C++17 and Beyond

Mark Isaacson

Roadmap

- What's coming in C++17 and C++20

- The surprising utility of `std::string_view`
- C++20's most deliciously scary feature: operator dot
- Making templates more accessible with `if constexpr`

What made it to C++17

- Filesystem TS
- Special Math TS
- Parallelism TS
- `if constexpr`
- `[[fallthrough]]`
- `[[nodiscard]]`
- logical operator type traits
- variable templates
- polymorphic allocators
- removed exception specifications
- Type deduction for constructors
- Stronger evaluation order
- Structured bindings

Library Fundamentals 1 TS:

- `std::not_fn`
- `std::string_view`
- `std::any` (unconstrained)
- `std::variant` (constrained)
- `std::optional`
- `std::shared_ptr for arrays`
- `std::uncaught_exceptions`
- `std::invoke`
- `std::apply`

C++20 Horizon

- Concurrency TS
- Transactional Memory TS
- Concepts TS
- Ranges TS
- Networking TS
- Library Fundamentals 2 TS
- Parallelism 2 TS
- Coroutine TS
- Numerics TS
- Reflection TS
- Modules TS
- Operator dot

Longer?

- Concurrency 2 TS
- 2D Graphics TS
- Automatic comparison operators

Now the fun stuff

- What's coming in C++17 and C++20

- The surprising utility of `std::string_view`
- C++17's most deliciously scary feature: operator dot
- Making templates more accessible with `if constexpr`

Subtle Bug

```cpp
// Foo.h
#include <string>
extern const string kFoo;
```

```cpp
// Foo.cpp
#include "Foo.h"
const string kFoo = "foo";
```

```cpp
// main.cpp
#include "Foo.h"
const string kBar = kFoo;

int main() { return 0; }
```

Good News

- This is automatically detectable
- Best practices avoid this entirely (for string constants)

```
ASAN_OPTIONS=check_initialization_order=1:strict_init_order=1:
detect_odr_violation=1
```

Pre-C++17

```cpp
// Foo.h
extern const char* const kFoo;
```

```cpp
// Foo.cpp
#include "Foo.h"
const char* const kFoo = "foo";
```

```cpp
// main.cpp
#include "Foo.h"
const char* const kBar = kFoo;

int main() { return 0; }
```

Post-C++17

```cpp
// Foo.h
#include <string_view>
const constexpr string_view kFoo = "foo";

// main.cpp
#include "Foo.h"
const constexpr string_view kBar = kFoo;

int main() { return 0; }
```

Safety & Convenience

- It knows its `size()`
- It works cleanly with the STL
- String comparisons are always intuitive

```
const char* const kFoo = "foo";
const constexpr string_view kFoo2 = "foo";

assert(kFoo == "foo");  // Oops
assert(kFoo2 == "foo"); // Ok
assert(kFoo2 == kFoo);  // Ok
```

Performance

```
void foo(const string& x);
void bar(string_view x);
```

```
string a = "hello";
const char* const b = "world";
foo(a);
foo(b);
bar(a);
bar(b);
```

Performance

```
// main
string x = "hello";
Widget w;
w.foo(x);
```

```
struct Widget {
  unordered_map<string, int> myMap;
  void foo(/* ??? */ intermediate) {
    return
      myMap.find(intermediate) !==
      myMap.end();
  }
};
```

Performance

```
// main
string x = "hello";
Widget w;
w.foo(x);
```

```
struct Widget {
  unordered_map<string, int> myMap;
  void foo(const string& intermediate) {
    return
      myMap.find(intermediate) !==
      myMap.end();
  }
};
```

Performance

```
// main
string x = "hello";
Widget w;
w.foo(x);
```

```
struct Widget {
  unordered_map<string, int> myMap;
  void foo(string_view intermediate) {
    return
      myMap.find(intermediate) !==
      myMap.end();
  }
};
```

...big caveat

I *must* explain the dangers

Operator Dot

```
struct IntProxy {
  int& operator.() {
    return value;
  }
  int value = 0;
};
```

```
IntProxy x;
x = 5;
x += 5;
cout << x << endl;
```

Operator Dot

```
struct IntProxy {
  int& operator.() {
    return value;
  }
  int value = 0;
};
```

```
IntProxy x;
x = 5;
x += 5;
cout << x << endl;
```

"Lowers" to `x.operator+=(5);`

Operator Dot

```cpp
struct IntProxy {
  int& operator.() {
    return value;
  }
  string toString() {
    return to_string(value);
  }
  int value = 0;
};
```

```cpp
IntProxy x;
x = 5;
x += 5;
cout << x << endl;
string msg =
  "The magic number is " +
  x.toString();
```

Contracts w/ Operator Dot

Goal: an `int` with the invariant: `0 <= x <= 100`

```
BoundedInt x = 5;
x += 5; //Ok
x = 1000; //Error, exceeded bound of 100
```

Contracts w/ Operator Dot

```
struct BoundedInt {
  int& operator.() {
    return val;
  }
  int val = 0;
};
```

Preconditions w/ Operator Dot

```
struct BoundedInt {
  int& operator.() {
    return val;
  }
  int val = 0;
};
```

Preconditions w/ Operator Dot

```
struct BoundedInt {
  int& operator.() {
    if (val < 0 || val > 100)
      throw exception{"BoundsError"};
    return val;
  }
  int val = 0;
};
```

End of Life Condtions

```
struct BoundedInt {
  ~BoundedInt() {
    if (val < 0 || val > 100)
      LOG(ERROR) << "BoundsError";
  }
  int& operator.() {return val;}
  int val = 0;
};
```

Postconditions w/ Operator Dot

```
struct BoundedInt {
  Impl operator.()
    {return Impl{val};}
  int val = 0;
};

// Example use:
BoundedInt x = 5;
x += 5; //Ok
x = 1000; //Error
```

```
struct Impl {
  Impl(int& x): ref{x} {};
  ~Impl() {
    if (ref < 0 || ref > 100)
      LOG(ERROR) << "BoundsError";

  }
  int& operator.() {return ref;}
  int& ref;
};
```

Postconditions w/ Operator Dot

```
struct BoundedInt {
  Impl operator.()
    {return Impl{val};}
  int val = 0;
};

// Example use:
BoundedInt x = 5;
x += 5; //Ok
x = 1000; //Error
```

Annotations: 1 → `x += 5; //Ok`; 2 → `{return Impl{val};}`

```
struct Impl {
  Impl(int& x): ref{x} {};
  ~Impl() {
    if (ref < 0 || ref > 100)
      LOG(ERROR) << "BoundsError";

  }
  int& operator.() {return ref;}
  int& ref;
};
```

Annotations: 3 → `int& operator.() {return ref;}`; 4 → `~Impl() {`

Postconditions w/ Operator Dot

```
struct BoundedInt {
  Impl operator.()
    {return Impl{val};}
  int val = 0;
};

// Example use:
BoundedInt x = 5;
x += 5; //Ok
x = 1000; //Error
```

```
struct Impl {
  Impl(int& x): ref{x},copy{x} {};
  ~Impl() {
    if (copy < 0 || copy > 100)
      LOG(ERROR) << "BoundsError";
    else { ref = copy; }
  }
  int& operator.() {return copy;}
  int& ref; int copy;
};
```

Operator Dot

Potential uses I won't outline:

- Smart references
- Pimpl idiom
- Proxies/Mixins (add operators) - see my other talk :)

Operator Dot vs Inheritance

- Can extend primitive types (like `int`)
- No notion of `final` or `protected`
- Wrapped type can't call Wrapper's functions (no NVI pattern)

Compile-time branching

```
template<typename T>
struct default_delete {
  void operator()(T* ptr) const { delete ptr; }
};

template<typename T>
struct default_delete<T[]> {
  template<typename U>
  void operator()(U* ptr) const { delete[] ptr; }
};
```

Compile-time branching

```
template<typename T>
struct default_delete {
  void operator()(T* ptr) cons    elete ptr; }
};

template<typename T>
struct default_delete<T[]> {
  template<typename U>
  void operator()(U* ptr) const { delete[] ptr; }
};
```

Compile-time branching

```
template<typename T>
struct default_delete {
  template<typename U> void operator()(U* ptr) const {
    if (is_array<T>::value) {
      delete[] ptr;
    } else {
      delete ptr;
    }
  }
};
```

Compile-time branching

```
constexpr bool example1 = conjunction(true); // true
constexpr bool example2 = conjunction(false); // false
constexpr bool example3 = conjunction(true, true); // true
constexpr bool example4 = conjunction(true, false, true); // false
```

Compile-time branching

```
template<typename Bool>
constexpr bool conjunction(Bool&& b) {
  return b;
}
template<typename Bool, typename... Rest>
constexpr bool conjunction(Bool&& b, Rest&&... rest) {
  return b && conjunction(std::forward<Rest>(rest)...);
}
constexpr bool enabled = conjunction(true, false, true);
```

Compile-time branching

```
template<typename Bool>
constexpr bool conjunction(Bool&& b) {
  return b;
}
template<typename Bool, typename... Rest>
constexpr bool conjunction(Bool&& b, Rest&&... rest)
  return b && conjunction(std::forward<Rest>(rest)...);
}
constexpr bool enabled = conjunction(true, false, true);
```

Compile-time branching

```
template<typename Bool, typename... Rest>
constexpr bool conjunction(Bool&& b, Rest&&... rest) {
  if constexpr(sizeof...(Rest)) {
    return b && conjunction(std::forward<Rest>(rest)...);
  } else {
    return b;
  }
}
constexpr bool enabled = conjunction(true, false, true);
```

Why `if constexpr`?

```
template<typename Bool, typename... Rest>
constexpr bool conjunction(Bool&& b, Rest&&... rest) {
  if (sizeof...(Rest)) {
    return b && conjunction(std::forward<Rest>(rest)...);
  } else {
    return b;
  }
}
constexpr bool enabled = conjunction(true);
```

Fold Expressions

```
template<typename Bool, typename... Rest>
constexpr bool conjunction(Bool&& b, Rest&&... rest) {
  return b && ...;
}
constexpr bool enabled = conjunction(true, false, true);
```

if constexpr vs Concepts

- if constexpr => Template branching
- Concepts => Template constraints

Design by Introspection

```
template<typename ITR>
void advance(ITR& itr, size_t n) {
  using C = typename iterator_traits<ITR>::iterator_category;
  if constexpr(is_same_v<C, random_access_iterator_tag>) {
    itr += n;
  } else {
    for (auto i = 0u; i < n; ++i) ++itr;
  }
}
```

if constexpr in C++14

```
IF_CONSTEXPR(true, { cout << "hi"; });
```

```
template<bool b, typename F>
struct CallIfImpl {
  CallIfImpl(F func) {}
};
template<typename F>
struct CallIfImpl<true, F> {
  CallIfImpl(F func) {

    func(                 );
  }
};
```

if constexpr in C++14

```
IF_CONSTEXPR(true, { cout << "hi"; });

template<bool b, typename F>
void callIf(F&& func) {
  CallIfImpl<b, F>{forward<F>(func)};
}
```

```
template<bool b, typename F>
struct CallIfImpl {
  CallIfImpl(F func) {}
};
template<typename F>
struct CallIfImpl<true, F> {
  CallIfImpl(F func) {

    func(                );
  }
};
```

if constexpr in C++14

```
IF_CONSTEXPR(true, { cout << "hi"; });

template<bool b, typename F>
void callIf(F&& func) {
  CallIfImpl<b, F>{forward<F>(func)};
}

#define IF_CONSTEXPR(condition, code) \
  callIf<condition>([&](auto) code);
```

```
template<bool b, typename F>
struct CallIfImpl {
  CallIfImpl(F func) {}
};
template<typename F>
struct CallIfImpl<true, F> {
  CallIfImpl(F func) {
    auto dummyArgument = true;
    func(dummyArgument);
  }
};
```

Review

- `string_view` provides correctness, efficiency, and interoperability
- Operator dot is black magic
- `if constexpr` makes branching templates easy

return 0;

Mark Isaacson
http://modernmaintainablecode.com