Initialisation in modern C++

Version 1.1

Timur Doumler

@timur_audio

CppOnSea
5 February 2019

https://imgur.com/3wlxtI0

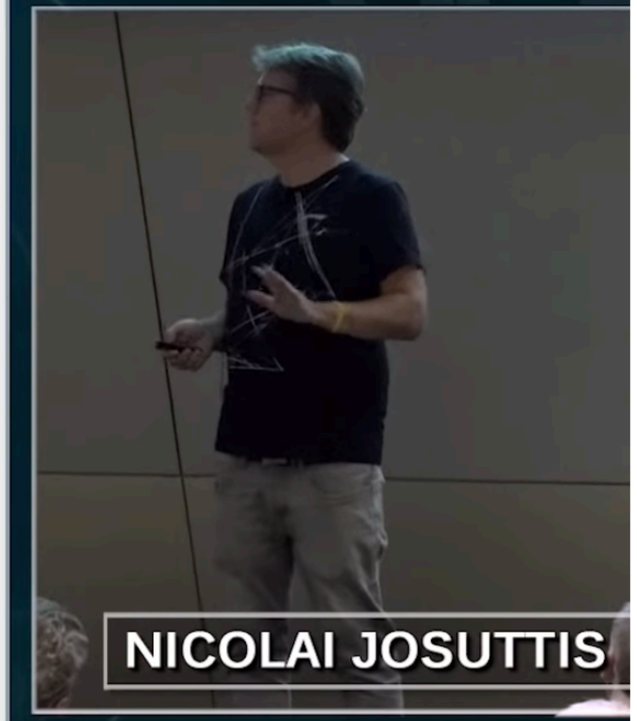
Consequences of Uniform Initialization

- **Couple of ways to initialize an int:**

```
int i1;                 // undefined value
int i2 = 42;            // note: inits with 42
int i3(42);             // inits with 42
int i4 = int();         // inits with 0
int i5{42};             // inits with 42
int i7{};               // inits with 0
int i6 = {42};          // inits with 42
int i8 = {};            // inits with 0
auto i9 = 42;           // inits int with 42
auto i10{42};           // C++11: std::initializer_list<int> , C++14: int
auto i11 = {42};        // inits std::initializer_list<int> with 42
auto i12 = int{42};     // inits int with 42
int i13();              // declares a function
int i14(7, 9);          // compile-time error
int i15 = (7, 9);       // OK, inits int with 9 (comma operator)
int i16 = int(7, 9);    // compile-time error
auto i17(7, 9);         // compile-time error
auto i18 = (7, 9);      // OK, inits int with 9 (comma operator)
auto i19 = int(7, 9);   // compile-time error
```

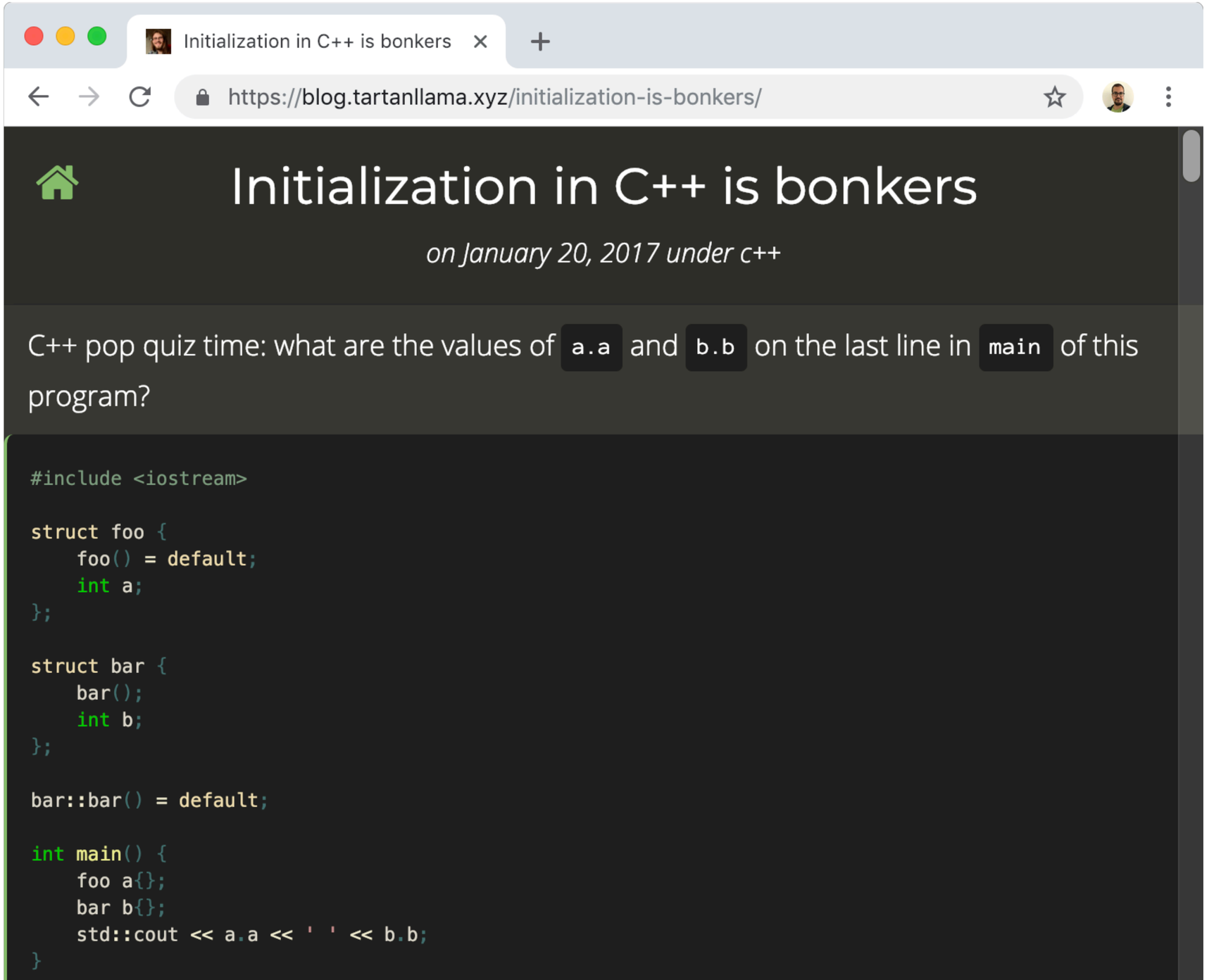

©2018 by IT-communication.com

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON



The Nightmare of Initialization in C++

CppCon.org



Initialization in C++ is bonkers

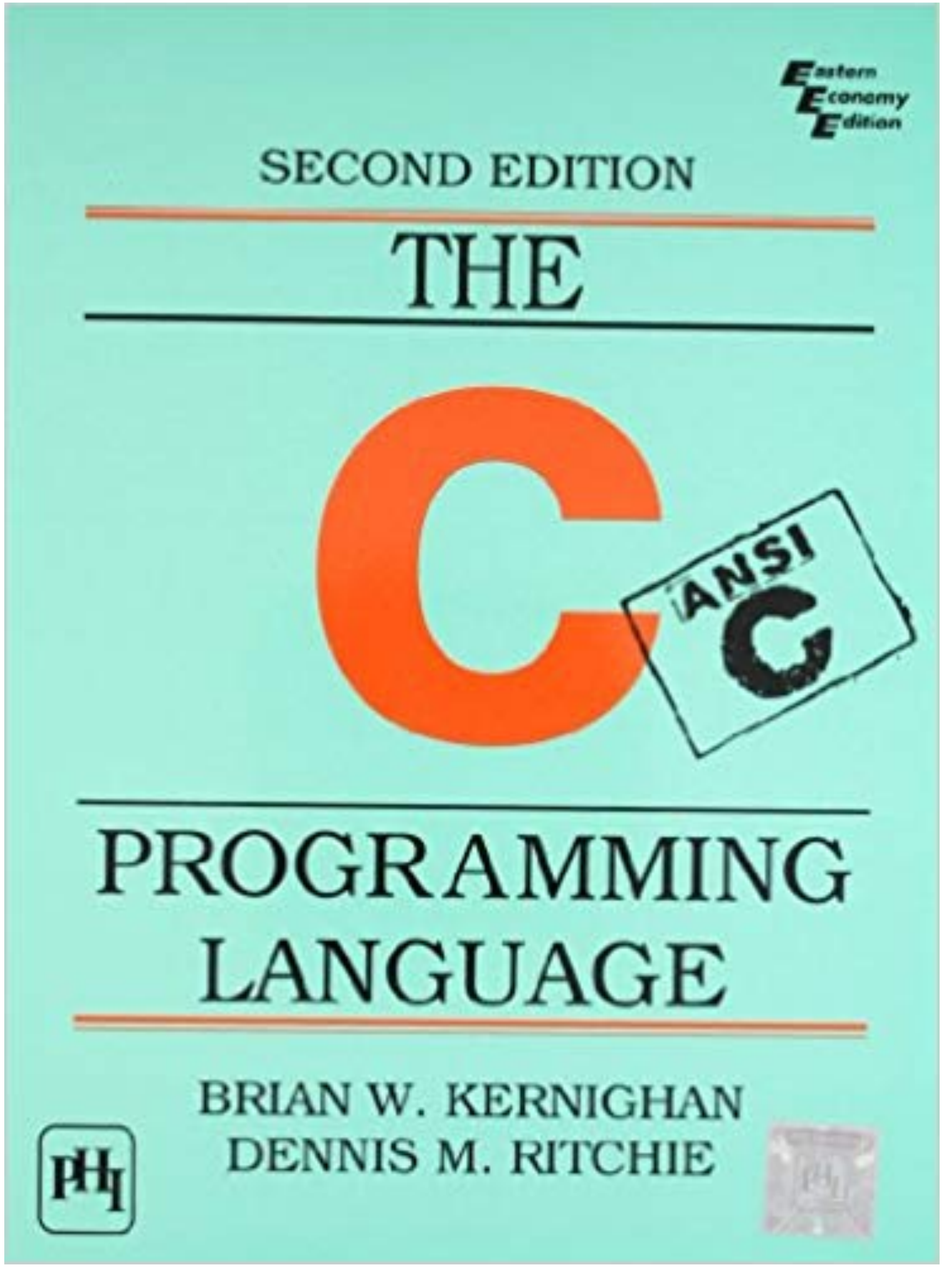
on January 20, 2017 under c++

C++ pop quiz time: what are the values of `a.a` and `b.b` on the last line in `main` of this program?

```
#include <iostream>

struct foo {
    foo() = default;
    int a;
};

struct bar {
    bar();
    int b;
};

bar::bar() = default;

int main() {
    foo a{};
    bar b{};
    std::cout << a.a << ' ' << b.b;
}
```

This talk

- Different ways to initialise an object in C++
  - In order of introduction: C, C++98, C++03, C++11, C++14, C++17
- The future: C++20
- Recommendations
- Overview table (updated!)


Eastern Economy Edition
SECOND EDITION
THE
C
ANSI C
PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PHI

Default initialisation

```
int main() {
  int i;
}
```

Default initialisation

```
int main() {
  int i;
  return i;   // Undefined behaviour!
}
```

Default initialisation

```
struct Foo {
  int i;
  int j;
};

int main() {
  Foo foo;
  return foo.i;   // Undefined behaviour!

}
```

Default initialisation

```
class Foo {
public:
  Foo() {}
  int get_i() const noexcept { return i; }
  int get_j() const noexcept { return j; }

private:
  int i;
  int j;
};

int main() {
  Foo foo;
  return foo.get_i();  // Undefined behaviour!
}
```

C++98: member initialiser list

```
class Foo {
public:
  Foo() : i(0), j(0) {}  // member initialiser list
  int get_i() const noexcept { return i; }
  int get_j() const noexcept { return j; }

private:
  int i;
  int j;
};

int main() {
  Foo foo;
  return foo.get_i();
}
```

C++11: default member initialisers

```
class Foo {
public:
  Foo() {}
  int get_i() const noexcept { return i; }
  int get_j() const noexcept { return j; }

private:
  int i = 0;   // default member initialisers
  int j = 0;
};

int main() {
  Foo foo;
  return foo.get_i();
}
```

Copy initialisation

```
int main() {
  int i = 2;
}
```

Copy initialisation

```
int main() {
  int i = 2;
}

int square(int i) {
  return i * i;
}
```

Copy initialisation

```
int main() {
  int i = 2;
}

int square(int i) {
  return i * i;
}
```

- Initialiser starting with `=`, or
- Passing argument by value, or
- Returning by value

Copy initialisation

```
int main() {
  int i = 2;
}

int square(int i) {
  return i * i;
}
```

- Initialiser starting with `=`, or
- Passing argument by value, or
- Returning by value

- Copy init is never an assignment

Copy initialisation

```
int main() {
  int i = 2;
}

int square(int i) {
  return i * i;
}
```

- Initialiser starting with `=`, or
- Passing argument by value, or
- Returning by value

- Copy init is never an assignment
- If types don’t match, copy init performs a *conversion sequence*

Aggregate initialisation

```
int i[4] = {0, 1, 2, 3};
```

Aggregate initialisation

```
int i[4] = {0, 1, 2, 3};
int j[]  = {0, 1, 2, 3};  // array size deduction
```

Aggregate initialisation

```
int i[4] = {0, 1, 2, 3};
int j[]  = {0, 1, 2, 3};  // array size deduction

struct Foo {   // aggregate type
  int i;
  float j;
};

Foo foo = {1, 3.14159};
```

Aggregate initialisation

```
int i[4] = {0, 1, 2, 3};
int j[]  = {0, 1, 2, 3};  // array size deduction

struct Foo {   // aggregate type
  int i;
  float j;
};

Foo foo = {1, 3.14159};   // foo is aggregate-initialised;
                          // foo.i and foo.j are copy-initialised
```

Zero initialisation of aggregate elements

```
struct Foo {
  int i;
  int j;
};

int main() {
  Foo foo = {1};
  return foo.j;
}
```

Zero initialisation of aggregate elements

```
struct Foo {
  int i;
  int j;
};

int main() {
  Foo foo = {1};     // elements with no initialiser are zero-initialised!
  return foo.j;      // OK, returns 0
}
```

Zero initialisation of aggregate elements

```
struct Foo {
  int i;
  int j;
};

int main() {
  Foo foo = {1};     // elements with no initialiser are zero-initialised!
  return foo.j;      // OK, returns 0
}

int arr[100] = {};     // all elements are zero-initialised!
```

Brace elision

```
struct Foo {
  int i;
  int j;
};

struct Bar {
  Foo f;
  int k;
};

int main() {
  Bar b = {1, 2};
  return b.k;       // What does this return?
}
```

Brace elision

```
struct Foo {
  int i;
  int j;
};

struct Bar {
  Foo f;
  int k;
};

int main() {
  Bar b = {1, 2};  // Equivalent to Bar b = {{1, 2}, 0};
  return b.k;      // returns 0!
}
```

Static initialisation

```
static int i = 3;    // Constant initialisation
```

Static initialisation

```
static int i = 3;    // Constant initialisation
static int j;        // Zero-initialisation
```

Static initialisation

```
static int i = 3;   // Constant initialisation
static int j;       // Zero-initialisation


int main()
{
    return i + j;  // OK, returns 3
}
```

Initialisation order fiasco

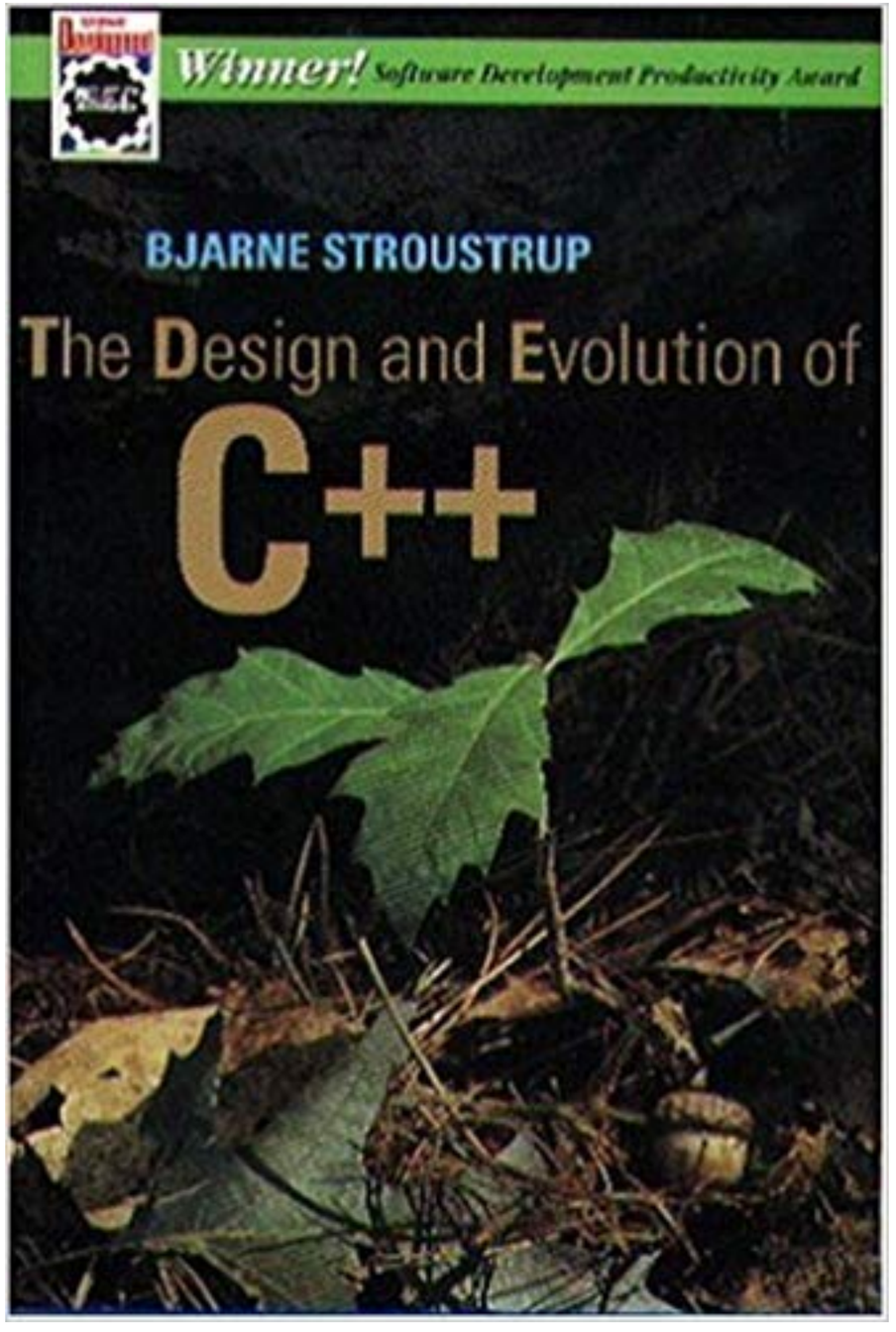
```
static Colour red = {255, 0, 0};  // Uh-oh :(
```

Initialisation order fiasco

```
static Colour red = {255, 0, 0};  // if constructor is constexpr (C++11)
                                  // -> constant initialisation :)
```

What have we got so far?

- **Default initialisation** (no initialiser)
  - built-in types: uninitialised, UB on access
  - class types: default constructor
- **Copy initialisation** (`= value`, pass-by-value, return-by-value)
- **Aggregate initialisation** (`= {args}`)
  - Elements without initialisers undergo **zero initialisation**
- **Static initialisation**
  - **zero-initialisation** by default
  - **constant initialisation** (`= constexpr`)


Winner! Software Development Productivity Award
BJARNE STROUSTRUP
The Design and Evolution of
C++

```
Foo foo(1, 2);    // C++ introduces constructors!
```

```
Foo foo(1, 2);
int i(3);
```

Direct initialisation

```
Foo foo(1, 2);
int i(3);
```

Direct initialisation

```
Foo foo(1, 2);
int i(3);
```

“whenever the initialiser is an argument list in parens”

Direct initialisation

```
Foo foo(1, 2);
int i(3);
```

- Differences to copy initialisation:
  - For built-in types: no difference
  - For class types:
    - Can take more than one argument
    - Does not perform “conversion sequence”,

      instead just calls constructor using normal overload resolution

Direct initialisation

```
struct Foo {
  explicit Foo(int) {}
};

Foo foo1 = 1;    // ERROR
Foo foo2(2);     // ok
```

Direct initialisation

```
struct Foo {
  explicit Foo(int) {}
  Foo(double) {}
};

Foo foo1 = 1;   // calls Foo(double)
Foo foo2(2);    // calls Foo(int)
```

Direct initialisation

```
Foo(1, 2);    // constructor call notation
auto* foo_ptr = new Foo(2, 3);    // new-expr with (args)
static_cast<Foo>(bar);    // casts
```

Problem: most vexing parse

```
struct Foo {};

struct Bar {
  Bar(Foo) {}
};

int main() {
  Bar bar(Foo());
}
```

Problem: most vexing parse

```
struct Foo {};

struct Bar {
  Bar(Foo) {}
};

int main() {
  Bar bar(Foo());    // This is a function declaration :(
}
```

What have we got so far?

- **Default initialisation** (no initialiser)
- **Copy initialisation** (`= value`, pass-by-value, return-by-value)
- **Aggregate initialisation** (`= {args}`)
- **Static initialisation**
- **Direct initialisation** (argument list in parens)
  - Problem: most vexing parse

03

Value initialisation

```
int main() {
  return int();
}
```

Value initialisation

```
int main() {
  return int();  // UB in C++98, OK since C++03
}
```

Value initialisation

```
int main() {
  return int();  // UB in C++98, OK since C++03
}
```

“whenever the initialiser is a pair of empty parens”

Value initialisation

When the initialiser is a pair of empty parens:

- If type has a *user-provided* default c'tor, it is called
- Otherwise, you get **zero initialisation**

Value initialisation

```
struct Foo {
  int i;
};

Foo get_foo() {
  return Foo();
}

int main() {
  return get_foo().i;
}
```

Value initialisation

```
struct Foo {
  int i;
};

Foo get_foo() {
  return Foo();  // Value initialisation
}

int main() {
  return get_foo().i;    // OK since C++03, returns 0
}
```

Value initialisation

```
struct Foo {
  Foo() {}   // user-provided ctor!
  int i;
};

Foo get_foo() {
  return Foo();  // Value initialisation
}

int main() {
  return get_foo().i;    // value is uninitialised -> UB!!!
}
```

Value initialisation

```
struct Foo {
  Foo() = default;   // (since C++11) user-defined, but not user-provided
  int i;
};

Foo get_foo() {
  return Foo();  // Value initialisation
}

int main() {
  return get_foo().i;    // OK, returns 0
}
```

Value initialisation

```
struct Foo {
  Foo();
  int i;
};

Foo::Foo() = default;  // out-of-line counts as user-provided!

Foo get_foo() {
  return Foo();  // Value initialisation
}

int main() {
  return get_foo().i;   // value is uninitialised -> UB!!!
}
```

What have we got so far?

- **Default initialisation** (no initialiser)
- **Copy initialisation** (`= value`, pass-by-value, return-by-value)
- **Aggregate initialisation** (`= {args}`)
- **Static initialisation**
- **Direct initialisation** (argument list in parens)
- **Value initialisation** (empty parens)
  - Performs default-init or zero-init
  - Problem: most vexing parse

++11

“Uniform initialisation”

- We’ve got too many different initialisation syntaxes
- Parens are vexing
- We cannot do:

```
std::vector<int> vec = {0, 1, 2, 3, 4};
```

- Instead we have to do:

```
std::vector<int> vec;
vec.reserve(5);
vec.push_back(0);
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
```

“Uniform initialisation”

- So let’s add one more initialisation syntax!

```
Foo foo{1, 2};
```

- It’s called **list-initialisation**
- Idea: it does “everything”

List initialisation

Direct-list-initialisation

```
Foo foo{1, 2};
```

Copy-list-initialisation

```
Foo foo = {1, 2};
```

List initialisation

Direct-list-initialisation

```
Foo foo{1, 2};
```

Copy-list-initialisation

```
Foo foo = {1, 2};
```

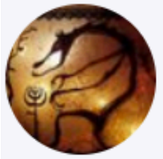
braced-init-list

How does this work?

```
std::vector<int> vec{0, 1, 2, 3, 4};
```

std::initializer_list

```
template <typename T>
class vector {
  // stuff...
  vector(std::initializer_list<T> init);  // init list ctor
};

std::vector<int> vec{0, 1, 2, 3, 4};  // calls that^
```



Shafik Yaghmour @shafikyaghmour · Nov 1

One of my favorite questions from @Cppcon 2018 Grill The Committee

“If you had a magic wand and allowed yourself to have some fun, what would you remove from C++?”

How committee members answered 👇

Also your turn w/ poll 👇

#CppCon #Cplusplus #Programming

18%	CTAD
31%	Volatile (or a lot of it)
39%	initializer_list
12%	Other, specify in reply

103 votes • Final results

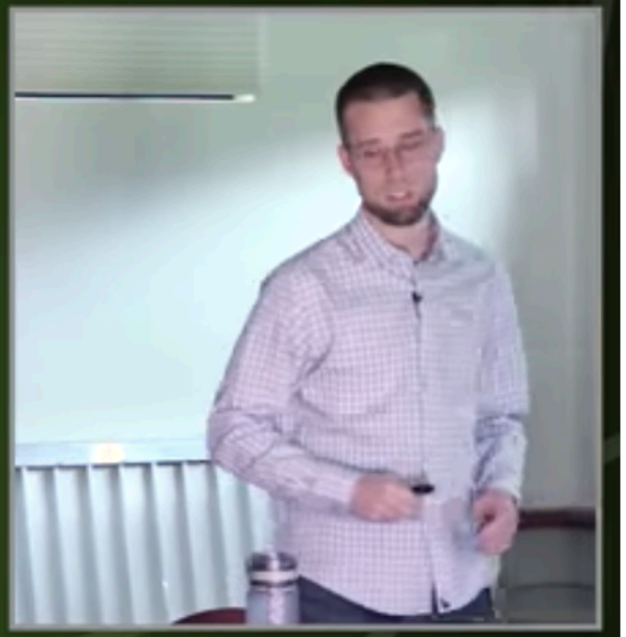
C++ now 2018 MAY 7 - 11 cppnow.org

initializer_lists Are Broken - Let's Fix Them

Copyright Jason Turner

@lefticus

1.1



Jason Turner

initializer_lists
are Broken
- Let's Fix Them

Video Sponsorship Provided By:

std::initializer_list

```
std::vector<int> v(3, 0);  // vector contains 0, 0, 0
std::vector<int> v{3, 0};  // vector contains 3, 0
```

std::initializer_list

```
std::string s(48, 'a');  // "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
std::string s{48, 'a'};
```

std::initializer_list

```
std::string s(48, 'a');  // "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
std::string s{48, 'a'};  // "0a"
```

std::initializer_list

```
template <typename T, size_t N>
auto test() {
  return std::vector<T>{N};
}

int main() {
  return test<std::string, 3>().size();   // what does this return??
}
```

List initialisation

- For aggregate types:
  - aggregate init
- For built-in types:
  - `{a}` is direct init, `= {a}` is copy-init
- For class types:
  - First, greedily try to call a ctor that takes a std::initializer_list
  - If there is none: direct-init

    (or copy-init if `= {a}` and a is a single element)

Empty braces {} are special

- For aggregate types: aggregate init (all elements zeroed)
- Only call std::initializer_list ctor if there is no default ctor

```
template <typename T>
struct Foo {
  Foo();
  Foo(std::initializer_list<T>);
};

int main() {
  Foo<int> foo{};   // calls default ctor!
}
```

Empty braces {} are special

- For aggregate types: aggregate init (all elements zeroed)
- Only call std::initializer_list ctor if there is no default ctor
- Otherwise: **value initialisation**

Empty braces {} are special

- For aggregate types: aggregate init (all elements zeroed)
- Only call std::initializer_list ctor if there is no default ctor
- Otherwise: **value initialisation**

```
struct Foo {
  Foo() = default;
  int i;
};

int main() {
  Foo foo{};   // value init -> zero init, no vexing parse!
  return foo.i;  // returns 0
}
```

Empty braces {} are special

- For aggregate types: aggregate init (all elements zeroed)
- Only call std::initializer_list ctor if there is no default ctor
- Otherwise: **value initialisation**

```
struct Foo {
  Foo() {}   // user-provided ctor!
  int i;
};

int main() {
  Foo foo{};   // value init -> default ctor gets called
  return  foo.i;  // uninitialised! UB
}
```

List init: no narrowing conversions!

```
int main() {
  int i{2.0};  // Error!
}
```

List init: nested braces

- The nice case:

```
std::map<std::string, int> my_map {{"abc", 0}, {"def", 1}};
```

List init: nested braces

- The nice case:

```
std::map<std::string, int> my_map {{"abc", 0}, {"def", 1}};
```

- The evil case:

```
std::vector<std::string> v1 {"abc", "def"};   // OK
std::vector<std::string> v2 {{"abc", "def"}}; // ???
```

List init: nested braces

- The nice case:

```
std::map<std::string, int> my_map {{"abc", 0}, {"def", 1}};
```

- The evil case:

```
std::vector<std::string> v1 {"abc", "def"};   // OK
std::vector<std::string> v2 {{"abc", "def"}}; // Undefined behaviour!!!
```

Copy list init

- The nice case:

```
std::map<std::string, int> my_map {{"abc", 0}, {"def", 1}};
```

- The evil case:

```
std::vector<std::string> v1 {"abc", "def"};   // OK
std::vector<std::string> v2 {{"abc", "def"}}; // Undefined behaviour!!!
```

Copy list init

```
Widget<int> f1()
{
    return {3, 0};  // copy-list init
}

void f2(Widget);
f2({3, 0});  // copy-list init
```

How does the number of braces affect uniform initialization?

21

7

Consider the following code snippet:

```
#include <iostream>

struct A {
  A() {}
  A(const A&) {}
};

struct B {
  B(const A&) {}
};

void f(const A&) { std::cout << "A" << std::endl; }
void f(const B&) { std::cout << "B" << std::endl; }

int main() {
  A a;
  f(   {a}   ); // A
  f(  {{a}}  ); // ambiguous
  f( {{{a}}} ); // B
  f({{{{a}}}}); // no matching function
}
```

Why does each call fabricate the corresponding output? How does the number of braces affect uniform initialization? And how does brace elision affect all this?

What have we got so far?

- **Default initialisation** (no initialiser)
- **Copy initialisation** (`= value`, pass-by-value, return-by-value)
- **Aggregate initialisation** (`= {args}`)
- **Static initialisation**
- **Direct initialisation** (argument list in parens)
- **Value initialisation** (empty parens)
- **List initialisation** (`{args}` is direct-list-init, `= {args}` is copy-list-init)
  - Performs aggregate-init or direct-init or copy-init or value-init
  - Problems with std::initializer_list, useless in templates

14

Fix #1: aggregates can have DMIs

```
struct Foo {
  int i = 0;
  int j = 0;
};

Foo foo{1, 2};  // OK since C++14
```

Fix #2: auto + list-initialisation

```cpp
int i = 3;     // int
int i(3);      // int
int i{3};      // int
int i = {3};   // int

auto i = 3;    // int
auto i(3);     // int
auto i{3};     // std::initializer_list<int> in C++11
auto i = {3}; // std::initializer_list<int> in C++11
```

Fix #2: auto + list-initialisation

```cpp
int i = 3;     // int
int i(3);      // int
int i{3};      // int
int i = {3};   // int

auto i = 3;    // int
auto i(3);     // int
auto i{3};     // int since C++14, ill-formed if more than one initialiser
auto i = {3}; // std::initializer_list<int> (always)
```

17

Guaranteed copy elision

```
auto num = 1;
auto foo = Foo{2, 3};
```

SUTTER'S MILL

Herb Sutter on software development

« Recommended reading: Why mobile web apps are slow (Drew Crawford)

GotW #7a: Minimizing Compile-Time Dependencies, Part 1 »

GotW #94 Solution: AAA Style (Almost Always Auto)

2013-08-12 by Herb Sutter

Toward correct-by-default, efficient-by-default, and pitfall-free-by-default variable declarations, using "AAA style"... where "triple-A" is both a mnemonic and an evaluation of its value.

Guaranteed copy elision

```cpp
auto foo = std::atomic<int>{0};  // C++11/14: Error
                                 // atomic is neither copyable nor movable
```

Guaranteed copy elision

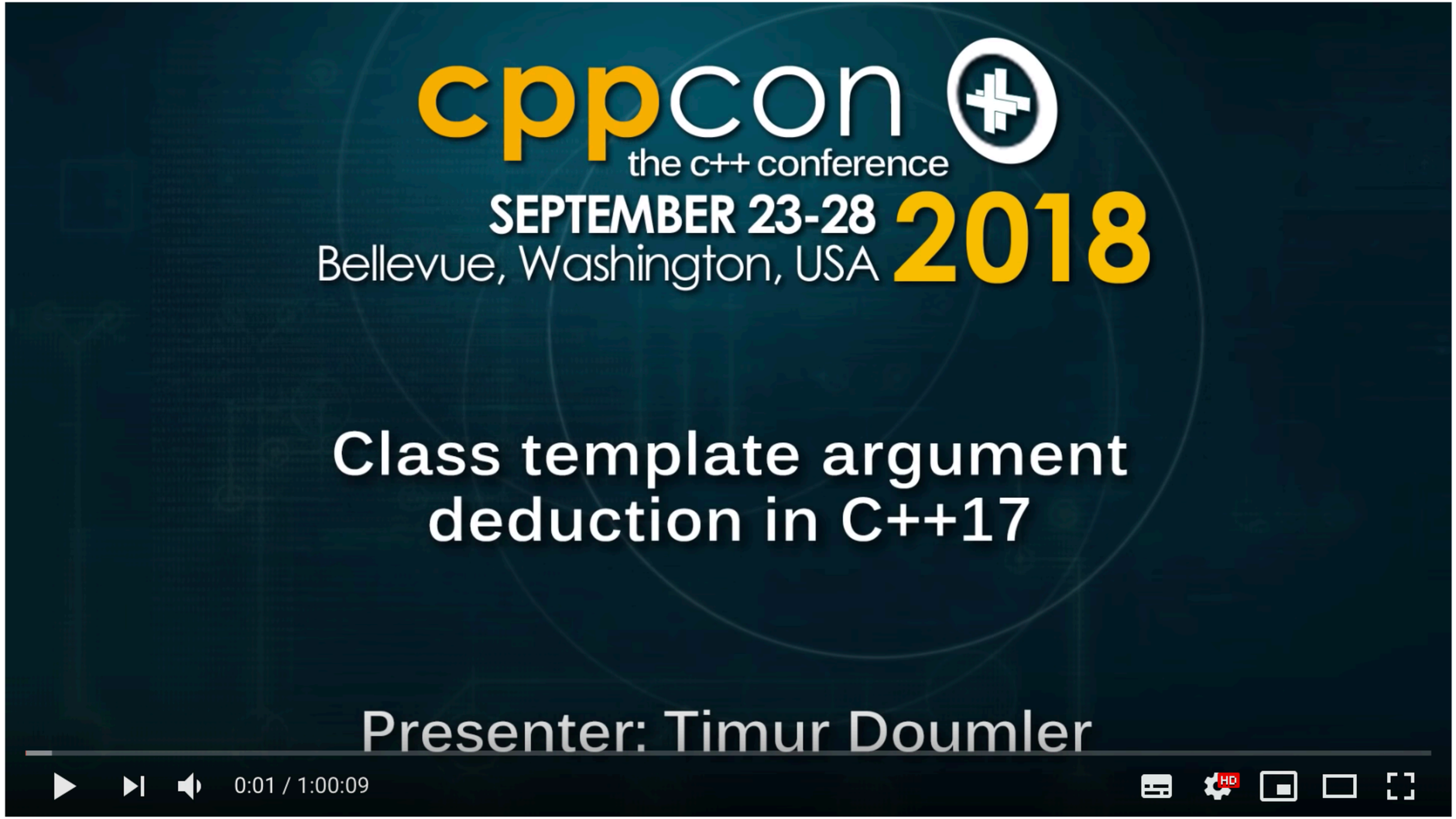
```cpp
auto foo = std::atomic<int>{0};  // Works in C++17 :)
```

Guaranteed copy elision

```cpp
auto foo = std::atomic<int>{0};  // Works in C++17 :)
```

“Almost always auto” is now “Always auto” !! :)

Initialisation and CTAD



CppCon 2018: Timur Doumler “Class template argument deduction in C++17”


C++
20

Designated initialisation

```
struct Foo {
  int a;
  int b;
  int c;
};

int main() {
  Foo foo{.a = 3, .c = 7};
}
```

Designated initialisation

```
struct Foo {
  int a;
  int b;
  int c;
};

int main() {
  Foo foo{.a = 3, .c = 7};
}
```

Only for aggregate types.

C compatibility feature.

Works like in C99, except:

- not out-of-order

```
Foo foo{.c = 7, .a = 3} // Error
```

- not nested

```
Foo foo{.c.e = 7} // Error
```

- not mixed with regular initialisers

```
Foo foo{.a = 3, 7} // Error
```

- not with arrays

```
int arr[3]{.[1] = 7} // Error
```

Array size deduction in new-expressions

http://wg21.link/p1009

```
double a[]{1,2,3};    // OK
double* p = new double[]{1,2,3}; // Error in C++17, will be OK in C++20
```

Aggregates can no longer declare constructors

http://wg21.link/p1008

```
struct Foo {
  Foo() = delete;
  int i;
  int j;
};

Foo foo1;    // Error
Foo foo2{};  // OK in C++17! Will be error in C++20
```

Problems with list init:

- Difficult to see when it'll call a std::initializer_list constructor, and when it won't
- std::initializer_list doesn't work with move-only types
- Useless in templates

  (you can't write a make_unique that works for aggregates!)
- Does not work with macros at all:

```
assert(Foo{2, 3});   // This breaks the preprocessor :(
```

Aggregate initialisation from parens

http://wg21.link/p0960

```
struct Foo {
  int i;
  int j;
};

Foo foo(1, 2);   // will work in C++20!
```

Aggregate initialisation from parens

http://wg21.link/p0960

```
struct Foo {
  int i;
  int j;
};

Foo foo(1, 2);   // will work in C++20!
int arr[3](0, 1, 2);  // will work in C++20!
```

Aggregate initialisation from parens

http://wg21.link/p0960

```
struct Foo {
  int i;
  int j;
};

Foo foo(1, 2);   // will work in C++20!
int arr[3](0, 1, 2);  // will work in C++20!
```

Idea: in C++20, `( )` and `{}` will do the same thing!

Except:

- `( )` does not call std::initializer_list constructors
- `{}` does not consider narrowing conversions

Recommendations:

- Use `auto`
- Use direct member initialisers (DMIs)
- Use `` `= value` `` for `int` and other simple value types
- Use `` `= {args}` `` for aggregate-init, std::initializer_list, DMIs

  → Recommendation for aggregates might change for C++20!)
- Use `` `{}` `` for value-init
- Use `` `(args)` `` to call constructors that take arguments

  → This is the controversial one. Other people say: use `` `{args}` ``

Initialisation in C++17

Version 2 – Copyright (c) 2019 Timur Doumler

`Type var`	**Default init** `;`	**Copy init** `= value;`	**Direct init** `(args);`	**Value init** `();`	**Empty braces** `{};  = {};`	**Direct list init** `{args};`	**Copy list init** `= {args};`
Built-in types	Uninitialised. Variables w/ static storage duration: Zero-initialised	Initialised with value (via conversion sequence)	1 arg: Init with arg >1 arg: Doesn't compile	Zero-initialised	Zero-initialised	1 arg: Init with arg >1 arg: Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile
auto	Doesn't compile	Initialised with value	Initialised with value	Doesn't compile	Doesn't compile	1 arg: Init with arg >1 arg: Doesn't compile	Object of type std::initializer_list
Aggregates	Uninitialised. Variables w/ static storage duration: Zero-initialised***	Doesn't compile	Doesn't compile (but will in C++20)	Zero-initialised***	Aggregate init**	1 arg: implicit copy/move ctor if possible. Otherwise aggregate init**	1 arg: implicit copy/ move ctor if possible. Otherwise aggregate init**
Types with std::initializer_list ctor	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor if there is one, otherwise std::initializer_list ctor	std::initializer_list ctor if possible, otherwise matching ctor	std::initializer_list ctor if possible, otherwise matching ctor****
Other types with no user-provided* default ctor	Members are default-initialised	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Zero-initialised***	Zero-initialised***	Matching ctor	Matching ctor****
Other types	Default ctor	Matching ctor (via conversion sequence), explicit ctors not considered	Matching ctor	Default ctor	Default ctor	Matching ctor	Matching ctor****

*not user-provided = not user-declared, or user-declared as `=default` *inside* the class definition

**Aggregate init copy-inits all elements with given initialiser, or value-inits them if no initialiser given

***Zero initialisation zero-initialises all elements *and initialises all padding to zero bits*

****Copy-list-initialisation considers explicit ctors, too, but doesn't compile if such a ctor is selected


Thank you!
@timur_audio
includecpp.org