C++ decade: C++11 → C++20

Serbian C++ User Group meetup,
Belgrade, May 2022

Goran Aranđelović
goran.arandjelovic@gmail.com
cppserbia@cpplang.com

Early C++

1979 — C with Classes first implemented
1982 — C with Classes reference manual published
1985 — Cfront 1.0 (first original C++ compiler, Bjarne Stroustrup)
1985 — The C++ Programming Language, 1st edition
1989 — Cfront 2.0
1990 — The Annotated C++ Reference Manual (de-facto C++ standard)
1991 — Cfront 3.0
1991 — The C++ Programming Language, 2nd edition

Standard C++

1990 — ANSI C++ Committee founded
1991 — ISO C++ Committee founded
1992 — STL implemented in C++ (Alexander Stepanov, David Musser)

Standard C++ (C++98/03 period)

1998 — C++98
1998 — The C++ Programming Language, 3rd edition
1999 — Boost founded (boost.org)
2003 — C++03
2006 — Performance TR
2007 — Library extension TR1 (shared_ptr, weak_ptr, function, bind, regex, tuple, ...)
2010 — Mathematical special functions

Standard C++ (C++11 period)

2011 — C++11
2011 — Decimal floating-point TR
2012 — The Standard C++ Foundation founded (isocpp.org)
2013 — The C++ Programming Language, 4th edition

Standard C++ (C++14 period)

2014 — C++14
2015 — Filesystem library TS
2015 — Extensions for Parallelism TS
2015 — Extensions for Transactional Memory TS
2015 — Extensions for Library Fundamentals TS
2015 — Extensions for Concepts TS
2016 — Extensions for Concurrency TS

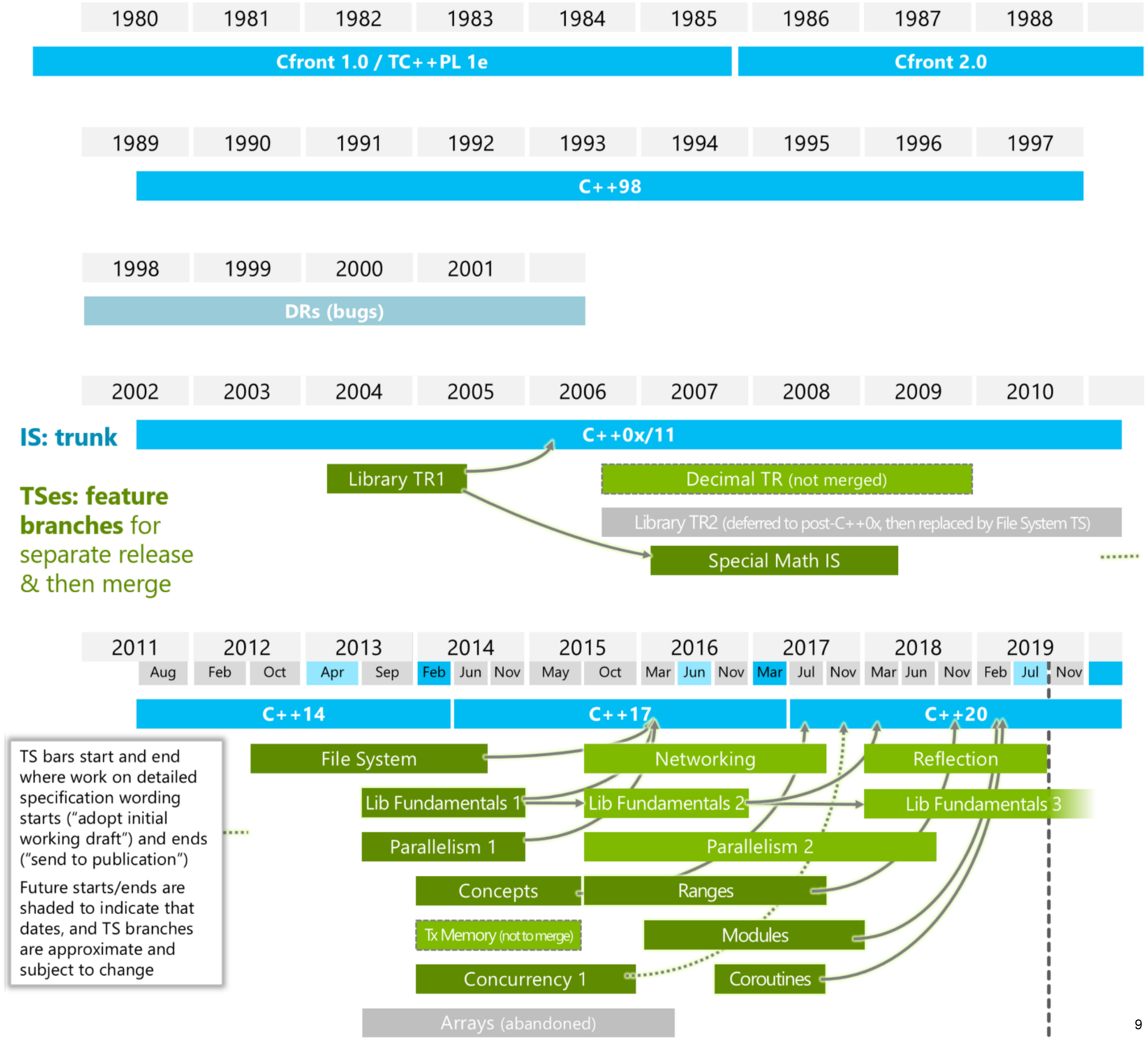
Standard C++ (C++17 period)

2017 — C++17
2017 — Extensions for Ranges TS
2017 — Extensions for Coroutines TS
2018 — Extensions for Networking TS
2018 — Extensions for modules TS
2018 — Extensions for Parallelism version 2 TS

Standard C++ (C++20 period)

2020 — C++20
2021 — Reflection TS

ISO C++ Standardization



C++11 (core language features)

(The list is not exhaustive)

- auto and decltype
- defaulted and deleted functions
- final and override
- trailing return type
- rvalue references
- move constructors and move assignment operators
- scoped enums
- constexpr and literal types
- list initialization
- delegating and inherited constructors
- brace-or-equal initializers
- nullptr
- type aliases
- variadic templates
- attributes
- lambda expressions
- noexcept specifier and noexcept operator
- thread-local storage
- range-for (based on a Boost library)
- static_assert (based on a Boost library)

C++11 (library features)

(The list is not exhaustive)

- concurrency support library
- emplace()
- std::unique_ptr
- std::move_iterator
- std::initializer_list
- std::forward_list
- chrono library
- ratio library
- new algorithms (std::all_of, std::any_of, std::none_of, std::find_if_not, std::copy_if, std::copy_n, std::move, std::move_backward, std::random_shuffle, std::shuffle, std::is_partitioned, std::partition_copy, std::partition_point, std::is_sorted, std::is_sorted_until, std::is_heap, std::is_heap_until, std::minmax, std::minmax_element, std::is_permutation, std::iota, std::uninitialized_copy_n)
- std::function
- std::exception_ptr
- std::error_code and std::error_condition
- iterator improvements (std::begin, std::end, std::next, std::prev)

C++14 (core language features)

(The list is not exhaustive)

- variable templates
- generic lambdas
- lambda init-capture
- return type deduction for functions
- decltype(auto)
- aggregate classes with default non-static member initializers

C++14 (library features)

(The list is not exhaustive)

- std::make_unique
- std::shared_timed_mutex and std::shared_lock
- std::integer_sequence
- std::exchange
- std::quoted

C++17 (core language features)

(The list is not exhaustive)

- fold-expressions
- class template argument deduction
- non-type template parameters declared with auto
- compile-time if constexpr
- inline variables
- structured bindings
- initializers for if and switch
- simplified nested namespaces
- made noexcept part of type system
- new order of evaluation rules
- guaranteed copy elision
- temporary materialization
- lambda capture of *this
- constexpr lambda
- new attributes ([[fallthrough]], [[maybe_unused]], [[nodiscard]])

C++17 (library features)

(The list is not exhaustive)

- std::tuple helpers (std::apply, std::make_from_tuple, deduction guides)
- std::any
- std::optional
- std::variant
- std::string_view
- std::as_const
- polymorphic allocators
- parallel algorithms and execution policies
- cache line interface
- std::launder
- std::uncaught_exceptions
- std::to_chars/std::from_chars
- std::scoped_lock

C++20 (core language features)

(The list is not exhaustive)

- Feature test macros
- 3-way comparison operator <=> and operator==() = default
- designated initializers
- init-statements and initializers in range-for
- New attributes: ([[no_unique_address]], [[likely]], [[unlikely]])
- pack-expansions in lambda init-captures
- consteval, constinit
- aggregate initialization using parentheses
- Coroutines
- Modules
- Constraints and concepts
- Abbreviated function templates

C++20 (library features)

(The list is not exhaustive)

- Library feature-test macros
- Formatting library
- Calendar and Time Zone library
- std::source_location
- std::span
- thread-coordination classes: barrier, latch, and counting_semaphore
- jthread and thread cancellation classes: stop_token, stop_source, and stop_callback
- bind_front
- Ranges
- uniform container erasure: (std::erase/std::erase_if, e.g. std::erase(std::list) or erase_if(std::map), etc.)

cpplang.com/survey

C++11

C++11 (rvalue references)

```
 1 int x = 0; // `x` is an lvalue of type `int`
 2 int& xl = x; // `xl` is an lvalue of type `int&`
 3 int&& xr = x; // compiler error -- `x` is an lvalue
 4 int&& xr2 = 0; // `xr2` is an lvalue of type `int&&` -- binds to the rvalue temporary, `0`
 5
 6 void f(int& x) {}
 7 void f(int&& x) {}
 8
 9 f(x);  // calls f(int&)
10 f(xl); // calls f(int&)
11 f(3);  // calls f(int&&)
12 f(std::move(x)); // calls f(int&&)
13
14 f(xr2);            // calls f(int&)
15 f(std::move(xr2)); // calls f(int&& x)
```

C++11 (forwarding references)

```
1 int x = 0; // `x` is an lvalue of type `int`
2 auto&& al = x; // `al` is an lvalue of type `int&` -- binds to the lvalue, `x`
3 auto&& ar = 0; // `ar` is an lvalue of type `int&&` -- binds to the rvalue temporary, `0`
```

C++11 (forwarding references)

Reference collapsing rules:

- `T& & becomes T&`
- `T& && becomes T&`
- `T&& & becomes T&`
- `T&& && becomes T&&`

C++11 (forwarding references)

```cpp
// Since C++11 or later:
template <typename T>
void f(T&& t) {
  // ...
}

int x = 0;
f(0); // T is int, deduces as f(int &&) => f(int&&)
f(x); // T is int&, deduces as f(int& &&) => f(int&)

int& y = x;
f(y); // T is int&, deduces as f(int& &&) => f(int&)

int&& z = 0; // NOTE: `z` is an lvalue with type `int&&`.
f(z); // T is int&, deduces as f(int& &&) => f(int&)
f(std::move(z)); // T is int, deduces as f(int &&) => f(int&&)
```

C++11 (variadic templates)

```
1 template <typename... T>
2 struct arity {
3   constexpr static int value = sizeof...(T);
4 };
5 static_assert(arity<>::value == 0);
6 static_assert(arity<char, short, int>::value == 3);
```

C++11 (variadic templates)

```cpp
template <typename First, typename... Args>
auto sum(const First first, const Args... args) -> decltype(first) {
  const auto values = {first, args...};
  return std::accumulate(values.begin(), values.end(), First{0});
}

sum(1, 2, 3, 4, 5); // 15
sum(1, 2, 3);       // 6
sum(1.5, 2.0, 3.7); // 7.2
```

C++11 (initializer lists)

```
1  int sum(const std::initializer_list<int>& list) {
2    int total = 0;
3    for (auto& e : list) {
4      total += e;
5    }
6
7    return total;
8  }
9
10 auto list = {1, 2, 3};
11 sum(list); // == 6
12 sum({1, 2, 3}); // == 6
13 sum({}); // == 0
```

C++11 (static assertions)

```
1 constexpr int x = 0;
2 constexpr int y = 1;
3 static_assert(x == y, "x != y");
```

C++11 (auto)

```
 1 auto a = 3.14; // double
 2 auto b = 1; // int
 3 auto& c = b; // int&
 4 auto d = { 0 }; // std::initializer_list<int>
 5 auto&& e = 1; // int&&
 6 auto&& f = b; // int&
 7 auto g = new auto(123); // int*
 8 const auto h = 1; // const int
 9 auto i = 1, j = 2, k = 3; // int, int, int
10 auto l = 1, m = true, n = 1.61; // error -- `l` deduced to be int, `m` is bool
11 auto o; // error -- `o` requires initializer
```

C++11 (auto)

```
1 std::vector<int> v = ...;
2 std::vector<int>::const_iterator cit = v.cbegin();
3 // vs.
4 auto cit = v.cbegin();
```

C++11 (auto)

```
1 template <typename X, typename Y>
2 auto add(X x, Y y) -> decltype(x + y) {
3   return x + y;
4 }
5 add(1, 2); // == 3
6 add(1, 2.0); // == 3.0
7 add(1.5, 1.5); // == 3.0
```

C++11 (lambda expressions)

```cpp
int x = 1;

auto getX = [=] { return x; };
getX(); // == 1

auto addX = [=](int y) { return x + y; };
addX(1); // == 2

auto getXRef = [&]() -> int& { return x; };
getXRef(); // int& to `x`
```

C++11 (lambda expressions)

```cpp
int x = 1;

auto f1 = [&x] { x = 2; }; // OK: x is a reference and modifies the original

auto f2 = [x] { x = 2; };
// ERROR: the lambda can only perform const-operations on the captured value

// vs.
auto f3 = [x]() mutable { x = 2; };
// OK: the lambda can perform any operations on the captured value
```

C++11 (decltype)

```
1 int a = 1; // `a` is declared as type `int`
2 decltype(a) b = a; // `decltype(a)` is `int`
3 const int& c = a; // `c` is declared as type `const int&`
4 decltype(c) d = a; // `decltype(c)` is `const int&`
5 decltype(123) e = 123; // `decltype(123)` is `int`
6 int&& f = 1; // `f` is declared as type `int&&`
7 decltype(f) g = 1; // `decltype(f) is `int&&`
8 decltype((a)) h = g; // `decltype((a))` is int&
```

C++11 (decltype)

```
1 template <typename X, typename Y>
2 auto add(X x, Y y) -> decltype(x + y) {
3   return x + y;
4 }
5 add(1, 2.0); // `decltype(x + y)` => `decltype(3.0)` => `double`
```

C++11 (type aliases)

```
1 template <typename T>
2 using Vec = std::vector<T>;
3 Vec<int> v; // std::vector<int>
4
5 using String = std::string;
6 String s {"foo"};
```

C++11 (nullptr)

```
1 void foo(int);
2 void foo(char*);
3 foo(NULL); // error -- ambiguous
4 foo(nullptr); // calls foo(char*)
```

C++11 (strongly-typed enums)

```
1 // Specifying underlying type as `unsigned int`
2 enum class Color : unsigned int { Red = 0xff0000, Green = 0xff00, Blue = 0xff };
3 // `Red`/`Green` in `Alert` don't conflict with `Color`
4 enum class Alert : bool { Red, Green };
5 Color c = Color::Red;
```

C++11 (attributes)

```
1 // `noreturn` attribute indicates `f` doesn't return.
2 [[ noreturn ]] void f() {
3   throw "error";
4 }
```

C++11 (constexpr)

```
 1 constexpr int square(int x) {
 2   return x * x;
 3 }
 4
 5 int square2(int x) {
 6   return x * x;
 7 }
 8
 9 int a = square(2);  // mov DWORD PTR [rbp-4], 4
10
11 int b = square2(2); // mov edi, 2
12                     // call square2(int)
13                     // mov DWORD PTR [rbp-8], eax
```

C++11 (constexpr)

```
1 const int x = 123;
2
3 constexpr const int& y = x;
4 // error -- constexpr variable `y` must be initialized by a constant expression
```

C++11 (constexpr)

```
1  struct Complex {
2    constexpr Complex(double r, double i) : re{r}, im{i} { }
3    constexpr double real() { return re; }
4    constexpr double imag() { return im; }
5
6  private:
7    double re;
8    double im;
9  };
10
11 constexpr Complex I(0, 1);
```

C++11 (delegating constructors)

```
1 struct Foo {
2   int foo;
3   Foo(int foo) : foo{foo} {}
4   Foo() : Foo(0) {}
5 };
6
7 Foo foo;
8 foo.foo; // == 0
```

C++11 (user-defined literals)

```cpp
// `const char*` and `std::size_t` required as parameters.
int operator "" _int(const char* str, std::size_t) {
  return std::stoi(str);
}

"123"_int; // == 123, with type `int`
```

C++11 (override)

```
1  struct A {
2    virtual void foo();
3    void bar();
4  };
5
6  struct B : A {
7    void foo() override; // correct -- B::foo overrides A::foo
8    void bar() override; // error -- A::bar is not virtual
9    void baz() override; // error -- B::baz does not override A::baz
10 };
```

C++11 (final)

```
 1 struct A {
 2   virtual void foo();
 3 };
 4
 5 struct B : A {
 6   virtual void foo() final;
 7 };
 8
 9 struct C : B {
10   virtual void foo(); // error -- declaration of 'foo' overrides a 'final' function
11 };
```

C++11 (final)

```
1 struct A final {};
2 struct B : A {}; // error -- base 'A' is marked 'final'
```

C++11 (default functions)

```
1 struct A {
2   A() = default;
3   A(int x) : x{x} {}
4   int x {1};
5 };
6 A a; // a.x == 1
7 A a2 {123}; // a.x == 123
```

C++11 (default functions)

```
 1 struct B {
 2   B() : x{1} {}
 3   int x;
 4 };
 5
 6 struct C : B {
 7   // Calls B::B
 8   C() = default;
 9 };
10
11 C c; // c.x == 1
```

C++11 (deleted functions)

```
 1 class A {
 2   int x;
 3
 4 public:
 5   A(int x) : x{x} {};
 6   A(const A&) = delete;
 7   A& operator=(const A&) = delete;
 8 };
 9
10 A x {123};
11 A y = x; // error -- call to deleted copy constructor
12 y = x; // error -- operator= deleted
```

C++11 (range-based for loops)

```
1 std::array<int, 5> a {1, 2, 3, 4, 5};
2 for (int& x : a) {
3   x *= 2;
4 }
5 // a == { 2, 4, 6, 8, 10 }
```

C++11 (special member functions)

```cpp
struct A {
  std::string s;
  A() : s{"test"} {}
  A(const A& o) : s{o.s} {}
  A(A&& o) : s{std::move(o.s)} {}
  A& operator=(A&& o) {
   s = std::move(o.s);
   return *this;
  }
};

A f(A a) {
  return a;
}

A a1 = f(A{}); // move-constructed from rvalue temporary
A a2 = std::move(a1); // move-constructed using std::move
A a3 = A{};
a2 = std::move(a3); // move-assignment using std::move
a1 = f(A{}); // move-assignment from rvalue temporary
```

C++11 (converting constructors)

```
 1 struct A {
 2   A(int) {}
 3   A(int, int) {}
 4   A(int, int, int) {}
 5 };
 6
 7 A a {0, 0}; // calls A::A(int, int)
 8 A b(0, 0); // calls A::A(int, int)
 9 A c = {0, 0}; // calls A::A(int, int)
10 A d {0, 0, 0}; // calls A::A(int, int, int)
```

C++11 (converting constructors)

```
1 struct A {
2   A(int) {}
3 };
4
5 A a(1.1); // OK
6 A b {1.1}; // Error narrowing conversion from double to int
```

C++11 (converting constructors)

```cpp
struct A {
  A(int) {}
  A(int, int) {}
  A(int, int, int) {}
  A(std::initializer_list<int>) {}
};

A a {0, 0}; // calls A::A(std::initializer_list<int>)
A b(0, 0); // calls A::A(int, int)
A c = {0, 0}; // calls A::A(std::initializer_list<int>)
A d {0, 0, 0}; // calls A::A(std::initializer_list<int>)
```

C++11 (explicit conversion functions)

```
 1 struct A {
 2   operator bool() const { return true; }
 3 };
 4
 5 struct B {
 6   explicit operator bool() const { return true; }
 7 };
 8
 9 A a;
10 if (a); // OK calls A::operator bool()
11 bool ba = a; // OK copy-initialization selects A::operator bool()
12
13 B b;
14 if (b); // OK calls B::operator bool()
15 bool bb = b; // error copy-initialization does not consider B::operator bool()
```

C++11 (inline namespaces)

```
 1 namespace Program {
 2   namespace Version1 {
 3     int getVersion() { return 1; }
 4     bool isFirstVersion() { return true; }
 5   }
 6   inline namespace Version2 {
 7     int getVersion() { return 2; }
 8   }
 9 }
10
11 int version {Program::getVersion()};                // Uses getVersion() from Version2
12 int oldVersion {Program::Version1::getVersion()}; // Uses getVersion() from Version1
13 bool firstVersion {Program::isFirstVersion()};     // Does not compile when Version2 is added
```

C++11 (NSDMI)

```cpp
// Default initialization prior to C++11
class Human {
    Human() : age{0} {}
  private:
    unsigned age;
};
// Default initialization on C++11
class Human {
  private:
    unsigned age {0};
};
```

C++11 (ref-qualified member functions)

```cpp
struct Bar {
  // ...
};

struct Foo {
  Bar getBar() & { return bar; }
  Bar getBar() const& { return bar; }
  Bar getBar() && { return std::move(bar); }
private:
  Bar bar;
};

Foo foo{};
Bar bar = foo.getBar(); // calls `Bar getBar() &`

const Foo foo2{};
Bar bar2 = foo2.getBar(); // calls `Bar Foo::getBar() const&`

Foo{}.getBar(); // calls `Bar Foo::getBar() &&`
std::move(foo).getBar(); // calls `Bar Foo::getBar() &&`

std::move(foo2).getBar(); // calls `Bar Foo::getBar() const&&`
```

C++11 (trailing return types)

```
 1 int f() {
 2   return 123;
 3 }
 4 // vs.
 5 auto f() -> int {
 6   return 123;
 7 }
 8
 9 auto g = []() -> int {
10   return 123;
11 };
```

C++11 (trailing return types)

```cpp
// NOTE: This does not compile!
template <typename T, typename U>
decltype(a + b) add(T a, U b) {
    return a + b;
}

// Trailing return types allows this:
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}
```

C++11 (noexcept)

```
1 void func1() noexcept;          // does not throw
2 void func2() noexcept(true);   // does not throw
3 void func3() throw();          // does not throw
4
5 void func4() noexcept(false); // may throw
```

C++11 (noexcept)

```
1 extern void f();  // potentially-throwing
2 void g() noexcept {
3     f();          // valid, even if f throws
4     throw 42;     // valid, effectively a call to std::terminate
5 }
```

C++11 (std::move)

```
1 std::unique_ptr<int> p1 {new int{0}};  // in practice, use std::make_unique
2 std::unique_ptr<int> p2 = p1; // error -- cannot copy unique pointers
3 std::unique_ptr<int> p3 = std::move(p1); // move `p1` into `p3`
4                                          // now unsafe to dereference object held by `p1`
```

C++11 (std::forward)

```cpp
struct A {
  A() = default;
  A(const A& o) { std::cout << "copied" << std::endl; }
  A(A&& o) { std::cout << "moved" << std::endl; }
};

template <typename T>
A wrapper(T&& arg) {
  return A{std::forward<T>(arg)};
}

wrapper(A{}); // moved
A a;
wrapper(a); // copied
wrapper(std::move(a)); // moved
```

C++11 (std::thread)

```cpp
void foo(bool clause) { /* do something... */ }

std::vector<std::thread> threadsVector;
threadsVector.emplace_back([]() {
  // Lambda function that will be invoked
});
threadsVector.emplace_back(foo, true);  // thread will run foo(true)
for (auto& thread : threadsVector) {
  thread.join(); // Wait for threads to finish
}
```

C++11 (std::async)

```
1 int foo() {
2   /* Do something here, then return the result. */
3   return 1000;
4 }
5
6 auto handle = std::async(std::launch::async, foo);  // create an async task
7 auto result = handle.get();  // wait for the result
```

C++11 (type traits)

```
1 static_assert(std::is_integral<int>::value);
2 static_assert(std::is_same<int, int>::value);
3 static_assert(std::is_same<std::conditional<true, int, double>::type, int>::value);
```

C++11 (std::unique_ptr)

```cpp
std::unique_ptr<Foo> p1 { new Foo{} };  // `p1` owns `Foo`
if (p1) {
  p1->bar();
}

{
  std::unique_ptr<Foo> p2 {std::move(p1)};  // Now `p2` owns `Foo`
  f(*p2);

  p1 = std::move(p2);  // Ownership returns to `p1` -- `p2` gets destroyed
}

if (p1) {
  p1->bar();
}
// `Foo` instance is destroyed when `p1` goes out of scope
```

C++11 (std::shared_ptr)

```cpp
void foo(std::shared_ptr<T> t) {
  // Do something with `t`...
}

void bar(std::shared_ptr<T> t) {
  // Do something with `t`...
}

void baz(std::shared_ptr<T> t) {
  // Do something with `t`...
}

std::shared_ptr<T> p1 = std::make_shared<T>();
// Perhaps these take place in another threads?
foo(p1);
bar(p1);
baz(p1);
```

C++11 (tuples)

```cpp
// `playerProfile` has type `std::tuple<int, const char*, const char*>`.
auto playerProfile = std::make_tuple(51, "Frans Nielsen", "NYI");
std::get<0>(playerProfile); // 51
std::get<1>(playerProfile); // "Frans Nielsen"
std::get<2>(playerProfile); // "NYI"
```

C++11 (unordered containers)

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

C++14

C++14 (generic lambda expr.)

```
1 auto identity = [](auto x) { return x; };
2 int three = identity(3); // == 3
3 std::string foo = identity("foo"); // == "foo"
```

C++14 (lambda capture initializers)

```
 1 int factory(int i) { return i * 10; }
 2 auto f = [x = factory(2)] { return x; }; // returns 20
 3
 4 auto generator = [x = 0] () mutable {
 5   // this would not compile without 'mutable' as we are modifying x on each call
 6   return x++;
 7 };
 8 auto a = generator(); // == 0
 9 auto b = generator(); // == 1
10 auto c = generator(); // == 2
```

C++14 (lambda capture initializers)

```cpp
auto p = std::make_unique<int>(1);

auto task1 = [=] { *p = 5; }; // ERROR: std::unique_ptr cannot be copied
// vs.
auto task2 = [p = std::move(p)] { *p = 5; };
// OK: p is move-constructed into the closure object

// the original p is empty after task2 is created
```

C++14 (lambda capture initializers)

```
1 auto x = 1;
2 auto f = [&r = x, x = x * 10] {
3   ++r;
4   return r + x;
5 };
6 f(); // sets x to 2 and returns 12
```

C++14 (return type deduction)

```
1  // Deduce return type as `int`.
2  auto f(int i) {
3   return i;
4  }
5
6  template <typename T>
7  auto& f(T& t) {
8    return t;
9  }
10
11 // Returns a reference to a deduced type.
12 auto g = [](auto& x) -> auto& { return f(x); };
13 int y = 123;
14 int& z = g(y); // reference to `y`
```

C++14 (decltype(auto))

```cpp
const int x = 0;
auto x1 = x; // int
decltype(auto) x2 = x; // const int
int y = 0;
int& y1 = y;
auto y2 = y1; // int
decltype(auto) y3 = y1; // int&
int&& z = 0;
auto z1 = std::move(z); // int
decltype(auto) z2 = std::move(z); // int&&
```

C++14 (decltype(auto))

```cpp
// Note: Especially useful for generic code!

// Return type is `int`.
auto f(const int& i) {
 return i;
}

// Return type is `const int&`.
decltype(auto) g(const int& i) {
 return i;
}

int x = 123;
static_assert(std::is_same<const int&, decltype(f(x))>::value == 0);
static_assert(std::is_same<int, decltype(f(x))>::value == 1);
static_assert(std::is_same<const int&, decltype(g(x))>::value == 1);
```

C++14 (variable templates)

```
1 template<class T>
2 constexpr T pi = T(3.14159265358979323385);
3 template<class T>
4 constexpr T e  = T(2.71828182845904523353);
```

C++17

C++17 (class template argument deduction)

```
1 template <typename T = float>
2 struct MyContainer {
3   T val;
4   MyContainer() : val{} {}
5   MyContainer(T val) : val{val} {}
6   // ...
7 };
8 MyContainer c1 {1}; // OK MyContainer<int>
9 MyContainer c2; // OK MyContainer<float>
```

C++17 (declaring non-type template parameters with auto)

```cpp
template <auto... seq>
struct my_integer_sequence {
  // Implementation here ...
};

// Explicitly pass type `int` as template argument.
auto seq = std::integer_sequence<int, 0, 1, 2>();
// Type is deduced to be `int`.
auto seq2 = my_integer_sequence<0, 1, 2>();
```

C++17 (folding expressions)

```cpp
template <typename... Args>
bool logicalAnd(Args... args) {
    // Binary folding.
    return (true && ... && args);
}
bool b = true;
bool& b2 = b;
logicalAnd(b, b2, true); // == true

template <typename... Args>
auto sum(Args... args) {
    // Unary folding.
    return (... + args);
}
sum(1.0, 2.0f, 3); // == 6.0
```

C++17 (new rules for auto)

```
1 auto x1 {1, 2, 3}; // error: not a single element
2 auto x2 = {1, 2, 3}; // x2 is std::initializer_list<int>
3 auto x3 {3}; // x3 is int
4 auto x4 {3.0}; // x4 is double
```

C++17 (constexpr lambda)

```cpp
auto identity = [](int n) constexpr { return n; };
static_assert(identity(123) == 123);

constexpr auto add = [](int x, int y) {
  auto L = [=] { return x; };
  auto R = [=] { return y; };
  return [=] { return L() + R(); };
};

static_assert(add(1, 2)() == 3);

constexpr int addOne(int n) {
  return [n] { return n + 1; }();
}

static_assert(addOne(1) == 2);
```

C++17 (lambda capture `this` by value)

```cpp
struct MyObj {
  int value {123};
  auto getValueCopy() {
    return [*this] { return value; };
  }
  auto getValueRef() {
    return [this] { return value; };
  }
};
MyObj mo;
auto valueCopy = mo.getValueCopy();
auto valueRef = mo.getValueRef();
mo.value = 321;
valueCopy(); // 123
valueRef(); // 321
```

C++17 (structured bindings)

```cpp
using Coordinate = std::pair<int, int>;
Coordinate origin() {
  return Coordinate{0, 0};
}

const auto [ x, y ] = origin();
x; // == 0
y; // == 0

std::unordered_map<std::string, int> mapping {
  {"a", 1},
  {"b", 2},
  {"c", 3}
};

// Destructure by reference.
for (const auto& [key, value] : mapping) {
  // Do something with key and value
}
```

C++17 (constexpr if)

```cpp
template <typename T>
constexpr bool isIntegral() {
  if constexpr (std::is_integral<T>::value) {
    return true;
  } else {
    return false;
  }
}
static_assert(isIntegral<int>() == true);
static_assert(isIntegral<char>() == true);
static_assert(isIntegral<double>() == false);
struct S {};
static_assert(isIntegral<S>() == false);
```

C++17 (std::variant)

```cpp
std::variant<int, double> v{ 12 };
std::get<int>(v); // == 12
std::get<0>(v); // == 12
v = 12.0;
std::get<double>(v); // == 12.0
std::get<1>(v); // == 12.0
```

C++17 (std::optional)

```cpp
std::optional<std::string> create(bool b) {
  if (b) {
    return "C++ Serbia";
  } else {
    return {};
  }
}

create(false).value_or("empty"); // == "empty"
create(true).value(); // == "C++ Serbia"
// optional-returning factory functions are usable as conditions of while and if
if (auto str = create(true)) {
  // ...
}
```

C++17 (std::any)

```
std::any x {5};
x.has_value() // == true
std::any_cast<int>(x) // == 5
std::any_cast<int&>(x) = 10;
std::any_cast<int>(x) // == 10
```

C++17 (std::string_view)

```cpp
// Regular strings.
std::string_view cppstr {"foo"};
// Wide strings.
std::wstring_view wcstr_v {L"baz"};
// Character arrays.
char array[3] = {'b', 'a', 'r'};
std::string_view array_v(array, std::size(array));
```

C++17 (std::string_view)

```
1 std::string str {"   trim me"};
2 std::string_view v {str};
3 v.remove_prefix(std::min(v.find_first_not_of(" "), v.size()));
4 str; //  == "   trim me"
5 v; // == "trim me"
```

C++17 (std::apply)

```cpp
auto add = [](int x, int y) {
  return x + y;
};
std::apply(add, std::make_tuple(1, 2)); // == 3
```

C++17 (...)

- std::invoke
- std::filesystem
- std::byte
- parallel algorithms

C++20

C++20 (coroutines)

```
generator<int> range(int start, int end) {
  while (start < end) {
    co_yield start;
    start++;
  }

  // Implicit co_return at the end of this function:
  // co_return;
}

for (int n : range(0, 10)) {
  std::cout << n << std::endl;
}
```

C++20 (coroutines)

```
task<void> echo(socket s) {
  for (;;) {
    auto data = co_await s.async_read();
    co_await async_write(s, data);
  }

  // Implicit co_return at the end of this function:
  // co_return;
}
```

C++20 (coroutines)

```
1 task<int> calculate_meaning_of_life() {
2   co_return 42;
3 }
4
5 auto meaning_of_life = calculate_meaning_of_life();
6 // ...
7 co_await meaning_of_life; // == 42
```

C++20 (concepts)

```cpp
template <typename T>
  requires my_concept<T> // `requires` clause.
void f(T);

template <typename T>
concept callable = requires (T f) { f(); }; // `requires` expression.

template <typename T>
  requires requires (T x) { x + x; } // `requires` clause and expression on same line.
T add(T a, T b) {
  return a + b;
}
```

C++20 (concepts)

```
1 template <typename T>
2 concept C = requires(T x) {
3   {*x} -> typename T::inner; // the type of the expression `*x` is convertible to `T::inner`
4   {x + 1} -> std::same_as<int>; // the expression `x + 1` satisfies `std::same_as<decltype((x + 1))>`
5   {x * 1} -> T; // the type of the expression `x * 1` is convertible to `T`
6 };
```

C++20 (template syntax for lambdas)

```
1 auto f = []<typename T>(std::vector<T> v) {
2   // ...
3 };
```

C++20 (constexpr virtual functions)

```
1  struct X1 {
2    virtual int f() const = 0;
3  };
4
5  struct X2: public X1 {
6    constexpr virtual int f() const { return 2; }
7  };
8
9  struct X3: public X2 {
10   virtual int f() const { return 3; }
11 };
12
13 struct X4: public X3 {
14   constexpr virtual int f() const { return 4; }
15 };
16
17 constexpr X4 x4;
18 x4.f(); // == 4
```

C++20 (immediate functions)

```
1 consteval int sqr(int n) {
2   return n * n;
3 }
4
5 constexpr int r = sqr(100); // OK
6 int x = 100;
7 int r2 = sqr(x); // ERROR: the value of 'x' is not usable in a constant expression
8                 // OK if `sqr` were a `constexpr` function
```

C++20 (lambda capture of parameter pack)

```cpp
template <typename... Args>
auto f(Args&&... args){
    // BY VALUE:
    return [...args = std::forward<Args>(args)] {
        // ...
    };
}

template <typename... Args>
auto f(Args&&... args){
    // BY REFERENCE:
    return [&...args = std::forward<Args>(args)] {
        // ...
    };
}
```

C++20 (...)

Libraries:

- concepts library
- std::span
- std::is_constant_evaluated
- ranges

Modules

C++20 (ranges)

```cpp
#include <ranges>
#include <iostream>

int main()
{
    auto const ints = {0,1,2,3,4,5};
    auto even = [](int i) { return 0 == i % 2; };
    auto square = [](int i) { return i * i; };

    // "pipe" syntax of composing the views:
    for (int i : ints | std::views::filter(even) | std::views::transform(square)) {
        std::cout << i << ' ';
    }

    std::cout << '\n';

    // a traditional "functional" composing syntax:
    for (int i : std::views::transform(std::views::filter(ints, even), square)) {
        std::cout << i << ' ';
    }
}
```

Thank you

References:

- isocpp.org
- cppreference.com
- github.com/AnthonyCalandra/modern-cpp-features