Shares
Doc. No. P4222R2
Date: 2026-07-11
Audience: EWG, SG12, SG20, SG23
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

An initialization profile (R2)

Bjarne Stroustrup (www.stroustrup.com)
Columbia University

Abstract

A design of the initialization profile was looked at by the EWG [LLG25] and the profiles are now processed by SG23. This note emphasizes the rationale of the initialization profile, considers a few alternatives, and suggests simplifications. This revision of the paper is primarily meant as input to the implementation efforts and for people looking at the profile's likely effect on code bases. Therefore, changes to the design are to be expected. With the exception of the overloading described in §4.8 and some notational details to match the latest specification, an implementation exists.

The key features proposed are

- Every object is initialized or noted by the compiler not to be.
  - An object that is uninitialized cannot be read or written to.
- The profile is completely compile-time enforced.
  - Language use is limited to make that possible and affordable.
  - When those limits are unmanageable, the profile can be suppressed or **now_init()** can be used to simplify unverifiable complex initialization techniques.
- Every object that is implicitly initialized under current rules is initialized (as ever).
- An object intended to be uninitialized can be marked **[[uninit]]**.
  - An uninitialized object can be explicitly initialized (e.g., using **construct_at()**).
  - An initialized object cannot be marked **[[uninit]]**.
- A pointer, reference, or "smart pointer" can be marked **[[ref_to_uninit]]**.
  - An object referred to by an **[[ref_to_uninit]]** is considered **[[uninit]]**.
- A argument can be marked **[[must_init]]** to indicate that it initializes a **[[ref_to_uninit]]**.
- Static objects must be compile-time initialized.
- Code compiled without enforcing this profile but doesn't violate its rules have exactly the same meaning as a compilation that enforces the profile.

This paper presents the beginnings of the library techniques needed to get the **[[uninit]]** and **[[ref_to_uninit]]** out of most application code, make code using the initialization profile simpler than code not using it, and minimize the need to suppress the initialization profile to express low-level code.

Changes since R0

- **[[uninitialized]]** has been shortened to **[[uninit]]** (§4.1).
- The rationale for using attributes rather than modifying the type system has been elaborated (§4.1).
- The use of **void*** has been explained (§4.3).
- The interaction between **[[ref_to_uninit]]** and templates has been elaborated (§4.4).
- The beginnings of the library techniques needed for simpler safe code is presented (§4.4, §4.5).
- The use of **construct_at()** and **destroy_at()** are discussed (§4.5).
- The function **now_init()** is introduced to limit the need for suppression of the profile (§4.2, §4.4).
- The attribute **[[must_init]]** is introduced to ease the use of initialization functions.
- The initialization of classes without constructors has been elaborated (§5.3).
- Definite assignment is discussed (and partly rejected) (§1.4).
- The drafting of wording had been extended (though not completed) (§9).
- The choice of notation is discussed (Appendix).

1. Introduction

The initialization profile should be the easiest to define, but there can be no profile that everybody can agree on without discussion and alternative choices. Also, the rules for initialization and uninitialized memory are far more complex than most people are willing to believe. The initialization profile is foundational to just about every profile, so the initialization profile must isolate those unmanageable complexities, leaving "ordinary code" simple. Here is a definition with supporting discussion of alternatives and suggested resolutions.

The basic design requires:

- Every object is initialized or marked as uninitialized at its point of initialization.
- Reading or writing uninitialized memory is an error.
- No complex flow analysis required.
- No run-time tests are needed to implement this profile.
- A class must ensure that all accessible data members are initialized before their use.
- Implicit initialization (through a default constructor or a language rule) is initialization.

- There is a way to pass uninitialized memory out of a scope (§4.3).
- Initializing an object twice is an error.
- Destruction makes an object uninitialized.
- Destroying an object twice is an error (the second attempt to destroy is an attempt to access to raw memory).
- The design is simple enough for most developers to use without understanding tricky concepts or implementation details.
- Most existing good code will work unmodified.
- There are ways to express code that's too complex for the profile to handle.
- The initialization attributes do not modify the type system; they simplify enforcing the profile's rules.
- A program that doesn't violate the initialization profile runs exactly the same if the profile is enforced or not.

Use of profiles is optional. If some code cannot reasonably be expressed using the initialization profile, don't write such code, don't use the profile, or suppress the profile while using such code. Remember that most language rules and profile guarantees assume that no uninitialized object is read from or (if a class with an assignment operator) written to. This profile enforces that.

Note that for arbitrarily complex code, the "every object is initialized" guarantee cannot be statically guaranteed, so notation must be provided to carry sufficient information for every potential use of an uninitialized object to be statically detectable.

The specification of a profile must focus on the guarantee provided ("No use of uninitialized objects") rather than being a long list of actions needed to achieve that (§8). That guarantee simplifies understanding as well as use.

The simplest way to conform is simply to initialize all objects. Had it not been for the need to manipulate uninitialized memory (§1.1), this paper would have been just three pages rather than 30. The most complicated parts of this paper relate to rejected alternatives. The proposed mechanisms are actually quite simple: just three attributes and one trivial template function.

This profile is designed to fit with the proposed profiles framework [GDR25].

1.1. Buffers and memory pools

One of the ways C++ differs from many other languages is in the extensive use of performance-critical memory buffers and user-defined memory pools. For example, when filling a memory buffer under severe performance constraints, we can't first initialize it with default values and later add the desired values without adding noticeable overhead or rely on specialized hardware support (that is not available everywhere). For example, **std::vector** must deal with a mixture of initialized and uninitialized memory (usually using **construct_at()** and **destroy_at()**).

It is common to have an allocator provide an uninitialized area of memory to be managed by some other class or function. Ideally, we would like a single abstraction to handle this, but we don't have one and I suspect the range of applications of this idea/need is too great for us to get one any year now. I'd love to been proven wrong at that, but I will progress on the assumption that I'm not. What we have is **void*** ("pointer to memory of unknow type"); §4.3.

The simplest solution would be to ban passing objects with directly accessible uninitialized memory. That is, require that such data be private data members so that the obligation to properly initialize is placed is isolated in a class. However, that would ban passing pointers and **span**s to **struct**s and arrays with uninitialized members, thus forcing the initializing profile to be suppressed for much existing critical code.

Alternatively, users would have to suppress the initialization profile without in-code hints of why that was needed. However, for some applications, just relying on suppression would not be manageable and would compromise the initialization profile. For example, large uninitialized buffers are common for good reasons. That use case is in fact the reason why local variables were left uninitialized in C (I once asked Dennis). Consequently, we need a mechanism to indicate that a pointer or "smart pointer" passed to or from a function refers to an object containing uninitialized objects (e.g., an array from an allocator).

1.2. Uninitialized memory

There are many places in code where we sometimes leave an area of memory uninitialized with the aim of possibly later turning it into a properly initialized object:

- A local variable.
- A member of a class.
- An area on the free store of a type without default initialization allocated by **new**.
- An area on the free store allocated by a non-initializing allocator such as **malloc()**.

Often, such objects are uninitialized for good reasons, but we might leave a variable uninitialized by mistake so that it later becomes the cause of errors. The aim of the initialization profile is

- To ban use of uninitialized memory except when explicitly requested.
- To use that ban to support and simplify use of reasonable initialization techniques.
- To make the use of uninitialized memory less error-prone.

Initialization (or explicit suppression of initialization) must be done at the point of definition and at every point where uninitialized memory is passed out of a scope must be explicitly marked. This is an application of the subset-of-superset strategy. Only after adding notation (and libraries) to simplify analysis can we subset to offer the guarantee. Not defining an object before we have a value with which to initialize it yields the simplest code. We should do that wherever possible.

We do not aim for the initialization profile to enable every possible technique. Techniques considered too error prone or too complex are relegated to code that suppresses the initialization profile. For example, using random access to selectively initialize elements of an array of uninitialized objects cannot be statically validated (§5.4) so it must be banned or relegated to code where the initialization profile is suppressed.

The initialization profile will be very widely used since its guarantees are relied on by most code and most profiles. Therefore, its overhead of enforcing it must be low at compile time and zero at run time.

C++26 ensures that accessing an uninitialized variable is no longer UB but just erroneous behavior. That's a significant improvement but not exactly what the initialization profile and other profiles that depend on it need. For starters, the response to erroneous behavior is implementation defined so it is nontrivial to know what the effect of hitting it is:

- Testing for erroneous behavior could be a run-time action. A programmer of portable code must be prepared to cope with whatever a possible violation handling is.
- If the resolution of erroneous behavior is to return a default value, the end result may be a logical error.
- If the resolution of erroneous behavior is termination (possible delayed termination), it will not be usable in applications that cannot tolerate unconditional termination.

The initialization profile offers a stronger guarantee at the cost of imposing stricter rules of use.

Note that essentially all the complexity of the initialization profile comes from offering support for handling uninitialized memory rather than just requiring every object to be initialized and relying on suppression of the profile to handle cases where uninitialized memory is needed.

Like all profiles, the initialization profile is opt-in. There will be code – typically code written in ancient styles – for which applying it will not be feasible.

For interaction with code compiled without the uninitialized profiles, see §7. For the simplest and safest code, opt into as many standard profiles as possible.

1.3. Static analysis: No complex flow analysis

Profiles in general and the initialization profile in particular critically rely on static analysis (ideally in the compiler) to ban unsafe uses. Much static analysis involves asking simple question about individual declarations and statements, such as "does this definition have an initializer?" However, some profiles require some form of flow analysis, such as "is this uninitialized object constructed before use?"

The answer to that last question is critical for the initialization profile's ability to deal with buffers of uninitialized memory (e.g., as used in the implementation of **std::vector**) and member variables initialized in a constructor body rather than in a member initializer.

To make static analysis affordable in general, consider all branches of a run-time conditional statement executed for the purpose of profile guarantees. The alternative would require unaffordable static analysis checking (global static analysis and/or symbolic execution) or added run-time. Also, the profiles I currently consider, including the initialization profile, are designed to require only local static analysis.

Random access to an uninitialized array must be banned because allowing it would make the initialize-before-use guarantee impossible to enforce statically (see §5.4 for an example).

Errors related to misused after initialization (e.g., access after deletion [§4.6]) must be handled by other profiles.

1.4. Definite assignment

There are languages (e.g., Ada and C#) that simply enforce that a variable is assigned to before use. This is often called "definite assignment." Others (e.g., Java), default-initialize every object. However, in C++

- Initialization and assignment can be very different.
- An address of some memory can be passed out of a scope to be initialized elsewhere (e.g., in a separate TU).
- If allowed, users can be confused by delayed initialization, especially when maintaining code.
- Code must have the same meaning compiled with and without the initialization profile.
- Some constructs (e.g., loops and conditionals) make compile-time determination of whether an object has been initialized impossible.
- Requiring perfection even when theoretically possible would put a serious burden on implementers.
- If a compiler can figure out to delay initialization without changing semantics, it can do so as an optimization.

For examples of how some initializations can be delayed, see §5.4.

Also, I am not proposing to change the semantics of correct C++.

Consider a simple example of the complexity we would face if initialization could be implicitly deferred and disguised as assignment:

```
void f(int v)
{
        X x;    // maybe uninitialized
        // …
        x = v;  // maybe an initialization
}
```

Depending on what **X** is, this has different meanings:

- If **X** is **std::byte**, **x** is uninitialized, but the assignment is valid.
- If **X** is **int**, **x** is uninitialized, but depending on the type of **v**, the assignment can be implementations defined or UB in older compilers and erroneous behavior in C++26.
- If **X** is **std::string**, **x** is default constructed and the assignment is well defined, but unless the default assignment is optimized away, the generated code differs from a simple initialization.
- If **X** is a class without a default constructor, **x** is uninitialized, but the assignment is UB in older compilers and erroneous behavior in C++26.
- If **X** is a class where the meaning of initialization and assignment differs, code compiled with and without the initialization differs.

Imagine writing generic code with such a definite-assignment extension to C++. If the extension was implemented only for the initialization profile, we'd get code that would have different meaning compiled with and without the profile.

2. Implicit initialization is initialization

A definition that invokes a default constructor is considered initialized (see §5.1).

A static variable with a default initializer is considered initialized (see also §3).

A dynamically created object (using **new**) with a default initializer or a default constructor is considered initialized.

3. Static objects

The order of initialization of static objects in different translation units is implementation defined and can lead to an object being accessed before it is initialized. For example:

```
int f() { extern int y; return y; }        // f.c
int x  = f();

int g() { extern int x; return x; }        // g.c
int y = g();
```

Either order of initialization leads to an uninitialized variable being read. This must be avoided. Naturally, good developers avoid such examples, but not all examples are this trivial and we want initialization before use to be guaranteed. So

- Non-local static objects must be initialized at compile time or link time. No run-time initialization is allowed for such objects.

This rule seems Draconian, but it conforms to common practice and has language support.

There are simple ways to obey this rule:

- Don't use global variables.
- Use only **constinit** global variables.
- Define default constructors to be **consteval**.
- Wrap statics in functions that guarantee initialization before use.

Variations of all four techniques have been used for decades. Even **constinit** is just a recent direct support of an old technique.

An example of the last alternative is:

```
X& var() { static X v = init(); return v; }
```

The purpose of wrapping a static object in a function or a class is often to control the timing of its initialization.

If the initialization is too complex to be guaranteed by these simple techniques, the initialization profile must be suppressed for just that initialization.

4. We need a way to say "leave uninitialized"

We need a way to say "not initialized" because we often pass uninitialized memory around (e.g. to initialize it; see §1.1) and in rare cases initialization of class members must be postponed. That's unfortunate because most of the complexity of the initialization profile comes from those two kinds of use cases.

For most code, "just initialize all objects" is a good simple rule. Consider every introduction of an uninitialized object part of an optimization technique and treat it as a potential source of complexity and potential logic errors. The initialization profile makes lack of initialization visible in code and eliminates the errors it could have caused.

In C++26, we have **[[indeterminate]]**. However, this is not meant to be "misused to document intentional lack of initialization" [TK24], so I suggest something slightly different for that.

The most important decision is "should uninitialized be a property in the type system or not?"

I think not

- Tracing uninitialized memory through our programs is often too expensive in compile-time or run-time (it is often flow dependent and passes function barriers).

- Tracing uninitialized memory through our programs is often impossible to do statically.
- Some invoked code is C or C-style C++ (e.g., an operating system).
- Not all code can be compiled with a profile (for years, at least).

To ensure that every object is initialized before it is used we

- provide an "uninitialized" marker for definitions to suppress the compile-time error that the initialization profile would trigger if we didn't immediately initialize.
- Provide a "refers to uninitialized" marker to indicate that a pointer or a reference to an uninitialized object is created.

This allows us to enforce "every object is initialized before use" for a restricted subset of C++ (§1).

4.1. Notation

There are three obvious styles of syntax

- A "pseudo value", e.g., **X x = uninitialized;**
- An attribute: e.g., **X x [[uninit]];**
- Say nothing and let the compiler note when an object is uninitialized: e.g., **X x;**

The difference is purely syntactic. The attribute notation, **[[uninit]]**, clearly indicates that it is just information to optional analysis (e.g., by the compiler). It makes it possible to verify the initialization profile on a modern compiler and then compile the code on an older compiler that ignores those attributes. Also, it saves us from introducing a keyword that (hopefully) would be rarely used.

Using the **[[uninit]]** notation leaves one case uncovered – uninitialized memory in an initializer list:

```
tuple<int, int, int> t = {1, [[uninit]], 3};      // syntax error
```

as opposed to

```
tuple<int, int, int> t = {1, uninitialized, 3};          // "magic" non-value
```

I suggest that this case is sufficiently rare for us to leave it without specific coverage. The most general alternative is to suppress the initialize profile, rather than to introduce a special notation. However, for the more common cases, I consider suppression verbose, open to misuse, and likely to be underused in favor of workarounds. Using a default value is often acceptable. For example:

```
int uninitialized = 0B1010101010101010;
```

This leaves the alternative of simply letting the compiler "remember" that an object is uninitialized. This shares the advantages and disadvantages with the **[[uninit]]** solution, but that

- is implicit so doesn't indicate whether the lack of initialization is intentional or not.
- doesn't extend to handle cases where the information of whether an object is initialized or not needs to be passed between functions (§5.2).

Should **[[uninit]]** be considered part of the initialization profile and therefore spelled **[[profiles::uninit]]**? I think not. The information it conveys is generally useful, not profile dependent, and can/will be used by people who do not use the initialization profiles (e.g., to help with code reviews) and most likely also by other profiles.

Should **[[uninit]]** be spelled out as **[[uninitialized]]**? I think not. The latter is more verbose and invites an English/American spelling problem. I tried the longer version for a while and found it a source of misspelling and unpleasing verbosity.

Attributes are typically assumed to be optionally enforced. For a profile, "optionally enforced" means "enforced when the profile is opted into."

4.2. [[uninit]] rules

Rules:

- An uninitialized object not marked **[[uninit]]** is an error.
- An initialized object marked **[[uninit]]** is an error.
- An object marked **[[uninit]]** is left uninitialized and must be initialized before use (e.g., using **construct_at()**; §6). After initialization, the object is no longer **[[uninit]]**.
- The function template **now_init()** can be used to provide a pointer to something initialized from a pointer to something previously uninitialized (§4.4).
- A pointer or reference to an **[[uninit]]** can be passed only to a **[[ref_to_uninit]]** (§4.3).
- A pointer or reference to something initialized can't be passed to a **[[ref_to_uninit]]** (§4.3).

For example:

```
int glob;                       // OK: default initialized
int glob2 [[uninit]];           // error: initialized and [[uninit]]

void f()
{
        int loc;                        // error: no initialization
        Int loc2 [[uninit];             // OK
        Int loc3 = 3;                   // OK
        Int loc4 [[uninit]] = 4;        // error: initialized and [[uninit]]
        vector<int> v1;                 // OK: default initialized
        vector<int> v2 [[uninit]];      // error: initialized and [[uninit]]
        string s1;                      // OK: default initialized
        string s2 [[uninit]]            // error: initialized and [[uninit]]
```

```
        int loc5 [[uninit]];
        int x = loc5;     // error

        int loc6 [[uninit]];
        loc6 = 7;         // initializing
        int y = loc6;     // OK
}
```

We could imagine using **[[uninit]]** to suppress default initialization, but that would make code more error prone rather than less so.

4.3. Pointers and references

As ever, a reference cannot be uninitialized. The initialization profile requires the same for pointers. When feasible, instead of leaving pointers uninitialized, use initialization to **nullptr**.

We need a way to state that an area of memory pointed to is uninitialized. We can't use **[[uninit]]** because it is not the pointer, reference, or iterator that is uninitialized; it is the area of memory pointed to (if any) that is uninitialized.

From about 1983, we have **void*** to point to memory of unknown type. Unfortunately, for compatibility reasons, by default, we must assume a **void*** points for something initialized. For example:

```
int x1 = 7;
void* p1 = &x1;         // OK

int x2 [[uninit]];
void* p2 = &x2;
```

All **void***s require casting to access what they point to, but what is pointed to by **p1** and **p2** is fundamentally different from the perspective of initialization. However, the type system doesn't reflect that.

We introduce **[[ref_to_uninit]]**, "reference to uninitialized", to help correctly manipulate areas of uninitialized memory. Manipulating a mix of initialized and uninitialized memory is hard. It would be far simpler to demand that all elements of an array are initialized (or not), but that would leave major critical areas of C++ use unserved. We get

```
int x1 = 7;
void* p1 = &x1;                         // OK
void* p2 [[ref_to_uninit]] = &x1;       // error

int x2 [[uninit]];
```

```
void* p3 = &x2;                          // error
void* p4 [[ref_to_uninit]] = &x2;        // OK
```

To use what **p1** points to, we must cast (as ever, hoping we cast correctly or rely on the casting profile). For **p4**, we must still cast, but we get an **[[uninit]]** object.

Now consider allocators:

- **malloc()** returns a **void*** pointing to uninitialized memory
- **calloc()** returns a **void*** pointing to zero-initialized memory

The obvious solution would be to define:

```
void* malloc [[ref_to_uninit]] ( std::size_t size );
void* calloc( std::size_t num, std::size_t size );
```

To require that, we would have to explore if the addition of **[[ref_to_uninit]]** to all allocators is manageable. The problem is system headers that might not be modified for the benefit of C++ profiles. In that case, either functions like **malloc()** will have to be called only in code where the initialization profile is suppressed or such functions must be known to an analyzer enforcing the initialization profile. The latter will already be the case in some implementations because of erroneous behavior implementation.

Now consider the more detailed rules for combinations of **[[ref_to_uninit]]** and **void***: Already, a **void*** must be cast before any use.

- By default, a **void*** is considered to point to memory initialized to some type.
- A **void*** marked with **[[ref_to_uninit]]** must point to something **[[uninit]]**.
- If a **void*** is assigned to another **void*,** neither or both must be **[[ref_to_uninit]**].
- If a **[[ref_to_uninit]] void*** is cast to another pointer type, the result is considered **[[ref_to_uninit]]**.
- After memory referred to by **[[ref_to_uninit]]** is initialized (e.g., by **construct_at()**) it is no longer **[[uninit]]**.
- The result of dereferencing a **[[ref_to_uninit]]** is **[[uninit]]**.

The name **[[ref_to_uninit]]** will be considered ugly by some, but since it belongs in inherently complex foundational code, I consider that acceptable and conventional. In such code, and in code directly using such code, it will be relatively common, so a longer and more descriptive name would be considered verbose; for example, **[[points_or_refers_to_uninitialized]]** or the **[[ref_to_uninitialized]]**. The former is fully explicit but would set a new record in verbosity. The latter is a muddled mix of abbreviation and explicitness.

Consider:

```
void f1(int* p [[ref_to_uninit]]);      // *p must be uninitialized
void f2(int& r [[ref_to_uninit]]);      // r must refer to uninitialized

int* p0;                                // static, thus initialized

void g(int x)
{
        int* p1;                                // error: uninitialized pointer
        int* p2 = nullptr;                      // OK
        int* p3 [[ref_to_uninit]]= &x;          // x must be uninitialized

        f1(p1);                                 // error: p1 is initialized
        f1(p2);                                 //  OK: p2 is initialized

        int* p4 [[uninit]];                     // p4 is uninitialized
}
```

Why not simply make **[[uninit]]** be part of the type system like **const** is? Basically, if it isn't opt-in, it would break a lot of code (mostly old code and C libraries). When enforced, **[[uninit]]** is viral. I hope it will become universal, but that will take time and a gradual introduction is essential.

4.4. Library support

Many – possibly most – uses of uninitialized arrays use "slots" to store objects of a given type, rather than **void***s. To avoid wide replication of messy code implementing such, we should provide library support for it.

- The standard-library family of **ininitialized_*()** function's iterator arguments should be annotated by **[[ref_to_uninit]]**.
- The **construct_at()** should take a **[[ref_to_uninit]]** and return a pointer to an initialized object.
- The object subjected to **destroy_at()** should be considered uninitialized.
- We need a **now_init()** that takes a **[[ref_to_uninit]]** pointer and returns a pointer to initialized objects in the formerly uninitialized memory.

Consider

```
template<class T> T* now_init(T* p [[ref_to_uninit]]) { return p; }
```

This function cannot be compiled with the initialization profile on because it provides a deliberate and explicit hole in the enforcement. It is a cast in disguise. Compiled without the initialization profile it is a no-op. If there was a way of writing it with the initialization profile on,

then the profile would be flawed. Having **now_init()** provides the user a way to avoid suppressing the initialization profile and offers an easy-to-spot point for code review. Something like that is needed in much code manipulating a mixture of initialized objects and uninitialized slots of memory, such as a **vector** implementation (§4.9, §5.3).

4.5. Templates and [[uninit]]

To avoid chaos, by default, objects of template argument types must be assumed to be initialized. However, uninitialized objects of such types are possible and sometimes even useful. Important examples include **unique_ptr<T>** and **span<T>**. Like for built-in pointers (**T***), the initialization profile needs to distinguish between initialized and uninitialized objects pointed to. We have three choices for classes that hold objects for which pointers or references can be returned:

- cannot return a reference to an uninitialized object (that must be the default).
- always returns a reference to an uninitialized object (the analyzer must know that).
- can return both initialized and uninitialized objects uniformly.

The third alternative cannot be implemented with a purely static guarantee:

```
class Wrong {
        int x = 7;
        int y [[uninit]];
public:
        int& get(bool b) { return (b) ? x : y; }        // error
};
```

Even if **[[uninit]]** had been part of the type, **get()** would be an error because a function can have only one return type. Separating the two alternatives requires some form of run-time test. Variants of this third alternative come up in several potential use cases.

When returning an object, we must identify the uninitialized ones. The default must always be that what is returned is a properly initialized object. The uninitialized version must be marked as such. For example:

```
class Slot {
        int x = 0b101010101010;
        int y [[uninit]];
pubic:
        Slot (int xx) : x{  xx } {}
        Slot() {}

        int& get_init() { return x; }
```

```
        int& get_uninit [[ref_to_uninit]] () { return y; }
};

Slot ii = 7;
Slot uu;                        // leaves uu.y uninitialized

int i1 = ii.get_init();         // OK
int i2 =  ii.get_uninit();      // error: tries to read from an [[uninit]]

int u1 [[uninit]] = uu.get_init();      // errror: tries to initialize an [[uninit]]
int u2 [[uninit]] = uu.get_uninit();    // OK
```

At least until we can gather a lot of experience, we must choose a simple solution. Templatize **Slot** and you have one solution. That – like all initialization profile – is compile-time enforced.

The weakness of this approach is that you can't ask a slot if it is initialized. That has to be managed externally. For example, if you implemented a vecto**r using** an array of **slot**s, you'd keep track of where the boundary between initialized members and "empty" slots is (as ever). In general, classes (incl. class template (such as **std::vector**) that completely control their allocation and deallocation and consistently return initialized objects are no separate problem (§5.4). Often, they can manipulate their "slots" directly relying on **[[uninit]]** and **[[ref_to_uninit]]** for guaranteed correctness without even introducing a **Slot** type.

It would also be easy to implement a "dynamic slot" class that used a **bool** to keep track of which alternative was present, but that would add a run-time cost, add memory cost, and force us to do run-time error-handling.

4.6. Lifetimes

To properly deal with uninitialized data, we need to look at both initialization and destruction. The reverse of initialization/construction is destruction. When the initialization profile is in force, the result of destruction is equivalent to an **[[uninit]]** annotation. It is an error to initialize an object twice (e.g., using **construct_at()**). Similarly, it is an error to destroy/uninitialize an object twice (e.g., using **destroy_at()**).

Reading an uninitialized object (except an **std::byte**) is erroneous behavior. Where the initiation profile is enforced that will be prevented at compile time.

Writing an uninitialized object must be an initialization:

- For a built-in type, that's simply a write-to or **construct_at()**.
- For a class object, that's **construct_at()**.

Initializing an object of a type with a constructor twice is an error, so **construct_at()** must require an uninitialized argument.

Enforcing this requires the simple flow analysis described in §1.2.

This relegates some unstructured techniques to code that suppresses the initialization profile. For example:

```
void init(span<X> s [[ref_to_uninit]], int i1, int i2)
{
        construct_at(&s[2],10);      // only s[2] is initialized
        X y = s[i2];                 // error: reading uninitialized? I2 could be != 2
        construct_at(&s[i1],10);     // error: double initialization? I1 could be == 2
}
```

More realistic examples would involve loops and conditions.

We have two choices:

- Allow code that is sufficiently simple for static analysis to guarantee initialization of all members and require profile suppression for more complex control structures.
- Fall back on erroneous behavior for more complex control structures.

To make the initialization guarantee purely compile-time, we must choose the first alternative (§1.2).

Destruction has a similar pattern and similar constraints. For example:

```
void init(span<X> s, int i1, int i2)
{
        destroy_at(&s[2]);
        destroy_at(&s[i1]);    // error: double destruction? I1 could be == 2
}
```

What is "sufficiently simple" for handling lifetimes? By keeping a separate vector of indicators of which slots are live we can cope with any use I can think of, but we cannot provide guarantees:

```
vector<Slot> v;          // §4.3
enum class Live { uninit,init };
vector<Live> live_slot;
// …
if (live_slot[x]==Live.init) {
        int xx = v[x];
        // …
}
```

For support, we could turn to runtime testing: add something like **Live** to a **Slot** template and test it upon every access. That would be simple, but too expensive for many uses.

Alternatively, or additionally, we could provide something like **vector**'s use of the first part initialized and the second part uninitialized with a range check and operations to move the barrier between them. Here is a much-simplified sketch of such a function template:

```
template<class T>
struct Vector_memory {
        T* elem [[ref_to_uninit]];
        int no_of_elem;
        int no_of slots;

        Vector_memory(int ne, int ns)
                : elem{(T*)malloc((ne+ns)*sizeof(T))},
                  no_of_elements{ne},
                  no_of_slots{ns}
                {
                }
        ~Vector_memory() { free(elem); }
};
```

Vector_memory knows only about uninitialized memory.

```
template<class T>
class Vector_memory {
        vector_memory<T> mem;
public:
        Vector(int ne, int ns, T& val = T{})
                : mem{ne,ns}
                {
                        uninitialized_fill(mem.elem,mem.elem+ne,val);
                }

        T& operator[](int i)  {
                if (0<=i && i<mem.no_of_elem)
                        return *now_init(mem.elem[i]);   // this slot has been initialized
                throw Bad_index{};
        }

        ~Vector() {destroy{mem.elem,mem.elem+mem.ne); }

        template<class T>
```

```
        void vector<T> reserve(int newsz)
        {
                If (newsz <= mem.ne+mem.ns) return;
                void*p = malloc(newsz*sizeof(T));
                unititialized_move(mem.elem, mem.elem+no_of_elem, (T*)p);
                mem.no_of_slots = newsz- mem.elem+no_of_elem;
                void* pp = mem.elem;
                mem.elem = p;
                free(pp);
        }

        void push_back(const T& x) {
                If (mem.no_elem==mem. no_slots)
                        reserve(mem.no_of_elem==0 ? 8 : 2*mem.no_of_slorts)
                construct_at(mem.elem+mem.no_of_elem,x);
                ++mem.no_of_elem;
                --mem.no_of_slots;
        }
};
```

The main point here is that the code is quite conventional (for this kind of C++ code). It is guaranteed free from initialization errors except for the call of **now_init()** call in the subscript operator. It is only called from within an initialized range but verifying that statically would be expensive and in some similar cases impossible. Having **now_init()** saves the user from suppressing the profile and offers an easy-to-spot point for code review.

We can, of course, make other kinds of errors (e.g., the error-handling is basically missing here), but that's the task for other profiles and – for logic errors – the skills of the programmer.

Consider another important use case, recycling a buffer. Here, reading or writing of a message should be once only and allocation should not be done while passing a stream of messages. Such code is – in many variations – used in much high performance and low latency applications. Again, this is just a sketch illustrating the techniques as they relate to initialization:

```
std::byte inibuf [[uninit]] [imax];      // or allocated in dynamic storage

class Message { /* operations to read typed members from a byte buffer rep */ };

int read(std::byte* p [[ref_to_uninit]], int max);  // some input function
                                                    // read at most max bytes into *p
                                                    // return the number of bytes read

while(live) {    // read into inbuffer and process the messages stored as bytes
```

```
        int n = read(ibuf,imax);        // some input function
        span<Message> messages {
                *(Message*)now_init(ibuf),          // this buffer contains Messages
                n/sizeof(Message)
        };
        for (Message& m: messages) {
                // … process a message from the buffer …
        }
        destroy(messages);
}
```

Note that assigning to an uninitialized **std::byte**, as read must do, is allowed. In addition to initialization, we require at least one explicit type conversion (bytes to messages done in **now_init()**) and must ensure that we don't have range errors. This is the kind of low-level code where various forms of suppression of guarantees are tempting (and common in "safe" languages).

I expect that in essentially all performance-critical cases **Message** will not have a destructor (it is after all an access mechanism to a buffer of bytes) and that the **destroy(messages)** will be a no-op.

4.7. Existing support for uninitialized

Many of our vocabulary types are templates. If they completely manage their own memory (like **std::vector** does) they are no separate problem.

Unfortunately, some can be initialized with uninitialed "objects" and we must not break significant amounts of code that isn't broken to provide guarantees. Consider:

```
unique_ptr<T> p (new T);
```

Is the **T** pointed to initialized or not? Do users of **p** care? They should care. The initialization profile must catch misuses. That implies that code like that must work as written if **new T** provides an initialized object and we must provide the user a way to handle the uninitialized cases. Unfortunately, **make_unique()** has equivalent problems relative to initialized/uninitialized.

Even **vector** is vulnerable:

```
int x [[uninit]];
vector v(10,x);                 // error
```

There are also templates that we want to take uninitialized memory.

By default, a template function's argument must be initialized; if not, it must be marked **[[uninit]]**. A pointer argument must point to an initialized object; if not, it must be marked **[[ref_to_uninit]]**. For example:

```
template<forward_range auto R, auto V>
        requires constructible_from<*iterator_v(R),V>
void uninitialized_fill(R r [[ref_to_uninit]], V val);

int a1[] = {1,2,3};
int a2[3] [[uninit]];

uninitialized_fill(a1,10);      // error: a1 is already initialized
uninitialized_fill(a2,10);      // OK
```

How do we know that **R** refers to something and what? After all, it isn't a (built-in) pointer or reference. We don't have to. Instead, when compiling **uninitialized_fill()**, writing through pointers (things that refer to elements) must be done using **construct_at()** (see §5.2) or profile suppression must be used.

Class and function templates that can take only either initialized or uninitialized arguments are not a problem. We can simply catch "the wrong alternative" at the call site, and in almost all cases, the usual rules from the last decades already works. The initialization profile just needs to validate that with a simple check.

How does the compiler know that **uninitialized_fill()** initializes?

```
int a2[[uninit]] [3];
int x = a2[0];          // error

uninitialized_fill(a2,10);
int y = a2[0];          // OK. Now a2 is initialized
```

We could simply require that the compiler knows about the semantics of standard-library functions such as **uninitilized_fill()**, but that's not general. Alternatively, we could introduce an attribute **[[now_init]]** (§4.4).

Typically, that test for or against uninitialized can be placed in a concept. For example, place a check of **[[uninit]]** in **constructible_from<>** (where it belongs), and we are back to the conventional user code:

```
template<forward_range auto R, auto V>
        requires constructible_from<*iterator_v(R),V>
void uninitialized_fill(R r, V val);
```

```
int a1[] = {1,2,3};
int a2 ] [[uninit]] [3];

uninitialized_fill(a1,10);      // error: a1 is already initialized
uninitialized_fill(a2,10);      // OK
```

This relies on the static analyzer (e.g., a compiler) "seeing" attributes, such as **[[uninit]]** and **[[ref_to_uninit]],** when verifying a template argument – r**equires**-clauses enable that. It does not assume that **[[uninit]]** is part of the type system the way **const** is; doing that would make it possible for code under the initialization profile to have a different meaning than the same code compiled without it.

4.8. Concept-based overloading

It is easy to define a function template that can be called with a **[[ref_to_uninit]]** argument only. For example:

```
template<class T> concept Pointer_to_uninit
        = Pointer<T> && requires(T* p) {  T* q [[ref_to_uninit]] = p; }
template<Pointer_to_uninit T> int f(T* p) { /* … */ }

int ii = 7;
int ui [[uninit]];

int y = f(&ii);          // error: ii is initialized
int x = f(&ui);          // OK
```

We could imagine overloading a pair of function templates to choose between a **[[ref_to_uninit]]** and an ordinary pointer to initialized object:

```
template<Pointer_to_uninit T> int f(T* p) { /* … */ }
template<Pointer T> int f(T* p) { /* … */ }

int ii = 7;
int ui [[uninit]];

int y = f(&ii);          // OK: calls to f) for initialized
int x = f(&ui);          // OK: calls the f() for uninitialized
```

This would result in code compiling under the initialization profile but failing to compile without it. That is acceptable because such code relies on a function that is part of the initialization

profile support. Without the initialization profile the definitions of **f()** would be rejected as a double definition.

However, we could also try to write a pair of function templates that resolved to different alternatives with and without the initialization profile. For example, we can define concepts to distinguish initialized and uninitialized pointers of the same type:

```
template<class T> concept Pointer_to_uninit
        = Pointer<T> && requires(T *p) {  T* q [[ref_to_uninit]] = p; }

template<class T> concept Pointer_to_init
        = Pointer<T> && requires(T* p) { *p = 7; }
```

There is a potentially serious implementation problem here, which will be discussed below, but in principle, this could work because all the init/uninit information needed to resolve it is in scope. For the moment assume this will work.

Now we can overload on those:

```
template<Pointer_to_uninit T> int g(T* p) { /* … */ }

template<Pointer_to_init T> int g(T* p) { /* … */ }
```

If allowed, with the initialization profile this would resolve to

```
int y = g(&ii);         // OK: calls the g() for initialized
int x = g(&ui);         // OK: calls the g() for uninitialized
```

Without the initialization profile, the **g()** assuming initialized has a stricter requirement than the other **g()** so this would resolve to

```
int y = g(&ii);         // OK: calls the g() for initialized
int x = g(&ui);         // OK: calls the g() for initialized
```

This violates the rule that code has the same meaning with and without the initialization profile. This is a silly example, but how do we prevent it. We have three choices:

- Accept such examples (thereby allowing violation of a fundamental principle)
  - This is not acceptable.
- Catch such examples at the point of definition
  - This implies a check for each template function
- Catch such examples at the point of use
  - This implies a check at each point call of a template function

Those checks will, of course, be done only when the initialization profile is requested. The last alternative is the most flexible but would involve the initialization profile in every template function call. That is not desirable.

So how can we specify a simple check of function templates? The **requires**-clauses can be arbitrarily complex, so we must rely on the subset-of-superset technique to make the test manageable:

- Unless a template declaration is an overload nothing need to be done
- If a template declaration is an overload, we only look further provided **[[uninit]]** or **[[ref_to_uninit]]** appears in the constraints.
- The overload will be accepted only if the overloads involving **[[uninit]]** or **[[ref_to_uninit]]** are identical when we ignore the provided **[[uninit]]** or **[[ref_to_uninit]]** in the constraints.

That rule allows simple overloads on initialized/uninitialized at a minimal cost. If someone wants more general rules, they must provide a plausible use case and a plausible algorithm **f**or accepting it.

Consider again:

```
template<class T> concept Pointer_to_uninit
        = Pointer<T> && requires(T *p) {  T* q [[ref_to_uninit]] = p; }

int ui [[uninit]];

int x = f(&ui);          // OK: calls the f() for uninitialized
```

How does the information that **ui** is uninitialized get into the **requires-**clause? Is the **T** in **T* q** the deduced type or the type of **ui** as presented as an argument? If the former, the **[[ref_to_uninit]]** of **ui** has been stripped and the requires clause will never fail. If the latter, all is well. The implementers are looking into this problem.

If the former, we need an alternative mechanism for getting init/uninit information into a template. This can be done, but the suggested solution is far more elegant. Consider a couple of alternatives (presented to demonstrate that the problem isn't fatal):

Using **[[uninit]]** and **[[ref_to_uninit]]**, we can distinguish what is right and what would be an error. That's all we need for a static analyzer, and we can define

span<int> taking initialized **int**s and not uninitialized **int**s

And

Uninitialized_span<int> taking uninitialized **int**s but not initialized **int**s

That's enough to provide safety guarantees.

What we can't do without help is to define a

span<int> that distinguishes between initialized and uninitialized **int**s

But that's what we need for simple and elegant use.

If we can't do the overloading presented above (and the implementers are looking into seeing if that is possible), we need a way of turning the initialized/uninitialized distinction into a check that we can use to guide overloading (a kind of tag dispatch). It should be easy to add one or two intrinsics to deliver what the compiler already understands:

is_init()

And

is_ref_to_init()

Given those, we can do all kinds of tag dispatch to get around our inability to encode attributes in types to smuggle them through type deduction (a problem that I underestimated).

Given the intrinsics, we can write

```
Int x [[uninit]];
Int* p [[ref_to_uninit]] = &x;

bool init = is_ref_to_init(p);

template<class T> class span {
        span (T* p, bool init) {
                if (init)
                   init_span_inplementation(p);
                else
                   uninit_span_implementation(p);
        }
```

4.9. Mixing initialized and uninitialized

Consider mixing uninitialized and initialized data. This is problematic. The problem comes in the relatively few cases (in the global picture of things) where a class or a function needs to pass along either initialized or uninitialized arguments. Static initialized/uninitialized analysis cannot be perfect. For example:

```
int* f(int x)
{
        return (0<x) ? (int*)malloc(x*sizeof(int)) : new int[x];
}
```

Providing an incomplete guarantee, especially one that is often valid, leads to overconfidence and becomes a source of subtle bugs. That's exactly the kind of problem the initialization profile (and other profiles) is aimed to prevent. So, if we want to do such mixing, we'd have to suppress the initialization profile.

We could imagine adding the initialized/uninitialized distinction to the type system but would add another alternative to every read and write for the compiler to consider. This would slow down compilation, not just for the initialization profile but for all code.

Adding initialized/uninitialized to the type system would imply that code could not be validated with the initialization profile on a modern compiler and the relied on for that code compiled with an older compiler.

What we can do is to use concepts to overload templates (§4.4)

5. Class objects

If a class has a constructor, every member not marked **[[uninit]]** must be initialized by the constructor. Members marked **[[uninit]]** must be initialized before they are exposed to users of the class

If a class doesn't have a constructor, unless not marked **[[uninit]]** at the point of definition, every member must be initialized.

This allows profiles to ensure type safety as it relates to initialization.

5.1. Constructors

In most cases, providing a constructor that initializes every member is the easiest to use as well as the easiest for the compiler to validate.

When a constructor is defined with the initialization profile requested, it is checked that it initializes every member (except those marked **[[uninit]]** or **[[ref_to_uninit]]**).

If every TU is compiled with the initialization profile, all is well. If it is not (as will be common for years), this is the best we can do anyway. The issue of differing profiles for different TU and modules is dealt with in the Profiles framework [GDR'26], rather than in an individual profile.

Note that complex code in a constructor can defeat static analysis and will be rejected (§1.2). Avoid that; it is not all that hard.

5.2. Member initializers

The simplest and most direct way of initializing a member is through explicit initialization. For example:

```
class X {
        int m1 = 7;
        int m2;
public:
        X(int x) : m2{x} {}
        // …
};
```

This is what the initialization profile requires.

If a member isn't initialized in one of those two ways but rather initialized in the constructor body, it must be declared **[[uninit]]**. Often, static analysis (§1.2) can determine that it really is given a value before use. For example:

example:

```
class X {
        int* p [[uninit]];
        int x;
public:
        X(int v) : x{v}
        {
                If (v<0 or sys_max<=v) error(v);
                p = new int(v);
        }
        // …
};
```

Is **p** is still **[[uninit]]** after construction? To answer "no", the compiler will have to examine all flow paths in all constructors to make sure they all initialized **p**. Alternatively, we could use initialize **p** elsewhere and use **now_init()** or suppression to expose that unverified **p** or ***p** to users. Often, using a "pseudo initialization" is a better choice. For example:

```
class X {
        int* p = nullptr;
        int x;
public:
```

```
        X(int v) : x{v}
        {
                If (v<0 or sys_max<=v) error(v);
                p = new int(v);
        }
};
```

In most such cases, the optimizer can eliminate the redundant initialization.

Often, an even better alternative is to use a member of a type that performs its own checks rather than more primitive types. For example:

```
class X {
        unique_ptr<int> p;
public:
        X(int x) : p{make_unique(x)} {}
};
```

Default construction works as ever. For example:

```
class X {
        string s1;
        string s2;
public:
        X(const char* ss1, const char* ss2) : s1{ss1} { s2 = ss2; }
};
```

The use of the member initialization (for **s1**) is preferred to the assignment (of **s2**) as being more readable and more efficient unless the optimizer is smart enough to eliminate the default initialization of **s2**.

5.3. Classes exposing uninitialized memory to users

Some classes need to expose uninitialized members to users, e.g., some memory pools.

Consider:

```
struct X {
        Y m1;           // maybe error
        Y* m2;          // error, see constructor
        int arr[10];    // error, see constructor

        X(int x) : m2{ (Y*)malloc(x)} {}
        // …
};
```

The initializer profile deems **m1** (if **Y** doesn't have a default constructor), **m2** (**malloc()** returns a pointer to uninitialized memory), and **arr** (has uninitialized members) errors because they try to expose users of **X** to uninitialized objects without any indication of that.

Thus, we must add attributes:

```
struct X {
        Y m1;
        Y* m2 [[ref_to_uninit]];
        int a [[uninit]] [10];

        X(int x) : m1{x}, m2{ (Y*)malloc(x)} {}
        // …
};
```

For uninitialized class objects, we'll have to use **construct_at()** rather than simple assignment to do the initialization.

5.4. Classes without constructors

When defining objects of classes without constructors we must initialize all members in their definitions. For example:

```
struct S { int x; string s; };

void f()
{
        S s1 = {1,"foo"};       // OK

        S s2 = {1};             // OK, s2.s defaults to ""

        S s3;                   // error: s3.x uninitialized
        s3.x = 1;
        s3.s = "foo";

        S s4 [[uninit]];        // error: s4.s is default constructed
        s4.x = 1;
        s4.s = "foo";
}
```

The member **s4.s** is a **string**, and **string** has a default constructor. That means that **S s5 [[uninit]];** declares **s4.s** as both initialized and uninitialized. That's an error (even before we start considering delayed initialization).

Consider

```
S s5 [[uninit]];
construct_at(&s5, {1, "bar"});
s5.s = “bar”;          // OK now s5 is initialized
```

This is accepted, but now the programmer and the compiler have to keep track of whether the initialization has taken place (§1.3).

For a single object, that’s complicating, unnecessary, and a potential source of errors. Immediate initialization is simpler and clearer.

```
S y = {1,”foobar”};
```

Don’t introduce a variable until you have an initializer for it.

5.5. Arrays

Like **struct**s, arrays can be initialized in the definition or marked **[[uninit]]**. For example:

```
int arr1[] = {1,2,3,4};
int arr2[10] [[uninit]];
```

However, it is quite common and quite reasonable for large arrays to have initialization delayed and initialized by – sometimes complex – algorithms (e.g., the input buffer example in §4.6).

The **uninitialized_*()** family of range algorithms are close to ideal for initializing arrays and can be verified, but using it is tricky. For example:

```
X arr3[20];                          // error unless X has a default constructor
ranges::uninitialized_fill(arr3,xval); // error if X has a default constructor
```

So we get one error or the other.

Adding **[[uninit]]** helps when the programmer knows **X**:

```
X arr4[20] [[uninit]];               // error if X has a default constructor
ranges::uninitialized_fill(arr4,xval);
```

5.6. Unions

Consider first the simplest solution (aligned with the handling of classes with public members):

```
union S {
        int x;
        string s;
};
```

```
S x1 = 7;

S x2;    // error

S x3 [[uninit]];
x3.x = 9;              // OK? (no)

S x4 [[uninit]];
x4.s = "foo";          // OK? (no)
```

Here, the handling of **x2** and **x2** is simple and obvious. Should delayed initialization of **x4** be accepted? No, because it would be an erroneous assignment. To avoid treating union members of different types dramatically different, **x3** should also be banned.

6. What is initialization?

To specify and implement the rules that suppress **[[uninit]]** and **[[ref_to_uninit]]** after initialization, we have to define what is considered initialization.

For now, I suggest:

- For class objects use **construct_at()**.
- For built-in types, use **construct_at()** or ordinary assignment.
- For ranges, use the **uninitialized_*()** family of functions.
- For more complex initialization, use functions marked **[[now_init]]** (§6.2)

6.1. Possible generalizations

Even simple extensions are more tricky than they appear. For example, should range-**for** loops be accepted? For example:

```
int aar6 [[uninit]] [10] ;
for (int& x : aa6) x=0b1010101010101010;
```

To allow that, the initialization profile would have to recognize the loop as an initialization, verify that the loop body really is initialization, and that the loop really complete.

We would have to handle partial initialization. For example:

```
int aar7 [[uninit]] [10];
// … no use of aar3 here …
for (int x = 0; x<10; x+=2) aar7[x] = 2;          // initialize even
// … restricted use of aar3 here …
```

```
for (int x = 1; x<10; x+=2) aar7[x] = 1;          // initialize odd
```

Even the simplest and most reasonable generalizations leave us with serious problems of verification. Consider:

```
X aar8 [[uninit]] [10];
// … no use of aar7 here …
int count = 0;
while (cin && count<10)
        arr8[count++] << cin;
// … we can use aar7 here …
```

This is an example of the problem of determining what an initialization is and whether it is complete. Also, to verify this, we'd have to track the use of **count** and how would we handle the case where **cin** read only **5** values? Real-world examples can be far more complex. The idea of delayed initialization leads us to complexities.

Suppressing the initialization and using a recognized initialization method (e.g. **uninitialized_fill()**) can be used to delay initialization without suppressing the profile. For example:

```
X arr9 [[uninit]] [20];
// … any recognized initialization technique used here …
// … use arr9 here
```

Initialization too complex for the analyzer to recognize is best avoided but can be handled by (the unverified) **now_init().** For example:

```
X arr10[[uninit]] [20];
// … any initialization technique used here …
span<X> s{*now_init(arr10)};
// … use s and not arr10 here …
```

In general, this technique is not verifiable, but it is far more manageable than suppressing the initialization profile would be, and is an obvious target for code review and more advance analysis

6.2. [[now_init]]

Often a pointer to uninitialized memory is passed to a function for initialization. For example:

```
T* initialize1(T* p [[ref_to_uninit]]);

void initialize2(T* p [[ref_to_uninit]]);
```

As the proposal stands, there is no way (except **now_init()**) to tell the analyzer that after the call of **initialize1()** or **initialize2()**, ***p** has been initialized. There are many such functions, we need a verifiable way of expressing that. Also, there are many functions that simply passed a pointer to uninitialized on to yet another function to do "the real work. So we need

- A way of saying a pointer points to uninitialized memory (we have that).
- A way of saying that a pointer must point to initialized objects after the call.

Consider a **[[must_init]]** attribute:

```
T* initialize1(T* p [[ref_to_uninit]]);

void initialize2(T* q [[must_uninit]]);
```

Here q(like p) must point to something uninitialized. However, after the call what q points to must have been initialized. Together **[[ref_to_uninit]]** and **[[must_init]]** cover the use cases, I have been able to think of. **[[must_init]]** implies **[[ref_to_uninit]]** so we avoid verbosity.

7. Code that does not obey **profiles::uninitialize**

The C++ type system doesn't distinguish between initialized objects and uninitialized memory – that's the job of the initialization profile. Some code doesn't obey the initialization profile and probably "never" will, being C code or trusted old-style code. There is a lot of such code, e.g., some operating systems.

Range errors can break the initialization profile (and most other profiles) but preventing that is the task of the invalidation and range profiles so are not discussed here. The assumption is that rage errors are prevented.

The major problem is system headers. Annotating C-style code with **[[uninit]]** and **[[ref_to_uninit]]** is not difficult, but system headers and much other foundational C style code is controlled by organizations that (at least in the short term) will not accept C++ attributes. This leaves us with a difficult problem.

8. How to specify this in the standard?

For profiles in general it is important that we specify the guarantee offered, rather than just long lists of places in the language affected. Such lists are necessary in addition to the specification of the guarantees but have a nasty tendency to be incomprehensible to non-experts and incomplete. If we list 27 cases, how can we be sure that the right number isn't 26 or 28? Whatever lists we

construct to help implementers must be checked about the general guarantees that they exist to deliver.

Having both guarantees and detailed specifications also allows us a way of consistency checking of a profile specification.

We need a common style for specifying standard profiles.

9. Early draft standard text

This is clearly an early draft. It needs input and review.

9.2. Guarantees

???

- Every object is initialized at the point of definition or marked **[[uninit]]**.
- Every object marked **[[uninit]]** at its point of definition is initialized by **construct_at()** or equivalent before use.
- Only local static analysis is used. Only very simple static analysis is used. For run-time alternatives, all execution paths are considered and for acceptance all alternatives must provide the desired solution.
- Random access to uninitialized arrays is banned. Instead use range algorithms (§5.4). Note: when random-access is needed the initialization profile must be suppressed.

9.3. Attributes

The profile defines three attributes

- **[[uninit]]** marks an object as uninitialized.
- **[[ref_to_uninit]]** marks a pointer, reference, or "smart pointer" as pointing to zero or more uninitialized objects.
- **[[must_init]]** implies **[[ref_to_uninit]]** and marks that pointer, reference, or "smart pointer to be initialized.

9.4. Classes

???

9.5. Pointers

???

- Pointers to functions are exactly like other pointers with respect to initialized/uninitialized.

- Virtual functions are guaranteed to be initialized by language rules (as ever).
- References are guaranteed to be initialized by language rules (as ever).

9.6. Concepts

Add uninitialized requirements to concepts expressing the requirement to take uninitialized memory as arguments.

- ???

9.7. Library functions

The standard-library support consists of a single function and the use of concepts that enforces the requirement to take uninitialized arguments to existing functions already requiring uninitialized memory as arguments.

- Apply **[[ref_to_uninit]]** to arguments and return types that refer to uninitialized objects.
- Add **now_init()** to the library.

References

- [TK24] Thomas Köppe: Erroneous behaviour for uninitialized reads. P2795R5. 2024-03-22.
- [LLG25] Marc-André Laverdière, Christopher Lapkowski, Charles-Henri Gros: A Safety Profile Verifying Initialization. P3402R3. 2025-05-16.
- [GDR25] Gabriel Dos Reis: C++ Profiles: The Framework. P3589R2. 2025-05-19.

Acknowledgements

Thanks to the work and comments of Vinnie Falco, Marc-André Laverdière, Christopher Lapkowski, Charles-Henri Gros, Thomas Köppe, Jason Merrill, Gabriel Dos Reis, Herb Sutter, David Vandevoorde, Vassill Vassililev, Michael Wong, the LEWG, and the EWG for work on the initialization profile.

Appendix: Attribute placement

Should the attributes **[[uninit]]** and **[[ref_to_uninit]]** be placed after the type, after the name in a declaration, or after any part of the type? The answer depends on logic, aesthetics, and the implementability in the major compilers. The conclusion is reflected in the main text of this paper. The alternatives and reasoning are presented here.

Consider:

```
int x [[uninit]];
int [[uninit]] y;

int* [[ref_to_uninit]] p;
int* q [[ref_to_uninit]];
```

There are three alternatives:

- **int* p3 [[ref_to_uninit]] = &x;**
  Plausible: **p3** is a pointer to something uninitialized.
- **int* [[ref_to_uninit]] p4 = &x;**
  Plausible: **p4** is an **int*** that points to something uninitialized,
  but someone might want to have an uninitialized marker in a typedef
  and that could lead to serious complexity.
- **int* p5 = &x [[ref_to_uninit]];**
  Odd: what is referencing something uninitialized, **x** or **p5**?

The same choice exists for arrays:

- **int* [[ref_to_uninit]] arr1 [10];**
  Plausible: the attribute is next to the type it describes.
- **int* arr2 [[ref_to_uninit]] [10];**
  Plausible: the attribute is just after the name as in variable definitions.
- **int* arr3 [10] [[ref_to_uninit]];**
  Plausible: the attribute is at the end, where the initializer would have been

Interestingly, in my examples, I had consistently used the first alternative for functions and the last for arrays. That's not consistent, likely to generalize, or easy to explain with a single rule. The middle choice is that.

So, I place the attribute after the name (where the initializer is for initialized variables). Also, the attribute applies to the object rather than the type of the object.

The attribute doesn't change the meaning of a program. That is, it doesn't change the type. It gives the analyzer information needed to reject programs that fail to properly initialize.

Now, digging a bit deeper into the possibilities, we see that we could do without **[[ref_to_uninit]]** altogether by allowing **[[uninit]]** to be placed anywhere in a type. For example:

- **int [[uninit]] * p6 = &x;**
  Plausible: the **p6** is a pointer to an uninitialized **int**.

This is tempting. It's general and saves us from introducing an attribute. However, that generality implies complexity. Consider:

```
// not proposed:
int [[uninit]]* p7 = nullptr;          // p7 points to something uninitialized
int* p8 [[uninit]];                    // p8 is uninitialized; must points to initialized
int [[uninit]]* p9 [[uninit]];   // p9 is uninitialized; must point to uninitialized
```

This is the simplest example. Types can be arbitrarily complex. For example:

```
void f88(int[[uninit]](&x)[10]);       // reference to array of uninitialized ints
```

Implementers point out that embedding attributes in types is distinctly nontrivial, that the implications of doing so are not completely specified in the standard, and that the existing implementations don't completely cover these cases. The fundamental problem is that it is hard (in the compiler architectures) to convey the provided information from the point where it is stated (somewhere in the type specification) to the point where it is used (in the declaration) without losing the meaning of the attribute (depending on the placement).

I fear that people will see **[[uninit]]** as part of the type and some will demand that it becomes part of the type, rather than just a simple handle for the initialization profile. Making initialized/uninitialized part of the type system would add complexity, development time, and compile time. In particular, it would affect the overload rules and break the important principle that profiles don't change the meaning of code.

We shouldn't go there.

We cannot completely escape the placement problems because we have to deal with "smart pointers" and function declarations. For example:

```
smart_ptr<int [[uninit]]> p;          // pointer to uninitialized int
void fct1(int [[uninit]]);            // error: argument passing is initialization
int fct(int* [[ref_to_uninit]]);
```

Here, there is no name to which we can attach the attribute. However, the attribute will always come last, in the position after the name, had there been one. This is a point where implementer input is essential. I proceed on the assumption that this notation is manageable.

A variant of the same problem occurs with function return types:

- **int* [[ref_to_uninit]] f1(int);**
  Plausible: the attribute is next to the type it describes.
- **int* f2 [[ref_to_uninit]] (int);**
  Plausible: the attribute is just after the name as in variable definitions.
- **int* f3 (int) [[ref_to_uninit]];**
  Odd: the attribute is very far from what it refers to.

The first alternative is the cleanest, the second is the easiest to implement and consistent with what's proposed for variables, and the third potentially confusing.

For now, I'm using what I consider the cleanest for arrays, functions, and template arguments; that is, annotations for return types are placed by the name and annotations on argument types after the argument type

```
int* f1 [[ref_to_uninit]] (int);
int arr [[uninit]]  [7];
void f2(int* [[ref_to_uninit]]);
void f3(int* arg [[ref_to_uninit]]);
auto f4 [[ref_to_uninit]] (int) -> int*;
unique_ptr<int [[uninit]]> up;
```

When doing initial implementations, we should look out for cleaner and easier to implement alternatives that doesn't change the semantics. As ever, removing the initialization attributes gives us a program with unchanged semantics.

The placement of [[now_init]] follows the rule for return types:. the attribute is next to the name it describes

- **void fct1 [[now_init]] (int* p [[ref_to_uninit]]);**

This is what is needed to fit existing initialization functions into the validation framework (§???).