

Staying Sane With C++ Initialization Rules

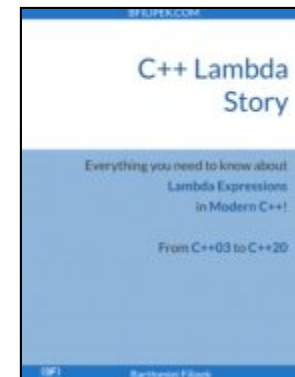
Down the Rabbit Hole...

About Me

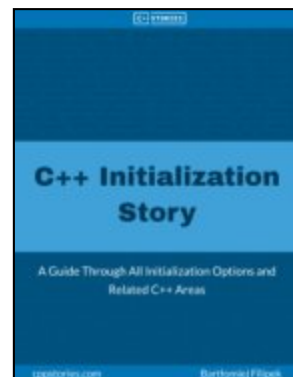
- Author of cppstories.com
- ~15y professional coding experience
- 4x Microsoft MVP, since 2018
- C++ ISO Member
- [@Xara.com](https://twitter.com/XaraDotCom) since 2014
 - Mostly text related features for advanced document editors
- Somehow addicted to C++ 😊



C++17 In Detail



C++ Lambda Story



C++ Initialization Story

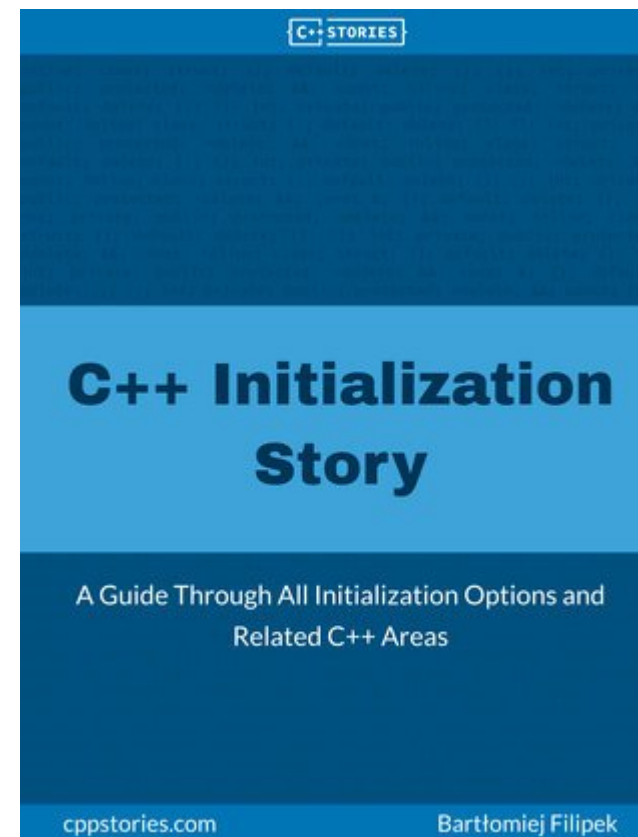


Xara Cloud Demo



The plan

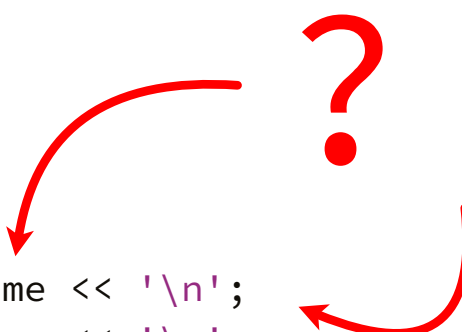
- *Simple types*
- **Non static data member initialization**
- *Experiments*
- *Constructors*
- *Non-local variables*
- *(Tricky) cases*
- *Summary*



<https://leanpub.com/cppinitbook>

Simple types

```
struct CarInfo {  
    std::string name;  
    unsigned year;  
    unsigned seats;  
    double power;  
};  
  
int main() {  
    CarInfo firstCar;  
    std::cout << "name: " << firstCar.name << '\n';  
    std::cout << "year: " << firstCar.year << '\n';  
    std::cout << "seats: " << firstCar.seats << '\n';  
    std::cout << "power (hp): " << firstCar.power << '\n';  
}
```



<https://godbolt.org/z/9GoPedjGY> (Clang)

name:
year: 1825205920
seats: 32767
power (hp): 0

or (GCC)

name:
year: 0
seats: 0
power (hp): 2.07438e-317

Default initialization

```
void foo() {  
    int i;           // indeterminate value!  
    double d;        // indeterminate value!  
    double x = d;     // UB !!  
    std::string name; // default ctor called  
}
```

This is the initialization performed when an object is constructed with no initializer.

Setting values to zero - Value initialization

```
CarInfo emptyCar{};
std::cout << "name: " << emptyCar.name << '\n';
std::cout << "year: " << emptyCar.year << '\n';
std::cout << "seats: " << emptyCar.seats << '\n';
std::cout << "power (hp): " << emptyCar.power << '\n';
```

name:
year: 0
seats: 0
power (hp): 0

```
int i{};           // i == 0
double d{};        // d == 0.0
std::string s{};   // s is an empty string
```

```
int j = {};        // other form of value initialization
std::string str = {}; // ...
```

```
CarInfo *p = new CarInfo{}; // Value Initialization
CarInfo *p = new CarInfo;   // Default Initialization
```

P2723R0 - Zero-initialize objects of automatic storage duration

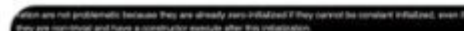


JF Bastien  @jfbastien@mastodon.social
@jfbastien

Wherein I propose that C++ initialize all stack variables to zero, preventing ~10% of CVEs.

Cost: none.

wg21.link/P2723R0



even are not problematic because they are already zero-inflated. If they cannot be constant inflated, even if they are non-zero and have a non-zero value after this inflation.

```
static char global[10]; // already zero-initialized
static int32_t global_status; // already zero-initialized

int main() {
    char buffer[128]; // not zero-initialized before this proposal

    struct {
        char name;
        int payment;
    } packet; // packet zero-initialized with this proposal

    union {
        char small;
        int big;
    } superset = 0; // getting zero-initialized with this proposal

    int value = 0; // 0
}
```

- **Read primitive:** this explicit could be formed from reading labels, security that resides on the stack or in registers, and using this information to use another bug. For example, looking an important address to below `ADDR`.
- **Write primitive:** an uninitialized value is used to perform a write to an arbitrary address. Such a write can be transformed into an execute primitive. For example a stack slot is used in a particular execution path to determine the address of a write. The attack can control the previous stack slot's content, and therefore control where the write occurs. Similarly, an attacker could control which value is written, but not where it is written.
- **Execute primitive:** an uninitialized value is used to call a function.

```
Here are a few examples:
```

```
int get_new_address(struct device *dev, struct user *u) {
    unsigned char *addr = (char *)0; // Get this memory
```

```
1. (Follow-up, address):
    return (unsigned)addr;
```

- The OS of every desktop, laptop, and smartphone that you own.
- The web browser you're using to read this paper.
- Many kernel extensions and userspace program in your laptop and smartphone, and
- Likely to your favorite videogame console.

The above codebases contain many millions of lines of code of C, C++, Objective-C, and Objective-C++.

The performance impact is negligible (less than 0.5% regression) to slightly positive (that is, some code gets faster by up to 1%). The code size impact is negligible (smaller than 0.5%). Compile-time regressions are negligible. Where overheads do matter for particular coding patterns, compilers would be able to obviate most of them.

The only significant performance/code regressions are when code has very large automatic storage duration objects. We provide an attribute to opt-out of zero-initialization of objects of automatic storage duration. We then expect that programmer can audit their code for this attribute, and ensure that the unsafe subset of C++ is used in a safe manner.

```

        backing = complete(backing, (300000000)); // map 🟡🟡🟡
    }
    return 0;
}

```

Example from [Jurya/Jurya](#): The attacker can control the value of `backlog` by making a previous function call, and ensuring the same stack slot gets reused.

Between 2017 and mid 2018, this feature would have killed 48 MSRC cases that involved uninitialized struct data leaking across a trust boundary. It would have also mitigated a number of bugs involving uninitialized struct data being used directly.

To date, we are seeing noise level performance regressions caused by this change. We accomplished this by improving the compilers ability to kill redundant stores. While everything is initialized at declaration, most of these initializations can be proven redundant and eliminated.

They use pure zero initialization, and claim to have taken the overheads down to within noise. They provide more

Aggregates

```
struct CarInfo {  
    std::string name;  
    unsigned year;  
    unsigned seats;  
    double power;  
};  
  
void printInfo(const CarInfo& c) {  
    std::cout << c.name << ", " << c.year << " year, " << c.seats << " seats, " << c.power << " hp\n";  
}  
  
int main() {  
    CarInfo firstCar{"Megane", 2003, 5, 116 };  
    printInfo(firstCar);  
    CarInfo partial{"unknown"};  
    printInfo(partial);  
    CarInfo largeCar{"large car", 1975, 10};  
    printInfo(largeCar);  
}
```

Without going into full definitions, an aggregate means a simple type (or an array) with all public data members, no virtual functions, and user-provided constructors.

Aggregates - C++20, Designated initializers

```
struct Point { double x; double y; };  
Point p { .x = 10.0, .y = 20.0 };
```

```
struct Date {  
    int year;  
    int month;  
    int day;  
};
```

```
// new  
Date inFutureCpp20 { .year = 2050, .month = 4, .day = 10 };  
// old  
Date inFutureOld { 2050, 4, 10 };
```

```
struct Date {  
    int year;  
    int month;  
    int day;
```

```
    static int mode;  
};
```

```
Date d { .mode = 10 }; // error, mode is static!  
Date d { .day = 1, .year = 2010 }; // error, out of order!  
Date d { 2050, .month = 12 }; // error, mix!
```

Default data member initialization, C++11

```
struct CarInfo {  
    std::string name { "unknown" };  
    unsigned year { 1920 };  
    unsigned seats { 4 };  
    double power { 100. };  
};  
  
void printInfo(const CarInfo& c) { /* */ }  
  
int main() {  
    CarInfo unknown;  
    printInfo(unknown);  
    CarInfo partial{"large car", 1975};  
    printInfo(partial);  
}
```

output:

```
unknown, 1920 year, 4 seats, 100 hp  
large car, 1975 year, 4 seats, 100 hp
```

Constructors

```
class Product {
public:
    Product() : name_{"none"}, id_{-1} { }
```

```
private:
    int id_;
    std::string name_;
};
```

```
<source>: In constructor 'Product::Product()':
<source>:15:17: warning: 'Product::name_' will be initialized after [-Wreorder]
    15 |         std::string name_;
        |         ^~~~~~
<source>:14:9: warning:   'int Product::id_' [-Wreorder]
    14 |         int id_;
        |         ^~~
```

- In case of default initialization, default ctor is called
- In case of value initialization, default ctor is also called

{ } vs ()

```
struct Box { };
```

```
struct Product {  
    Product(): name{"default product"} { }  
    Product(const Box& b) : name{"box"}{ }  
    std::string name;  
};
```

```
int main() {  
    Product p();           // << 1.  
    std::cout << p.name;  
    Product p2(Box());    // << 2.  
    std::cout << p2.name;  
}
```

we can fix it:

```
Product p{};  
Product p1;  
Product p2{Box()};  
Product p3{Box{}};
```

{ } vs ()

The curly list initialization has the following advantages:

- *the syntax is similar to aggregate initialization,*
- *adds a way to initialize containers with a list of the objects. For example*
`std::vector<int> v { 1, 2, 3, 4 },`
- *allowing for a safer way of initialization that checks for narrowing. For example, `int v{10.3}` won't compile and reports a narrowing error, while `int v(10.3)` works and might produce an unwanted result.*

```
std::vector<int> vec1 { 1, 2 }; // holds two values, 1 and 2
std::vector<int> vec2 ( 1, 2 ); // holds one value, 2!
```

{ } vs ()

ES.23: Prefer the { } initializer syntax

Reason: Prefer { }. The rules for { } initialization are simpler, more general, less ambiguous, and safer than for other initialization forms. Use = only when you are sure there can be no narrowing conversions. For built-in arithmetic types, use = only with auto. Avoid () initialization, which allows parsing ambiguities.

The guideline also mentions some exceptions:

Exception: *For containers, there is a tradition for using { . . . } for a list of elements and (. . .) for sizes:*

```
vector<int> v(10); // 10 elements with the default value 0
vector<int> v2{10}; // vector of 1 element with the value 10
```

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-list>

explicit

```
struct Product {  
    Product()  
    : name{"default product"}  
    , value{}  
    { }  
    Product(int v)  
        : name{"basic"}  
        , value{v}  
    { }  
    Product(const std::string& n, int v)  
        : name{n}  
        , value{v}  
    { }  
  
    std::string name;  
    int value;  
};
```

```
Product numbers = 100.2;    // copy initialization  
Product box = {"a box", 1}; // copy list-initialization  
  
void printProduct(const Product& prod) {  
    std::cout << prod.name << ", " << prod.value << '\n';  
}  
  
int main() {  
    double someRandomNumber = 100.1;  
    printProduct(someRandomNumber);  
    printProduct({"a box", 2});  
}
```

better with explicit <https://godbolt.org/z/7Kab4eT5T>

direct vs copy

// copy:

```
int x = 42;           // a form of a copy initialization
```

```
void foo(int param) { }
```

```
foo(x);               // copy initialization is performed on the argument
```

```
int anotherFoo() { return 42; } // a copy initialization is done on the return value
```

```
struct Point { int x; int y; };
```

```
Point pt { 0, 1 };    // aggregate initialization
```

```
Point p2 = { 10, 11 }; // uses copy initialization for each element
```

// direct

```
int y {42};           // a form of a direct initialization
```

```
double z (42.2);      // direct with parens
```


direct vs copy

In summary:

- *Direct initialization behaves like a function call to an overloaded function:*
 - *The functions, in this case, are the constructors of the type (including explicit ones).*
 - *Overload resolution will find the best matching constructor and, when needed, will do any implicit conversion required.*
- *Copy initialization constructs an implicit conversion sequence:*
 - *It tries to convert arguments to an object of the given type.*
 - **Explicit constructors are not considered for copy initialization.**

https://en.cppreference.com/w/cpp/language/implicit_conversion

An expression `e` is said to be implicitly convertible to `T2` if and only if `T2` can be copy-initialized from `e`, that is the declaration `T2 t = e;` is well-formed (can be compiled), for some invented temporary `t`. Note that this is different from direct initialization (`T2 t(e)`), where explicit constructors and conversion functions would additionally be considered.

Putting all ctors together

<https://godbolt.org/z/zeY5sE57E> - adding logging

NSDMI - going back

```
int initA() { }  
std::string initB() { }  
struct SimpleType { };  
  
int main() {  
    std::cout << "SimpleType t10:\n";  
    SimpleType t0;  
    std::cout << "SimpleType t1(10):\n";  
    SimpleType t1("world");  
    std::cout << "SimpleType t2 = t1:\n";  
    SimpleType t2 = t1;  
}
```

<https://godbolt.org/z/dKqfsaTWT>

<https://www.cppstories.com/2015/02/non-static-data-members-initialization/>

NSDMI - more cases

```

struct S {
    int zero {}; // fine, value initialization
    int a = 10; // fine, copy initialization
    double b { 10.5 }; // fine, direct list initialization
    // short c ( 100 ); // err, direct initialization with parens
    int d { zero + a }; // dependency, risky, but fine
    // double e { *mem * 2.0 }; // undefined!
    int* mem = new int(d); // only for demo, use smart pointers...
    std::unique_ptr<int[]> pInts = std::make_unique<int[]>(10);
    long arr[4] = { 0, 1, 2, 3 };
    std::array<int, 4> moreNumbers { 10, 20, 30, 40 };
    // long arr2[] = { 1, 2 }; // cannot deduce
    // auto f = 1; // err, type deduction doesn't work
    double g { compute() };
    // int& ref { }; // error, cannot set ref to null!
    int& refOk { zero };

    ~S() { delete mem; }
    double compute() { return a*b; }
};

```

NSDMI - limitations

```
class Type {  
    static inline auto theMeaningOfLife = 42; // int deduced  
};
```

```
class Type {  
    auto myField { 0 }; // error  
    auto param { 10.5f }; // error  
};
```

```
class Type {  
    std::vector<int> ints { 1, 2, 3, 4, 5 }; // error!  
};
```

```
class DataPacket {  
    std::string data_ (40, '*'); // syntax error!  
    size_t checksum_ { calcChecksum(data_) };  
    size_t serverId_ { 404 };  
  
    /* rest of the code*/  
};
```

Non local objects

Storage duration	Explanation
<i>automatic</i>	<i>Automatic means that the storage is allocated at the start of the scope. Most local variables have automatic storage duration (except those declared as <code>`static`</code>, <code>`extern`</code>, or <code>`thread_local`</code>).</i>
<i>static</i>	<i>The storage for an object is allocated when the program begins (usually before the <code>`main()`</code> function starts) and deallocated when the program ends. There's only one instance of such an object in the whole program.</i>
<i>thread</i>	<i>The storage for an object is tied to a thread: it's started when a thread begins and is deallocated when the thread ends. Each thread has its own "copy" of that object.</i>
<i>dynamic</i>	<i>The storage for an object is allocated and deallocated using explicit dynamic memory allocation functions. For example, by the call to <code>`new`</code>/<code>`delete`</code>.</i>

static

```
#include <iostream>
```

```
struct Value {  
    Value(int x) : v(x) { std::cout << "Value(" << v << ")\n"; }  
    ~Value() noexcept { std::cout << "~Value(" << v << ")\n"; }  
  
    int v {0};  
};  
  
Value v{42};  
  
int main() {  
    puts("main starts...");  
    Value x { 100 };  
    puts("main ends...");  
}
```

Output

```
Value(42)  
main starts...  
Value(100)  
main ends...  
~Value(100)  
~Value(42)
```

Static initialization order fiasco

<https://wandbox.org/permlink/h19Hkk8qdlU2PwLS>

Fixed with `constinit` (C++20):

<https://wandbox.org/permlink/atGGtco0AJd5eRyx>

inline variables

```
template <typename Derived>                                     https://godbolt.org/z/Pcs13d17v
class InstanceCounter {
    static inline size_t counter_ { 0 };

public:
    InstanceCounter() noexcept { ++counter_; }
    InstanceCounter(const InstanceCounter& ) noexcept { ++counter_; }
    InstanceCounter(InstanceCounter&& ) noexcept { ++counter_; }
    ~InstanceCounter() noexcept { --counter_; }

    static size_t GetInstanceCounter() { return counter_; }
};

struct Value : InstanceCounter<Value> {
    int val { 0 };
};

struct Wrapper : InstanceCounter<Wrapper> {
    double val { 0.0 };
};
```

(Tricky) cases

```
constinit int x = 10;  
constinit int y = x; // does it compile?
```

(Tricky) cases

```
struct S {  
    int a { 10 };  
    int b { 42 };  
};  
S s { 1 };  
std::cout << s.a << ", " << s.b;
```

1. 1, 0
2. 10, 42
3. 1, 42

(Tricky) cases

```
struct Number {  
    Number(int n) { }  
};
```

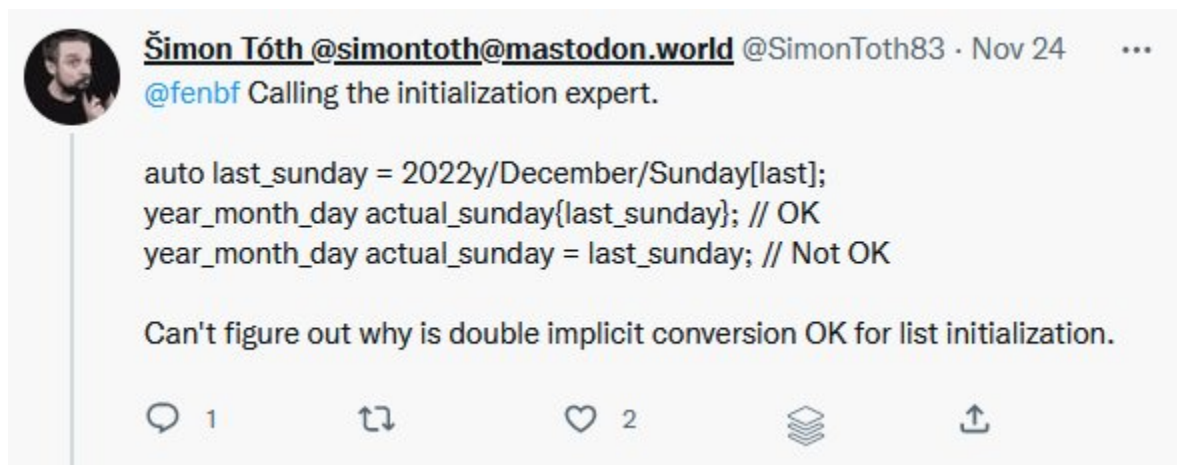
```
struct Special {  
    Special(Number num) {}  
};
```

```
Special spec1 { 42 };  
Special spec2 = 42; // doesn't compile!
```

<https://godbolt.org/z/xaq3Ya8qY>
<https://cppinsights.io/s/fac160a8>

This one doesn't compile, because the compiler would have to first convert integer into `Number` and then `Number` into `Special` (copy initialization).

(Tricky) cases



```
#include <chrono>
```

```
int main() {  
    using namespace std::chrono;
```

```
    auto last_sunday = 2022y/December/Sunday[last];
```

```
    year_month_day actual_sunday{last_sunday}; // OK
```

```
    //year_month_day actual_sunday = last_sunday; // Not OK
```

```
}
```

Summary

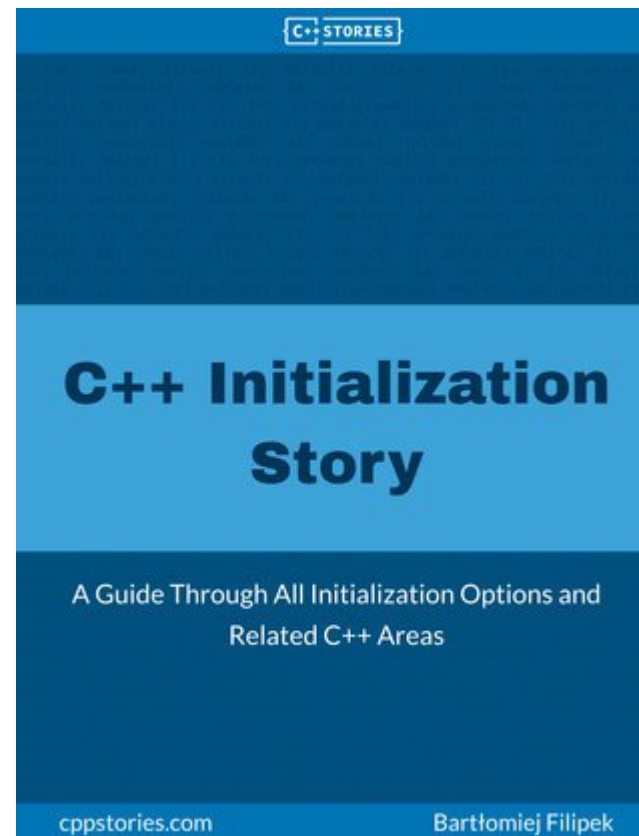
“How to stay sane?”

Additional guides

- *In Item 7 for Effective Modern C++, Scott Meyers said that “braced initialization is the most widely usable initialization syntax, it prevents narrowing conversions, and it’s immune to C++’s most vexing parse.*
- *Nicolai Josuttis had an excellent presentation about all corner cases: [CppCon 2018: Nicolai Josuttis “The Nightmare of Initialization in C++” - YouTube](#), and suggests using {}*
- *Only [abseil / Tip of the Week #88: Initialization: =, \(\), and {}](#) - prefers the old style. This guideline was updated in 2015, so many things were updated as of C++17 and C++20.*
- *In [Core C++ 2019 :: Timur Doumler :: Initialisation in modern C++ - YouTube](#) - Timur suggests {} for all, but if you want to be sure about the constructor being called then use (). As () performs regular overload resolution.*

More

- *thread_local*
- *techniques*
- *lazy initialization*
- *initializer_list*
- *default and deleted constructors* - <https://godbolt.org/z/6adcddf4q>
- *compiler generated constructors and special member functions*
- *and more!*



<https://leanpub.com/cppinitbook/>