MeetingC++ 2018

50 Shades of C++

Nicolai Josuttis
@NicoJosuttis

A Trivial Class

```
class Customer {
  private:
    std::string name;
  public:
    Customer (const std::string& n) : name(n) {
      assert(n.size() > 0);
    }
    void setName(const std::string& n) {
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

A Customer that must have a name

```
Customer c("Nico");
c.setName(c.getName() + "lai");
std::cout << "c: " << c << '\n';
```

Output:

```
c: Nicolai
```

Initialization

Uniform Initialization

```
int i1;                                  // undefined value
int i2 = 42;                             // note: initializes with 42
int i3(42);                              // initializes with 42
int i4 = int();                          // initializes with 0
int i5{42};                              // initializes with 42
int i7{};                                // initializes with 0
int i6 = {42};                           // initializes with 42
int i8 = {};                             // initializes with 0

int vals1[] = { 1, 2, 3 };               // initialization of aggregates
int vals2[] { 1, 2, 3 };                 // initialization of aggregates

std::complex<double> c1(4.0,3.0);        // initialization of classes
std::complex<double> c2{4.0,3.0};        // initialization of classes
std::complex<double> c3 = {4.0,3.0};     // initialization of classes

std::vector<std::string> cities{"Berlin", "Rome"};
std::vector<int> vals2{0, 8, 15, i1+i2};
std::vector<int> vals2 = {i1, i2, 42};
```

How 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};              // ???
auto i11 = {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
```


How 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};              // ???
auto i11 = {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 (com
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
```

don't use ( ) in initializations

Dealing with auto

```
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>
```

Fixed even in C++11 mode with:
- clang 3.8
- g++ 5
- Visual Studio 2015

Dealing with auto

```
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};           // inits int with 42 (unless "old" compiler)
```

Fixed even in C++11 mode with:
- clang 3.8
- g++ 5
- Visual Studio 2015

Dealing with auto

```
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};           // inits int with 42 (unless "old" compiler)
auto i11 = {42};        // still inits std::initializer_list<int> with 42
```

An = in initialization might change the type

Dealing with auto

```
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};           // inits int with 42 (unless "old" compiler)
auto i11 = {42};        // still inits std::initializer_list<int> with 42

std::initializer_list<auto> l = {42};   // OK since C++23 ???
std::initializer_list l = {42};         // OK since C++23 ???
```

don't use = in initializations

Almost Always Auto

```
int i = 42;                          auto i = 42;
long v = 42;                         auto v = 42l;
Customer c{"Jim", 77};               auto c = Customer{"Jim", 77};
vector<int>::const_iterator p        auto p = v.cbegin();
  = v.begin();

int i = {42};                        auto i = {42};     // initializer_list<int>
                                     auto i = int{42}; // OK

                                     using namespace std::literals;
std::string x = "42";                auto x = "42"s;    // since C++14

std::atomic<int> a{9};               auto a = std::atomic<int>{9}; // OK since C++17
std::array<int,5> r{};               auto r = std::array{};        // fast since C++17

long long ll{getInt()};              auto ll = long long{getInt()};  // ERROR
                                     auto ll = int64_t(getInt());    // different
                                     auto ll = static_cast<long long>(getInt());

const C& r = f();                    auto r = static_const<const C&>(f());    // NO
                                     auto& r = static_const<const C&>(f());   // OK
                                     auto&& r = static_const<const C&>(f()); // OK
```


AAAA (~~Almost~~ Always Auto&&)

```
int i = 42;                          auto i = 42;
long v = 42;                         auto v = 42l;
Customer c{"Jim", 77};               auto c = Customer{"Jim", 77};
vector<int>::const_iterator p        auto p = v.cbegin();
  = v.begin();

int i = {42};                        auto i = {42};     // initializer_list<int>
                                     auto i = int{42}; // OK

                                     using namespace std::literals;
std::string x = "42";                auto x = "42"s;    // since C++14

std::atomic<int> a{9};               auto a = std::atomic<int>{9}; // OK since C++17
std::array<int,5> r{};               auto r = std::array{};        // fast since C++17

long long ll{getInt()};              auto ll = long long{getInt()};  // ERROR
                                     auto ll = int64_t(getInt());    // different
                                     auto ll = static_cast<long long>(getInt());

const C& r = f();                    auto r = static_const<const C&>(f());    // NO
                                     auto& r = static_const<const C&>(f());   // OK
                                     auto&& r = static_const<const C&>(f()); // OK
```

The New Way of Initialization

```
int i{42};

std::array<int,4> arr{0, i, i+10, i+20};

for (auto pos{arr.cbegin()}; pos < arr.cend(); pos += 2) {
  std::cout << *pos << '\n';
}


class Derived : Base {
  public:
    Derived(int val) : Base{val} {
    }
};

for (int i{0}; i < 32;  ++i) {
  std::cout << i << '\n';
}
```

By default, prefer direct uniform initialization

- uniform: **with {…}**
- direct: **without =**

A Trivial Class: Initializations

```
class Customer {
  private:
    std::string name;
  public:
    Customer (const std::string& n) : name{n} {
      assert(n.size() > 0);
    }
    void setName(const std::string& n) {
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

```
Customer c{"Nico"};
c.setName(c.getName() + "lai");
std::cout << "c: " << c << '\n';
```

Output:

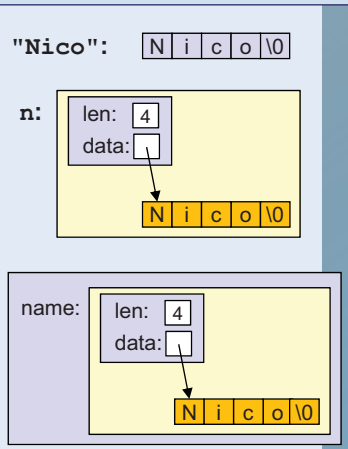
```
c: Nicolai
```

Move Semantics

A Trivial Class: Constructors

```
class Customer {
  private:
    std::string name;
  public:
    Customer (const std::string& n) : name{n} {
      assert(n.size() > 0);
    }
    void setName(const std::string& n) {
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

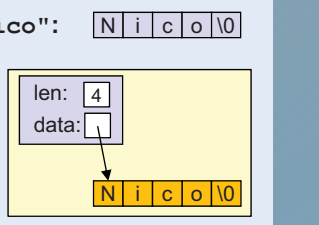
```
Customer c{"Nico"};
```



A Trivial Class: Constructors

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(const std::string& n) {
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

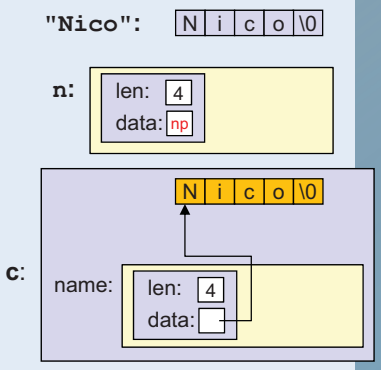
```
Customer c{"Nico"};
```



A Trivial Class: Constructors

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(const std::string& n) {
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

```
Customer c{"Nico"};
```



A Trivial Class: Setters

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

Pass Sink parameters
- to initialize or
- to set members

by-value and move

A Trivial Class: Getters

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

Breaks encapsulation of internal members

```
Customer c{"Nico"};
c.getName() = "";
```

A Trivial Class: Getters

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }

    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

```
Customer c{"Nico"};
c.getName() = "";   // OK, ERROR
```


A Trivial Class: Getters

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }

    const std::string& getName() const {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

```
Customer crC(...); // return new customer

// loop over chars of name of a temporary:
for (char c : crC(p).getName()) {
  cout << c;   // run-time ERROR
}
```

A Trivial Class: Getters

References extend lifetime of returned temporaries:

```
const auto& tmp1 = crC(p);           // does extend lifetime of return value
const auto& tmp2 = tmp1.getName();   // OK
std::cout << tmp2 << '\n';           // OK
```

but not of indirectly referenced temporaries:

```
const auto& tmp = crC(p).getName(); // does not extend lifetime of return value of crC()
std::cout << tmp << '\n';           // ERROR
```

Range-based `for` loop:

```
auto&& _rg = crC(p).getName();      // does not extend lifetime of return value of crC()
for (auto _pos=_rg.begin(), _end=_rg.end(); _pos!=_end; ++_pos ) {
  char c = *_pos;
  cout << c;
}
```

```
Customer crC(...); // return new customer

// loop over chars of name of a temporary:
for (char c : crC(p).getName()) {
  cout << c;  // run-time ERROR
}
```

```
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```


A Trivial Class: Getters

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }
    std::string getName() && {
      return name;
    }
    const std::string& getName() const& {
      return name;
    }
};

std::ostream& operator<< (std::ostream& os, const Customer& c) {
  return os << c.getName();
}
```

Don't return references to members of temporaries (prvalues)

Hidden Friends

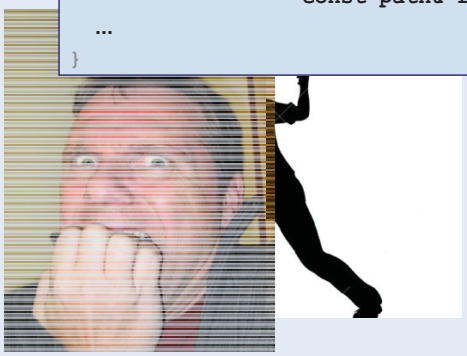
Hidden Friends?

```
#include <cassert>
#include <string>


using namespace std;


int main()
{
  auto x = L"a//b";
  auto y = string("a/b");
  assert(x == y); // fails
}
```

Hidden Friends?

```
#include <cassert>
#include <string>
#include <filesystem>

using namespace std;
using namespace std::filesystem;

int main()
{
  auto x = L"a//b";
  auto y = string("a/b");
  assert(x == y); // passes
}
```

```
namespace std::filesystem {
 class path {
  public:
    …
    path(string&& source);
    template<class Source>
      path(const Source& source);
    …
  };

  bool operator==(const path& l,
                  const path& r) noexcept;
  …
}
```



Hidden Friends?

```
#include <cassert>
#include <string>
#include <filesystem>

using namespace std;
using namespace std::filesystem;

int main()
{
  auto x = L"a//b";
  auto y = string("a/b");
  assert(x == y); // OK, fails
}
```

```
namespace std::filesystem {
 class path {
  public:
    …
    path(string&& source);
    template<class Source>
      path(const Source& source);
    …
    friend bool
      operator==(const path& l,
                 const path& r) noexcept;
    …
  };
}
```

A Trivial Class

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n)} {
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }
    std::string getName() && {
      return name;
    }
    const std::string& getName() const& {
      return name;
    }

    friend std::ostream& operator<< (std::ostream& os, const Customer& c) {
      return os << c.getName();
    }
};
```

Error messages list
30 instead of 31 candidates
for operator<<

Shades

Features Change Style

Rookie style

```
class Customer {
  private:
    std::string name;
  public:
    Customer (const std::string&
      assert(n.size() > 0);
    }
    void setName(const std::strin
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName()
      return name;
    }
};

std::ostream& operator<< (std::os
  return os << c.getName();
}
```

Experienced style

```
class Customer {
  private:
    std::string name;
  public:
    Customer (std::string n) : name{std::move(n
      assert(n.size() > 0);
    }
    void setName(std::string n) {
      assert(n.size() > 0);
      name = std::move(n);
    }
    std::string getName() && {
      return name;
    }
    const std::string& getName() const& {
      return name;
    }

    friend std::ostream& operator<< (std::ostre
      return os << c.getName();
    }
};
```


Roles Change Style

Rookie style

```
class Customer {
  private:
    std::string name;
  public:
    Customer (const std::string&
      assert(n.size() > 0);
    }
    void setName(const std::strin
      assert(n.size() > 0);
      name = n;
    }
    std::string& getName() {
      return name;
    }
    const std::string& getName()
      return name;
    }
};

std::ostream& operator<< (std::os
  return os << c.getName();
}
```

Experienced style

```
class Customer {
  private:
    std::string name;
  public:
    Customer
      assert
    }
    void setl
      assert
      name =
    }
    std::str
      return
    }
    const st
      return
    }

    fri
    rn
```

```
class Customer {
  private:
    std::string name;
  public:
    Customer (const std::string& n
      assert(n.size() > 0);
    }
    Customer (std::string&& n) : n
      assert(n.size() > 0);
    }
    Customer (const char* n) : nam
      assert(n.size() > 0);
    }
    void setName(const std::string
      assert(n.size() > 0);
      name = n;
    }
    void setName(std::string&& n)
      assert(n.size() > 0);
      name = std::move(n);
    }
    void setName(const char* n) {
```

Foundation programmer style

50 Shades of C++

There is no single C++ style

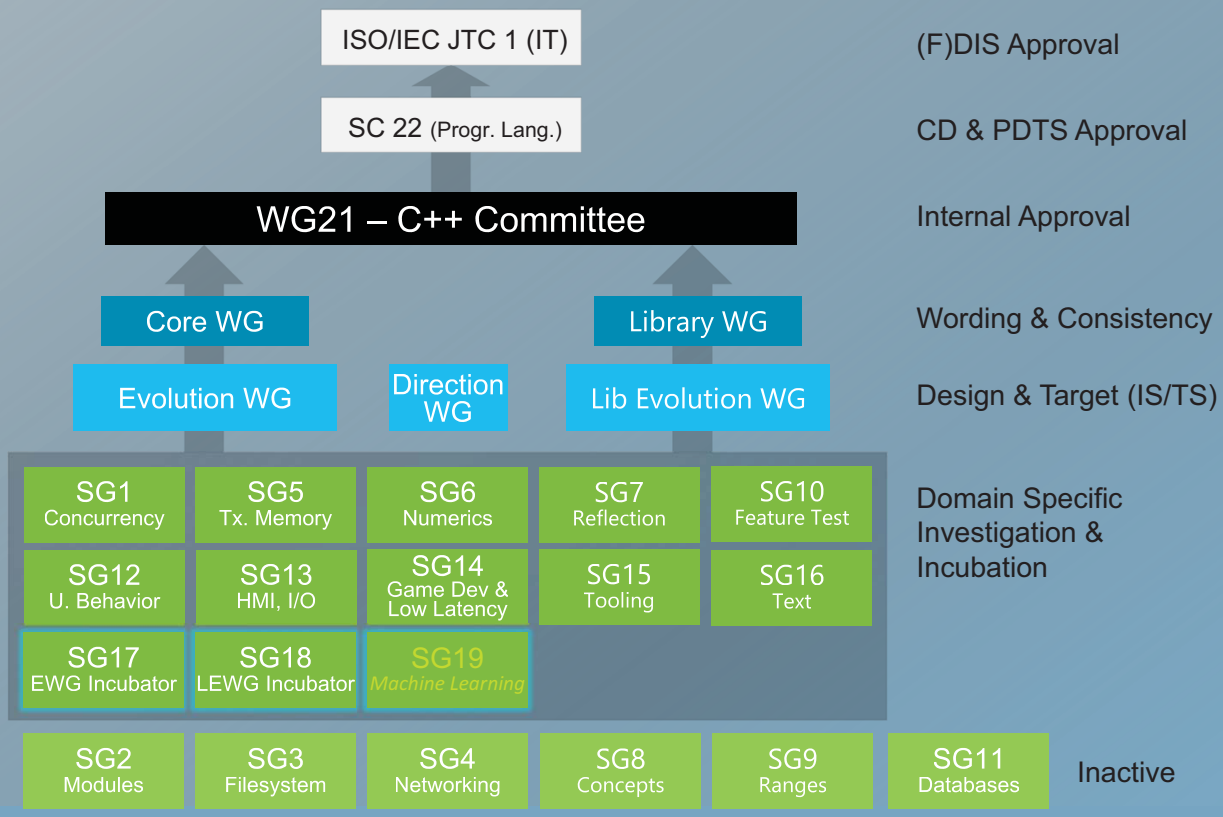
There is no single way teaching C++

Why?

C++

- A project that started ~50 years ago
- Constantly evolving
- Backward compatibility is key
  - API & ABI
- Community driven

Formal C++ Standardization Organization



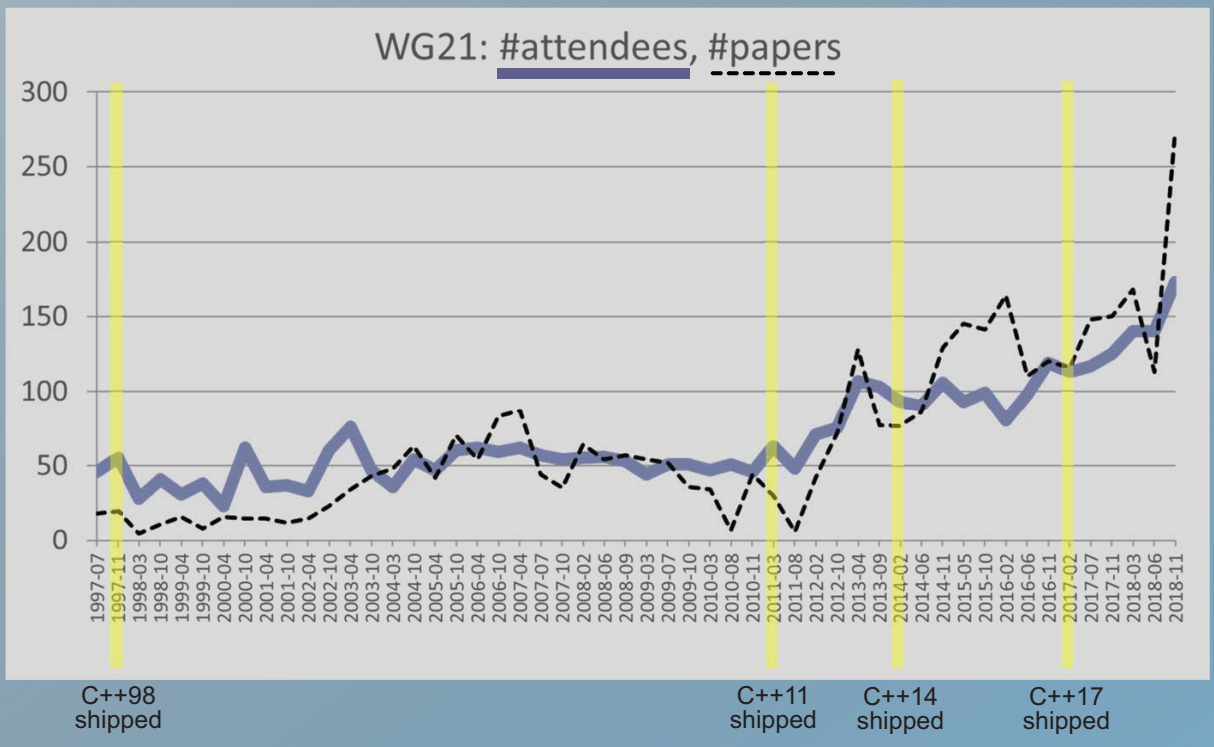
Slide by Herb Sutter 11/2018 with permission

Community Driven

- **Community-Driven Software**
  - No chief architect
  - We all care

It's your fault, Andrei
when "`if constexpr`" is not the way you want it !
You didn't come to propose it.

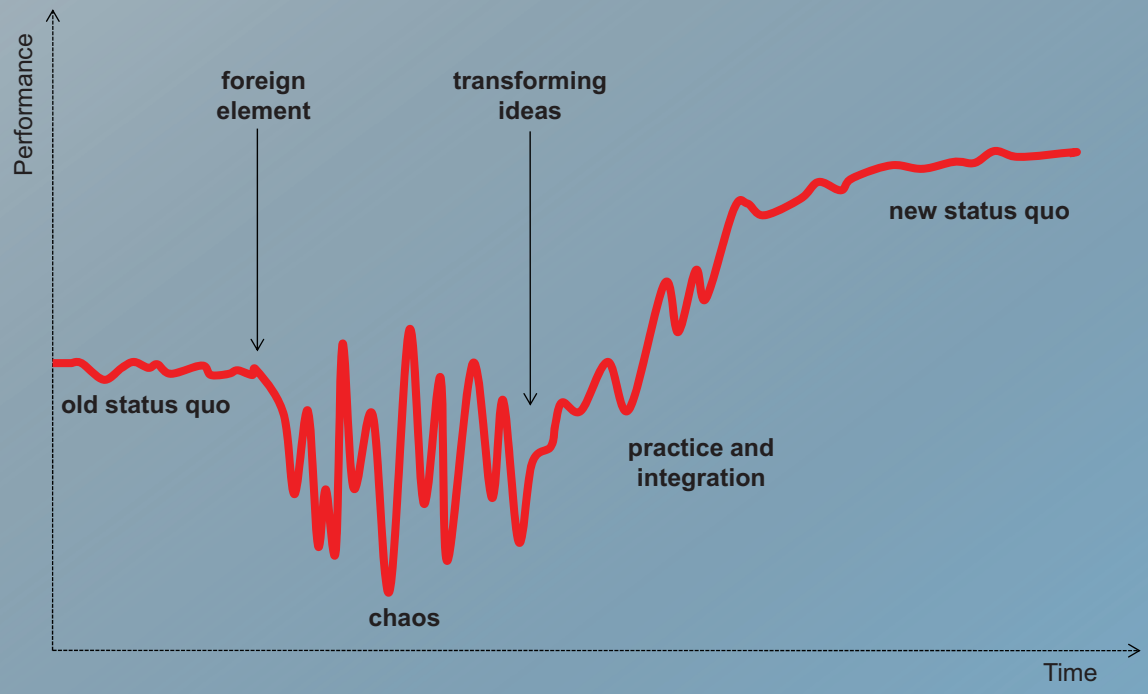
Growth of C++ Standardization



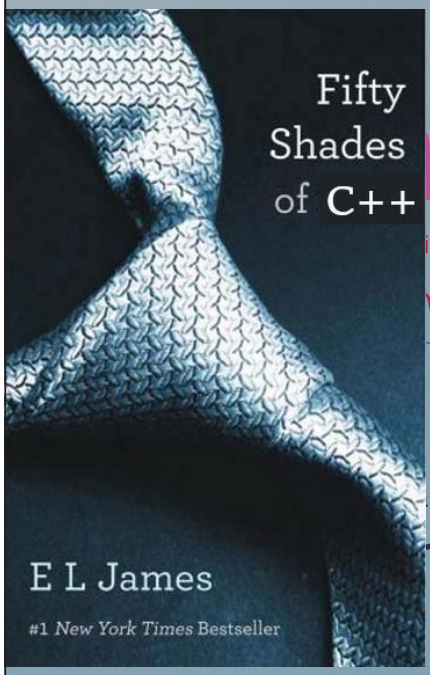
Slide by Herb Sutter 11/2018 with permission

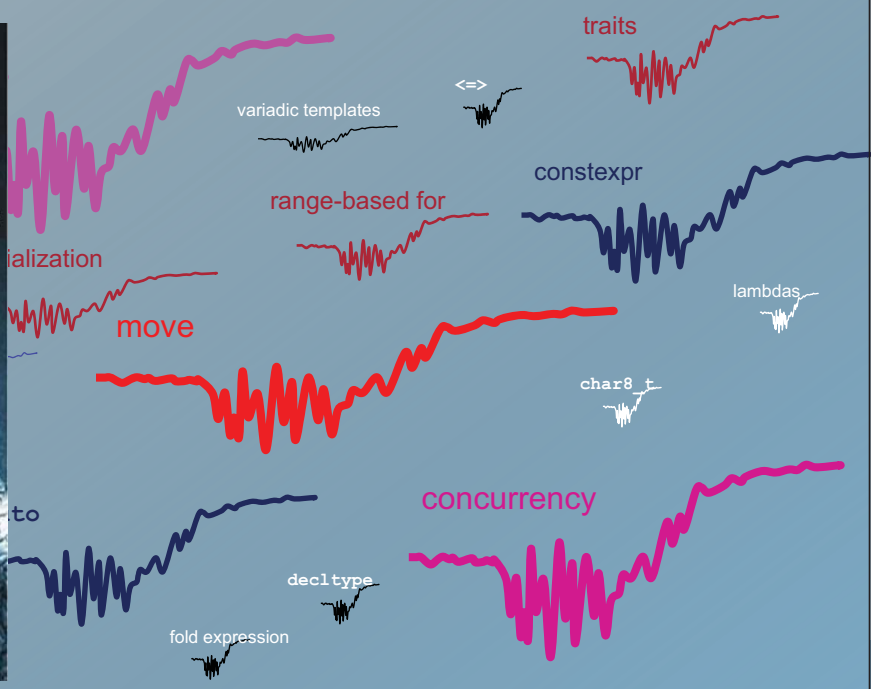
Virginia Satir Change Model

http://www.satirworkshops.com/files/satirchangemodel.pdf

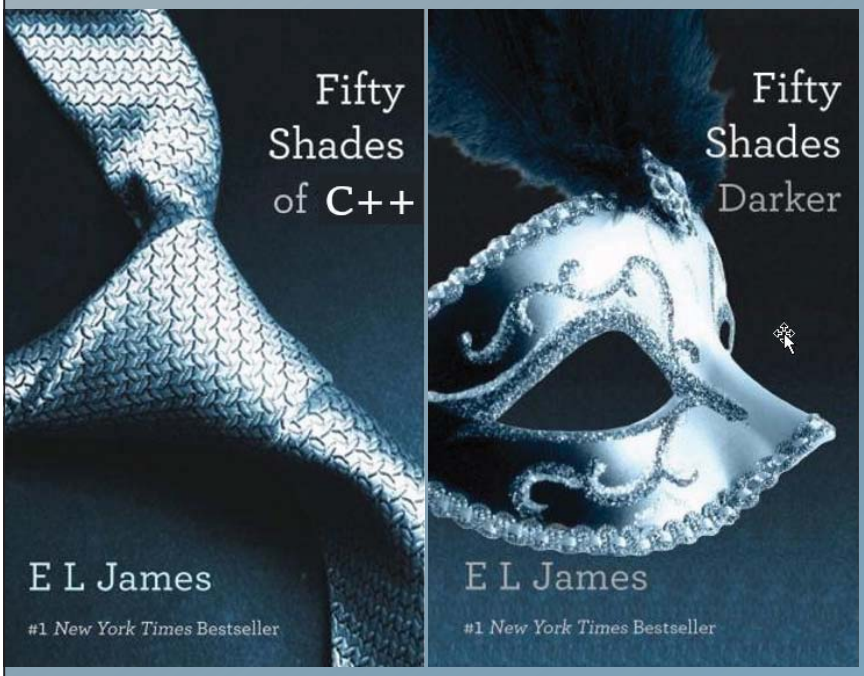


50 Shades of C++





Darker



Value Categories

K&R C:

```
int i;
i = 42;          // OK
42 = i;          // ERROR
int* p = &i;     // OK for lvalue only
int* q = &42;    // ERROR
```

i is lvalue:
OK on left-hand side of an assignment
42 is rvalue:
only right-hand-side of an assignment

ANSI C:

```
const int c = 0;
c = 42;                  // ERROR => no lvalue
const int* r = &c;       // OK    => lvalue
```

c is lvalue:
localizable value
rvalue:
read-only value ("roughly")

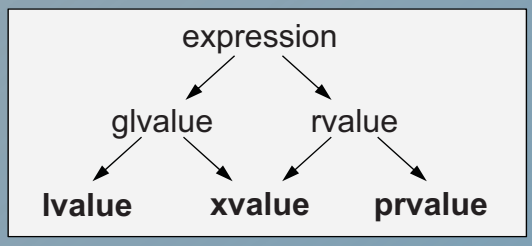
C++11:

```
std::string s;
std::move(s) = "hello";   // OK    => lvalue
auto sp = &std::move(s);  // ERROR => no lvalue
```

std::move(...) is xvalue:
- a bit like lvalue
- a bit like rvalue

Most of rvalue rules apply to xvalues:
- Let's rename rvalue to prvalue
- and let rvalue represent both

Value Categories since C++11



- **LValue: Localizable value**
  - Variable, data member, function, string literal, returned lvalue reference
  - Can be on the left side of an assignment only if it's modifiable

Everything that has a name and string literals

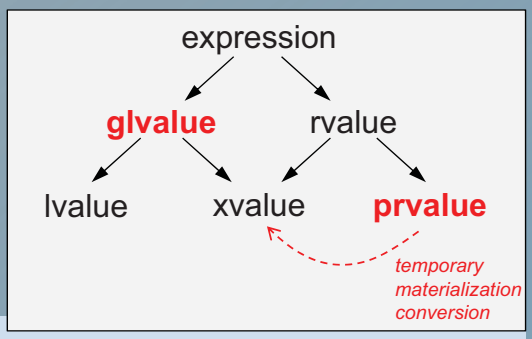
- **PRValue: Pure RValue** (former RValue)
  - All Literals except string literals (`42`, `true`, `nullptr`,...), `this`, lambda, returned non-reference, result of constructor call (`T(...)`)

Temporaries and other literals

- **XValue: eXpiring value**
  - Returned rvalue reference (e.g. by **std::move()**), cast to rvalue reference

Value from std::move()

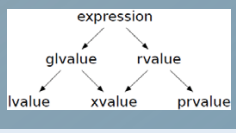
Value Categories since C++17



C++17:

- **prvalues perform initialization**
  - No temporary object yet
- **glvalues produce locations**
- ***Materialization* as a temporary object:**
  - prvalue-to-xvalue conversion

Type versus Value Category



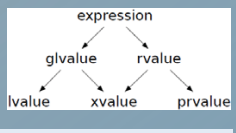
can only pass rvalues

```
void passRValueRef (std::string&&);   // declaration

std::string s;              // s is lvalue
passRValueRef(s);           // ERROR: cannot bind lvalue to rvalue reference
passRValueRef("hello");     // OK (string literal (lvalue) converted to std::string (prvalue))
passRValueRef(std::string("hello"));   // OK (can pass prvalue)
passRValueRef(std::move(s));           // OK (can pass xvalue)


void passRValueRef (std::string&& s)
{
  passRValueRef(s);                // ERROR: cannot bind lvalue to rvalue reference
  passRValueRef(std::move(s));     // OK (can pass xvalue)
}
```

s has:
- Type std::string&&
- Value category lvalue

move() casts s back to its type:
static_cast<std::string&&>(s)

Universal References



- can only pass rvalues
- s not const

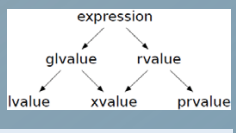
```
void passRValueRef (std::string&& s);

std::string s;              // s is lvalue
passRValueRef(s);           // ERROR: cannot bind lvalue to rvalue reference
const std::string cs;       // cs is lvalue
passRValueRef(cs);          // ERROR: cannot bind lvalue to rvalue reference
```

- universal reference
  - "rvalue reference that is a *forwarding reference*"
- s can be const

```
template <typename T>
void passRValueRef (T&& s);

std::string s;
passRValueRef(s);
const std::string cs
passRValueRef(cs);
```

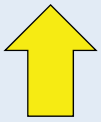
```
template <typename T>
void passRValueRef (typename T::iterator&&);

std::string::iterator pos1;          // s is lvalue
passRValueRef(pos1);                 // ???
const std::string::iterator pos2;    // cs is lvalue
passRValueRef(pos2);                 // ???
```

Rejected Syntax for Universal References




```
void passRValueRef (std::string&& s);
```

```
template <typename T>
void passRValueRef (T&&& s);
```



Overloading syntax with different semantics is
- **cool for experts**
- **a nightmare for application programmers**

Tough Guys

- Google for "*C++ Bondage*"


```
// "const int ptr":
int const *
```

"Just to confuse the rookies"

Cool Guys


```
const int *
```

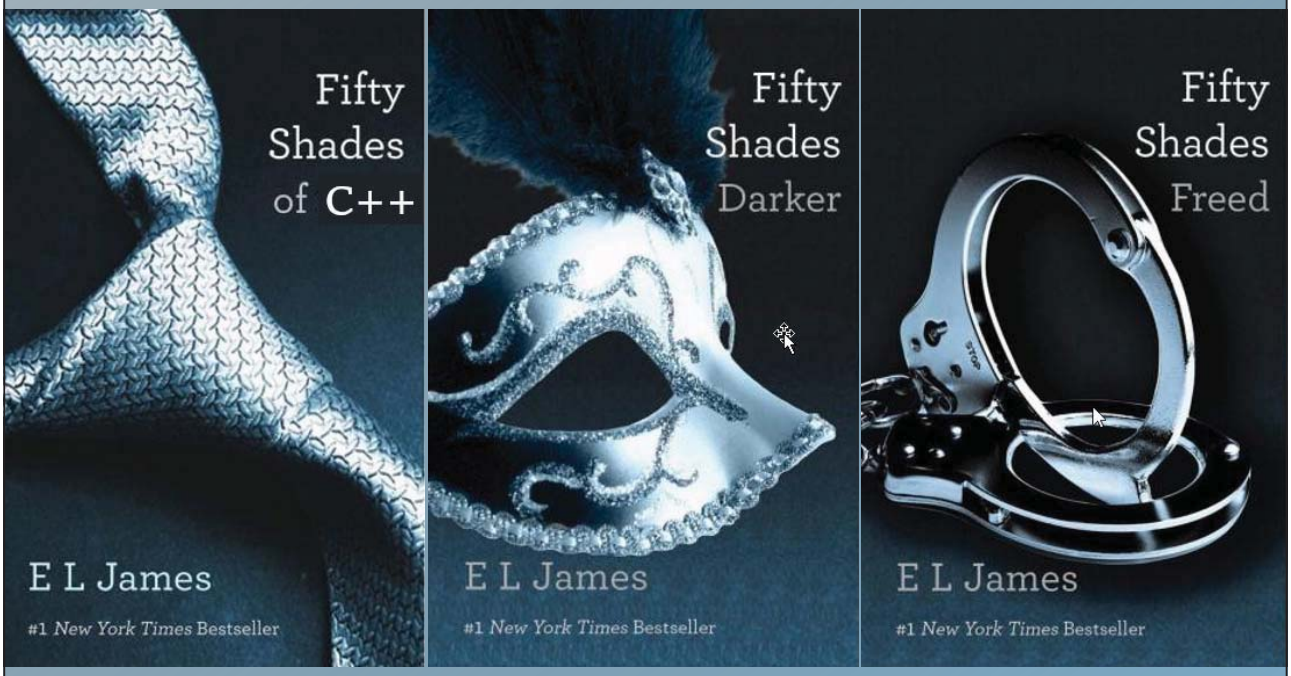
✓

```
int * const
```

✓

German Chaos

Freed



Freed

We want you, application programmer for the standardization of C++

Scott Meyers
Herb Sutter

We need style guides

When Scott and Herb Give Different Advice

```
void sink(std::unique_ptr<MyType> up)  // sink() gets ownership in up
{
    ...
}

std::unique_ptr<MyType> up(new MyType);
...
sink(std::move(up));  // up loses ownership
sink(up);             // Error because copying of
                      //       type of up is deleted
```

Herb's style
- **no claim to move**
- **guarantee to move**

```
void sink(std::unique_ptr<MyType>&& up)  // up refers to passed argument
{
    ...
}

std::unique_ptr<MyType> up(new MyType);
...
sink(std::move(up));  // up might lose ownership
sink(up);             // Error due to declaration of sink()
```

Scott's style
- **claim to possibly move**
- **no guarantee to move**

Contradicting Style Guides: Core Guidelines

- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-list

ES.23: Prefer the {} initializer syntax

– **Reason**
  - The rules for { } initialization are simpler, more general, less ambiguous, and safer than for other forms of initialization.

– **Note**
  - Old habits die hard, so this rule is hard to apply consistently, especially as there are so many cases where = is innocent.

Contradicting Style Guides: Abseil 1/2

- https://abseil.io/tips/88#best-practices-for-initialization
  As of 9/2018

  ...
  - Direct uniform initialization has two shortcomings;
    1. "uniform" is a stretch: there are cases where ambiguity still exists (for the casual reader, not the compiler) in what is being called and how.
    2. this syntax is not exactly intuitive: no other common language uses something like it.

  ...
  - The important question is: how much should we change our habits and language understanding to take advantage of that change?
  - For uniform initialization syntax, we don't believe in general that the benefits outweigh the drawbacks.

Contradicting Style Guides: Abseil 2/2

- https://abseil.io/tips/88#best-practices-for-initialization
  As of 9/2018
  - Use assignment syntax when
    - initializing directly
      - with the intended literal value (for example: int, float, or std::string values),
      - with smart pointers such as std::shared_ptr, std::unique_ptr,
      - with containers (std::vector, std::map, etc),
    - when performing struct initialization, or
    - doing copy construction.
  - Use the traditional constructor syntax (with parentheses) when the initialization is performing some active logic, rather than simply composing values together.
  - Use {} initialization without the = only if the above options don't compile.
  - Never mix {}s and auto.

C++

- **Houston, we have a problem**
  - A project that started ~50 years ago
  - Constantly evolving
  - Backward compatibility is key
    - API & ABI
- **Driven by the community**
  - of experts
- **Please, take care**
  - Style Guides

Nicolai Josuttis

Thank you and take care

@NicoJosuttis
josuttis.com