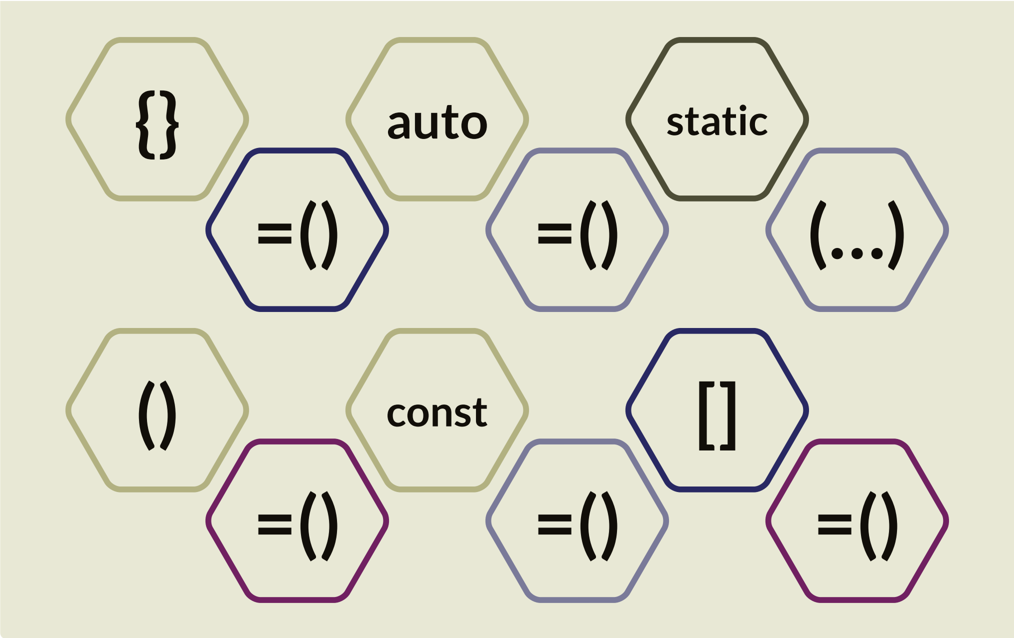
C++ STORIES



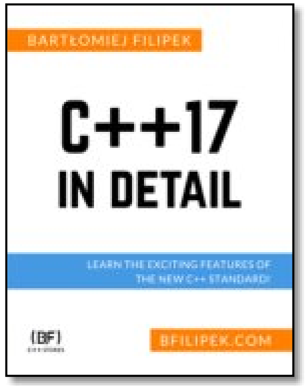
Staying Sane With C++ Initialization Rules

Down the Rabbit Hole...

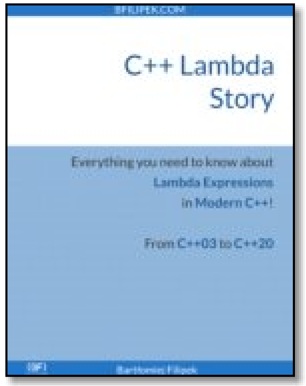
Bartłomiej Filipek, 1st December 2022, cppstories.com

About Me

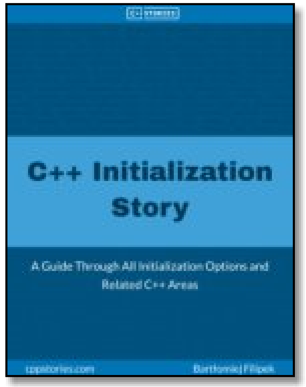
- *Author of [cppstories.com]*
- *~15y professional coding experience*
- *4x Microsoft MVP, since 2018*
- *C++ ISO Member*
- *[@Xara.com] since 2014*
  - *Mostly text related features for advanced document editors*
- *Somehow addicted to C++ ☺*



C++17 In Detail



C++ Lambda Story



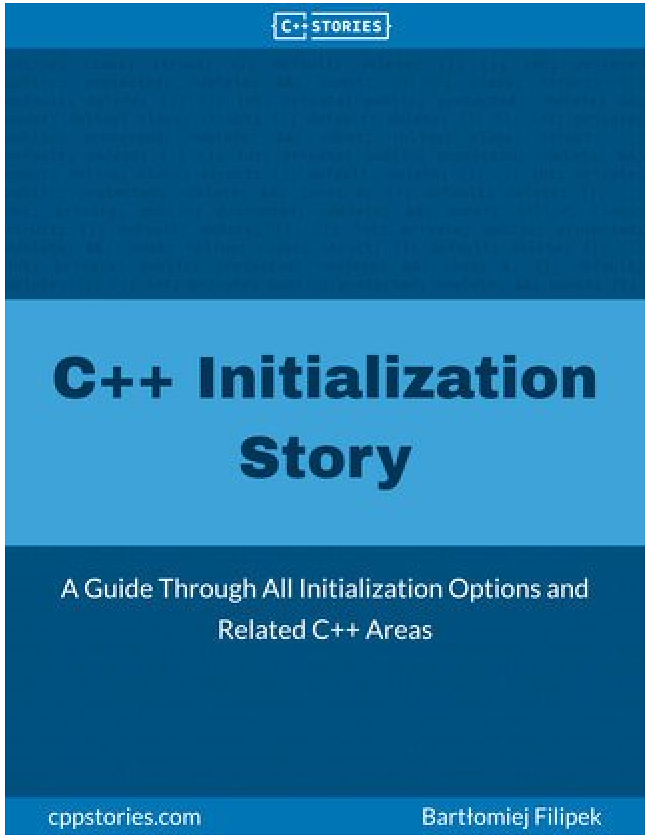
C++ Initialization Story

Xara Cloud Demo

Microsoft® Most Valuable Professional

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
```

Setting values to zero - Value initialization

P2723R0 - Zero-initialize objects of automatic storage duration

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 ()

```cpp
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:

```cpp
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

```cpp
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;
};
```

```cpp
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

```cpp
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 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
automatic	*Automatic means that the storage is allocated at the start of the scope. Most local variables have automatic storage duration (except those declared as `static`, `extern`, or `thread_local`).*
static	*The storage for an object is allocated when the program begins (usually before the `main()` function starts) and deallocated when the program ends. There's only one instance of such an object in the whole program.*
thread	*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.*
dynamic	*The storage for an object is allocated and deallocated using explicit dynamic memory allocation functions. For example, by the call to `new`/`delete`.*

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/atGGtcoOAJd5eRyx

inline variables

https://godbolt.org/z/Pcs13d17v

```
template <typename Derived>
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
```

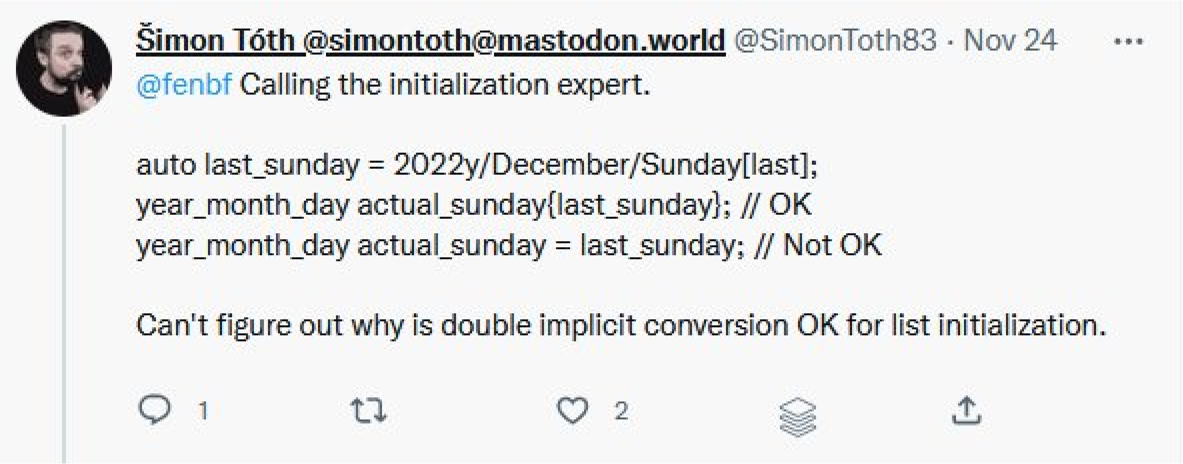
(Tricky) cases

```
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


Šimon Tóth @simontoth@mastodon.world @SimonToth83 · Nov 24

@fenbf Calling the initialization expert.

auto last_sunday = 2022y/December/Sunday[last];
year_month_day actual_sunday{last_sunday}; // OK
year_month_day actual_sunday = last_sunday; // Not OK

Can't figure out why is double implicit conversion OK for list initialization.

1 2

```
#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

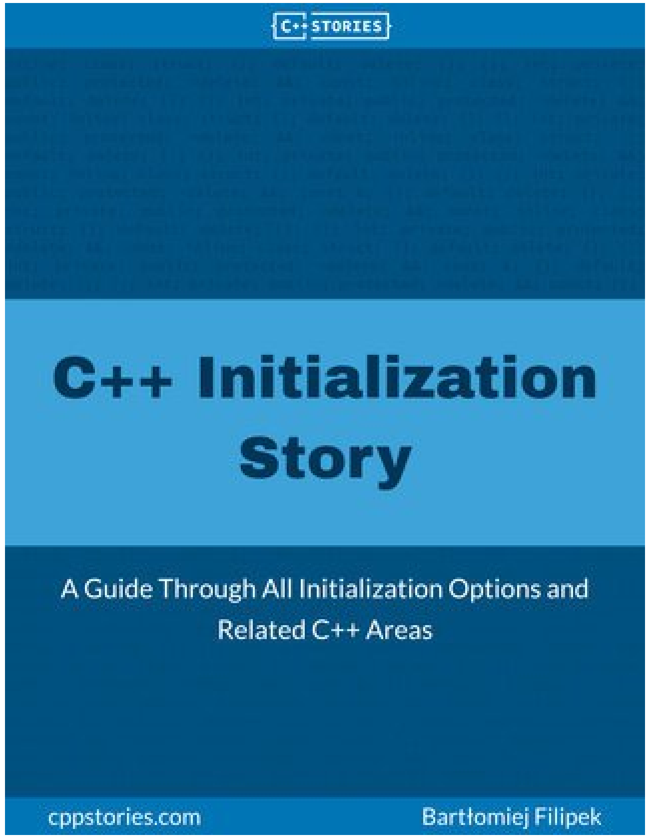
“How to stay sane?”

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/