systems don't use exceptions. Other potential violation responses include "log and terminate" and "log and continue."

The selection of violation response must be a global property and will have to be known to the implementors of standard library components, such as **vector** and **span**. I suggest we have the option to set that in code. For example

[[enforce(range) runtime-violation-response = throw]]

The build system must ensure that all requests for such a violation response are identical.

When throwing, we might want to specify which exception to throw. For builds that neither throw nor terminate, a function to be called could be specified.

14. Undefined behavior

Undefined behavior (UB) is a difficult and often misunderstood phenomenon. I will not go into details here. UB is being re-examined in the committee (SG12). For the **type_safety** profile the only UB that absolutely must be eliminated is the so-called "time-travel optimization" where an occurrence of UB is used to eliminate a test on the path leading to it. The range checking and pointer dereference checking turns UB into a well-defined response (§13). Undefined just means that the standard doesn't define the meaning, so giving a well-defined meaning is among the valid alternatives.

15. References

- B. Stroustrup and G. Dos Reis: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. P2816. The slides from my SG23 and EWG presentations February 2023.
- B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++ . P2687R0. Last year's discussion of profiles.
- B. Stroustrup: Delivering Safe C++. My CppCon'23 keynote putting profiles and "safety" into a historical context and introducing "profiles" for people who don't already know what they are supposed to be.
- B. Stroustrup and H. Sutter (editors): C++ Core Guidelines .