Doc. No. P4222R0
Date: 2026-02-22
Audience: EWG, SG12, SG20, SG23
Author: Bjarne Stroustrup
Reply to: bjarne@stroustrup.com

An initialization profile

Bjarne Stroustrup (www.stroustrup.com)
Columbia University

Abstract

A design of the initialization profile is being processed by the EWG [LLG25]. This note is in support of that, further emphasizing its rationale, and in a few cases suggesting simplifications.

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.
- The profile is completely compile-time enforced.
  - Language use is limited to make that possible and affordable.
- Every object that is implicitly initialized under current rules is initialized (as ever).
- An object intended to be uninitialized can be marked **[[uninitialized]]**.
  - An uninitialized object can be explicitly initialized (e.g., using **construct_at()**).
  - An initialized object cannot be marked **[[uninitialized]]** or explicitly initialized.
- A pointer, reference, or "smart pointer" can be marked **[[ref_to_uninit]]**.
  - An object marked **[[ref_to_uninit]]** cannot be initialized to point to initialized objects.
- Every use of **[[uninitialized]]** and **[[ref_to_uninit]]** can be verified at compile-time.
- Static objects must be compile-time initialized.
- Code compiled without enforcing this profile but doesn't violate its rules has exactly the same meaning as a compilation that enforces the profile.

Terminological differences make it tricky to see if unintentional differences with [LLG25] have been introduced, and what is presented here needs to be merged with [LLG25] and the rationale updated and improved.

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. Here is a definition with supporting discussion of alternatives and my 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 (except **std::byte**).
- No complex flow analysis required.
- No run-time tests are needed to implement this profile.
- Trust a constructor to initialize all public members if its definition isn't visible.
- Implicit initialization (through a default constructor or a language rule) is initialization.
- There is a way to pass uninitialized memory out of a scope.
- Initializing an object twice is an error.
- The design is simple enough for most developers to manage.

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 or written.

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 (§7).

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

There are many places in code where we can possibly want to 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 uninitialized objects are input buffers, but we might leave a variable uninitialized by mistake so that it later becomes the cause of an error. 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.

We do not aim to enable every possible technique. Techniques considered too error prone or too complex are relegated to code suppressing the initialization profile. For example, using random access to selectively initialize elements of an array of uninitialized objects cannot be statically validated (e.g., §6.3) so it must be banned and if needed relegated to a section of code that suppresses the initialization profile.

The initialization profile will be very widely used since its guarantees are relied on by most code and most other profiles. Therefore, its overhead of enforcing the 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 profiles that depend on it needs:

- Testing for erroneous behavior could be a run-time action – avoiding that is implementation defined and not guaranteed. The programmer must be prepared to cope with whatever that 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 uninitialized memory rather than just relying on suppression of the profile to handle cases where uninitialized memory is needed.

Note that (as usual with 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.

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 relying on specialized hardware support that's not available everywhere. Even **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.

The simplest solution would be to ban passing objects with directly accessible uninitialized memory out of a scope. That is, require that such data be private data members so that the obligation to probably 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. Alternatively, users would have to suppress the initialization profile without in-code hints of why that was needed.

I don't think suppression would be realistic. 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 passed to or from a function points to an object containing potentially uninitialized objects (e.g., an array).

Dealing with such areas of uninitialized memory requires us to deal with such through pointers and references, including iterators and other "smart pointers."

1.2. 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?" and "is this **for**-statement on a form accepted by this profile?" 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., **std::vector**) and member variables initialized in a constructor body rather than in a member initializer.

There are languages (e.g., Ada and C#) that simply require that a variable is assigned to before use. 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).
- 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.

Consequently, 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 marked. This is an application of the subset-of-superset strategy. Only after adding notation to simplify analysis can we subset to offer the guarantee.

To make static analysis affordable in general, I suggest to 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 initialize-before-use impossible to enforce statically (see §6.3 for an example). However, range-for and some standard-library algorithms enforcing an order of access can be handled.

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

2. Implicit initialization is initialization

A definition that invokes a default constructor is considered initialized (see §6.1).

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 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 exactly to control the timing of its initialization.

If the initialization is too complex to be guaranteed by these simple techiques, 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 that.

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 catches the most blatant errors it can cause.

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 to 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 simply too difficult (it is often flow dependent and passes function barriers).
- Tracing uninitialized memory through our programs is often impossible to do perfectly.

- 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 of 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 [[unitialized]];**
- Say nothing and let the compiler note when an object is uninitialized: **X x;**

The difference is purely syntactic. The **[[uninitialized]]** clearly indicates that it is just information to analyzers (e.g., 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 very rarely used.

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

```
tuple<int, int, int> t = {1, [[uninitialized]], 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 initializing with some "random" value (which can be costly and is error prone).

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

- is implicit so doesn't indicate whether the lack of initialization is intentional or not.
- doesn't extend to handle the cases where the information of whether an object is initialized or not needs to be passed between functions (§6.2).

Should **[[uninitialized]]** be considered part of the initialization profile and therefore spelled **[[profiles::uninitialized]]**? I think not. The information it conveys is generally useful and can/will be used by people who do not use the initialization profiles and most likely also by other profiles.

4.2. Pointers and references

As ever, a reference cannot be uninitialized. The initialization profile requires the same for pointers. 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 **[[uninitialized]]** 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. Instead, we introduce **[[ref_to_uninit]]** to help correctly manipulate a mix of initialized objects and areas of uninitialized memory Manipulating such a mix 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.

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 using such code, it will be relatively common, so I don't think a longer and more descriptive name would be appropriate; for example, **[[points_or_refers_to_uninitialized]]**.

For example:

```
Int* p1;                          // error: uninitialized pointer
int* p1 = nullptr;                // OK
int* p3 [[ref_to_uninit]]= &x;    // x must be uninitialized

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

template<forward_range auto R, auto V>  // just an example, not the std version
        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] [[uninitialized]];

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

In that last example, how do we know that **R** refers to something and what? We don't have to. Instead, when compiling **uninitialized_fill()**, writing through pointers must be done using **construct_at()** or profile suppression must be required.

Similarly, an uninitialized array can be returned from a function. For example:

```
int* [[ref_to_uninit]] allocate(int n);
```

Choosing the notation **[[ref_to_uninit]]** raises the question**s** "why **[[uninitialized]]** rather than the shorter and equivalent **[[uninit]]?**" and "why **[[ref_to_uninit]]** rather than the equivalent but longer **[[ref_to_uninitialized]]**?" I'm sure that different people will have difference preferences. Personally, I have a slight preference for the shorter versions.

5. Repeated access

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.
- For a class object, that's **construct_at()**.

Initializing an object of a type with a constructor twice is an error.

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

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

```
void init(span<X> s [[ref_to_uninit]], int i1, int i2)
{
        construct_at(&s[2],10);
        X y = s[i2];                    // reading uninitialized? I2 could be not 2
        construct_at(&s[i1],10);        // double initialization? I1 could be 1
}
```

More realistic examples would involve loops and conditions.

We have three choices:

- Allow code to be 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). Standard-library range algorithms and range-**for** loops are obvious examples of what we can handle and the **span** example above an obvious example of what we cannot.

6. Class objects

If a class has a constructor, every member not marked **[[uninitialized]]** must be initialized.

If a class – directly or indirectly – holds uninitialized memory, it must ensure that that memory is either not accessible from users or only accessible through references or pointers marked **[[ref_to_uninit]]**.

This allows profiles to ensure type safety.

6.1. Trust 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 initialization of an object is done by a constructor, the constructor is trusted to do proper initialization of the complete object. This rule is essential with the constructor is defined in a separate TU.

When a constructor is defined with the initialization profile requested, it is checked that it initializes every member (except those with an uninitialized indicator).

If every TU is compiled with the initialization profile, all is well. If it is not (as is common), this is the best we can do anyway.

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.

6.2. 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 **[[uninitialized]]** and simple static analysis (§1.2) is needed to determine that it really is given a value. For example:

example:

```
class X {
        int* p [[uninitialized]];
        int x;
public:
        X(int x) : m2{x}
        {
                If (x<0 or sys_max<=x) error(x);
                p = new int(x);
        }
};
```

If the static analysis is considered inappropriate, a "pseudo initialization" can be used:

For example:

```
class X {
        int* p = nullptr;
        int x;
public:
        X(int x) : m2{x}
        {
                If (x<0 or sys_max<=x) error(x);
                p = new int(x);
        }
};
```

In most such cases, the optimizer can eliminate the apparent overhead.

Often, a better alternative is to use an 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{x} {}
};
```

Default construction works as ever. For example:

```
class X {
        string s1;
        string s2;
public:
        X(const char* ss, 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 optimized is smart enough to eliminate the default initialization of **s2**.

Next, we must consider classes that expose members to users.

6.2. 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;
        Y* m2;
        int a[10];

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

The initializer profile deems **m1** and **m2** errors because they expose users of **X** to uninitialized objects without the type system reflecting that.

Thus, we must add attributes:

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

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

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

6.3. Classes without constructors

We require that Classes without constructors have all elements initialized in their definition. 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 = 1;
        S3.s = "foo";

        S s4 [[uninitialized]]; // OK? (no)
        S4.x = 1;
        S4.s = "foo";

        S s5 [[uninitialized]]; // OK? (no)
        S5.x = 1;
}
```

Should the delayed initialization of **s4** be accepted? No, because it would be an erroneous assignment when compiled without the initialization profile.

To avoid treating structs with different member types dramatically different, delayed initialization of struct members where of types where assignment and initialization are the same should also be banned. For example:

```
struct S { int x,y;}
S x; [[uninitilized]];
x.x = 1;
x.y = 2;
```

So how do we delay initialization of **struct**s, such as **struct**s in a buffer pool? Use **construct_at()** to make it clear that initialization is intended (rather than assignment). For example:

```
S s5 [[uninitialized]];
construct_at(&s5,1,2);
```

6.4. Arrays

Like structs, for large **[[uninitialized]]** arrays verifying that all elements are initialized we need the restrictions mentioned in §5 and §6.3. In general, using random access to initialize uninitialized elements must be banned as too complex for static analysis and impossible in general. For example:

```
void f(int v)    // hypothetical handling of complex examples (not done)
{
        int a [[uninitialized]];
        int x = a[0];    // error: read of uninitialized
        a[1] = 7;
        int y = a[1];    // could be OK: a[1] has been initialized
        int z = a[v];    // would be OK if v is 1 and an error otherwise
}
```

A run-time solution is possible, but that would be very expensive.

6.5. 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 [[uninitialized]];
X3.x = 9;                // OK? (no)

S x4 [[uninitialized]];
X4.s = "foo";            // OK? (no)
```

Here, the handling of **x1** and **x2** is simple and obvious. Should the delayed initialization of **x4** be accepted? No, because if it was it would be an erroneous assignment when compiled without the initialization profile. To avoid treating union members of different types dramatically different, **x3** should also be banned.

7. How to specify this in the standard?

For profiles in general it is important that we specify the guarantee offered, rather than long lists of places in the language affected. Long lists 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?

We need a common style for specifying standard profiles.

<<This section is obviously incomplete. To be completed for R1 ASAP. It should reflect the design sections with greater precision, and we need to consult with CWG about the form of that?>>

7.1. Guarantee

Every object is initialized before use or marked **[[uninitialized]]**.

7.2. Flow analysis

Only local static analysis is used. Only very simple static analysis is used. For run-time alternatives, all alternatives are considered and for acceptance all alternatives must provide the desired solution.

Random access to uninitialized arrays is banned.

<<Consider If, switch, loops, algorithms (e.g., **uninitialized_fill()**), goto?>>

7.3. Attributes

[[uninitialized]] ???

[[ref_to_uninit]] ???

7.4. Classes

???

7.5. Pointers

???

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 of Marc-André Laverdière, Christopher Lapkowski, Charles-Henri Gros, Thomas Köppe, Gabriel Dos Reis, the LEWG, and the EWG for work on the initialization profile.