Modern C++ (oh come on, C++17 is five years ago)

谢天 2022.10

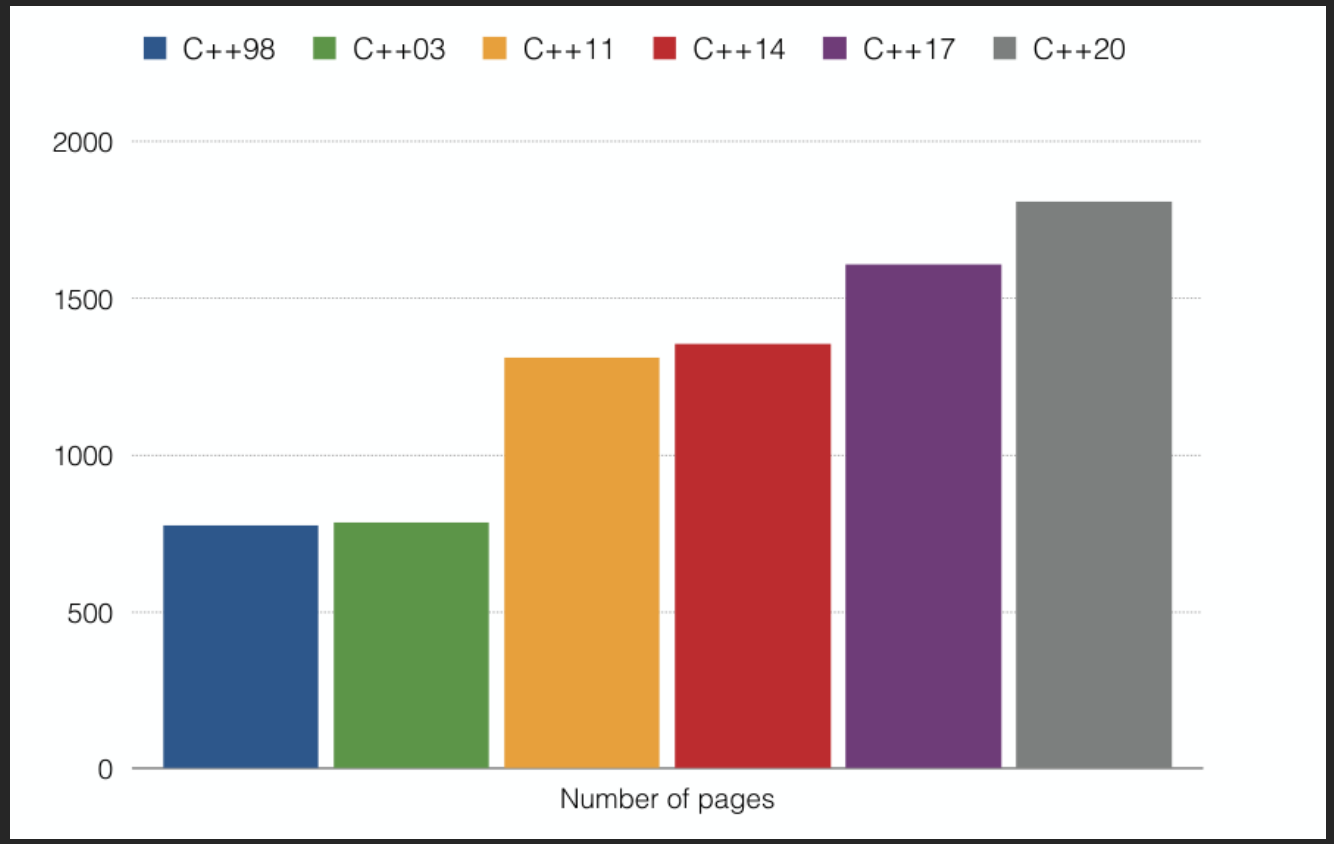
C++ timeline

We will mainly focus on 11 – 17 in this talk, with a glimpse at C++20 “the big four”

- **C++98, the first ISO C++ standard**
  - Language: namespaces, named casts, bool, dynamic_cast
  - Library: the STL (containers and algorithms), string, bitset
- **C++11**
  - Language: memory model, auto, range-for, constexpr, lambdas, user-defined literals, ...
  - Library: threads and locks, future, unique_ptr, shared_ptr, array, time and clocks, random numbers, unordered containers (hash tables), ..
- **C++14**
  - Language: generic lambdas, local variables in constexpr functions, variable templates, digit separators, ...
  - Library: user-defined literals, ...
- **C++17**
  - Language: structured bindings, template argument deduction from constructors, ...
  - Library: file system, scoped_lock, shared_mutex (reader-writer locks), any, variant, optional, string_view, parallel algorithms, ...
- **C++20**
  - Language: concepts, modules, coroutines, three-way comparisons, improved support for compile-time computation, ...
  - Library: concepts, ranges, dates and time zones, span, formats, improved concurrency and parallelism support, ...

C++ timeline

We will mainly focus on 11 – 17 in this talk, with a glimpse at C++20 “the big four”



Agenda

You can learn it by yourself!

- This talk is not
  - a complete guide to Cpp
  - template meta programming, although it is important and interesting
  - ranting Cpp because it is complex and disgusting
  - detailed Cpp grammar
  - repeating interview questions about Cpp that can be found everywhere
- This talk is about
  - important features in the language that I think we (as engine devs) should know
  - AND important features in the language that I think will help us develop a better game engine

Outlines

01 (Crazy) Modern Cpp

02 From 11 to 17

03 What’s Next

04 Tools We Love

01

(Crazy) Modern Cpp

A comparison to Python

(Crazy) Modern Cpp

Let’s see if it’s as simple as Python, create variables

x = 42

y = 2.71

z = 'hello'

```
auto x = 42;
auto y = 2.71;
auto z = "hello";
```

(Crazy) Modern Cpp

define a (Parametric polymorphic) function

```
def f(x):
    return x + x

x = f(21) # x == 42
y = f('hello') # y == 'hellohello'
```

```
auto f(auto &&x)
{
    return x + x;
}

// x == 42
auto x = f(21);
// y == "hellohello"
auto y = f("hello"s);
```

(Crazy) Modern Cpp

loop over a container

```
c = [1, 2, 3, 4]
for x in c:
    print(x)
```

```cpp
auto c = std::array{1, 2, 3, 4};

for (auto x : c)
    std::cout << x;
```

(Crazy) Modern Cpp

function with multiple return values

```
def f(x, y):
    return x + x, y + y

x, y = f(21, 'hello’)
# x == 42, y == 'hellohello'
```

```cpp
auto f(auto &&x, auto &&y)
{
    return std::tuple{x + x, y + y};
}

auto [x, y] = f(21, "hello"s);
// x == 42, y == "hellohello"
```

(Crazy) Modern Cpp

latent typing

```
class A:
    def f(self):
        pass

class B:
    pass

def g(x):
    if hasattr(x, 'f'):
        return 42
    else:
        return 'hello’

x = g(A()) # type(x) == int
y = g(B()) # type(y) == str
```

```
struct A
{
    auto f() {}
};
struct B
{
};

auto g(auto &&x)
{
    if constexpr (requires { x.f(); })
        return 42;
    else
        return "hello"s;
}

// std::same_as<decltype(x), int> == true
auto x = g(A{});
// std::same_as<decltype(y), std::string> == true
auto y = g(B{});
```

(Crazy) Modern Cpp

type inference for polymorphic recursive functions

```
// no type annotation for cat()
auto cat(auto &&x, auto && ... p)
{
    auto y = [&]
    {
        if constexpr (requires { std::string{x}; })
            return std::string{x};
        else
            return std::to_string(x);
    }();
    if constexpr (sizeof ... (p) == 0)
        return y;
    else
        return y + cat(std::forward<decltype(p)>(p) ... );
}

auto s = cat(0, " is zero, ", 1, " is one, pi = ", 3.14);
// s == "0 is zero, 1 is one, pi = 3.14"
```

02

From 11 to 17

Important features that game dev will benefit from (in my opinion)

RAII and the rule of three/five/zero

Maybe it is the most import part in C++

- To avoid leaks and the complexity of manual resource management. C++'s language-enforced constructor/destructor symmetry mirrors the symmetry inherent in resource acquire/release function pairs such as fopen/fclose, lock/unlock, and new/delete.
- Whenever you deal with a resource that needs paired acquire/release function calls, encapsulate that resource in an object that enforces pairing for you -- acquire the resource in its constructor, and release it in its destructor.

RAII and the rule of three/five/zero

Example, bad

```
void send(X* x, string_view destination)
{
    auto port = open_port(destination);
    my_mutex.lock();
    //  ...
    send(port, x);
    //  ...
    my_mutex.unlock();
    close_port(port);
    delete x;
}
```

In this code, you have to remember to unlock, close_port, and delete on all paths, and do each exactly once.

Further, if any of the code marked ... throws an exception, then x is leaked and my_mutex remains locked.

RAII and the rule of three/five/zero

Example, better

```
// x owns the X
void send(unique_ptr<X> x, string_view destination)
{
    // port owns the PortHandle
    Port port{destination};

    // guard owns the lock
    lock_guard<mutex> guard{my_mutex};
    //  ...
    send(port, x);
    //  ...
}
// automatically unlocks my_mutex and deletes the pointer in x
```

Now all resource cleanup is automatic, performed once on all paths whether or not there is an exception.

As a bonus, the function now advertises that it takes over ownership of the pointer.

RAII and the rule of three/five/zero

If you can avoid defining default operations, do

```
struct Named_map {
public:
    // ... no default operations declared ...
private:
    string name;
    map<int, int> rep;
};

Named_map nm;          // default construct
Named_map nm2 {nm};    // copy construct
```

Since std::map and string have all the special functions, no further work is needed.

This is known as "the rule of zero".

RAII and the rule of three/five/zero

If you define or =delete any copy, move, or destructor function, define or =delete them all

```
// bad: incomplete set of copy/move/destructor operations
struct M2 {
public:
    //  ...
    //  ...  no copy or move operations  ...
    ~M2() { delete[] rep; }
private:
    pair<int, int>* rep;   // zero-terminated set of pairs
};

void use()
{
    M2 x;
    M2 y;
    //  ...
    x = y;    // the default assignment
    //  ...
}
```

The semantics of copy, move, and destruction are closely related, so if one needs to be declared, the odds are that others need consideration too.

This is known as "the rule of five".

Memory Model

One of the most important part in C++11

- C++98/03: does not have concurrency
- C++11 have multithreading support
  - **Memory model, atomics API**
  - Language support: TLS, static init, termination, lambda function
  - Library support: thread start, join, terminate, mutex, condition variable
  - Advanced abstractions: basic futures, asyncs

Memory Model

A real-world example

CppCon 2014: Jeff Preshing "How Ubisoft Develops Games for Multicore - Before and After C++11"

https://www.youtube.com/watch?v=X1T3IQ4N-3g

Memory Model

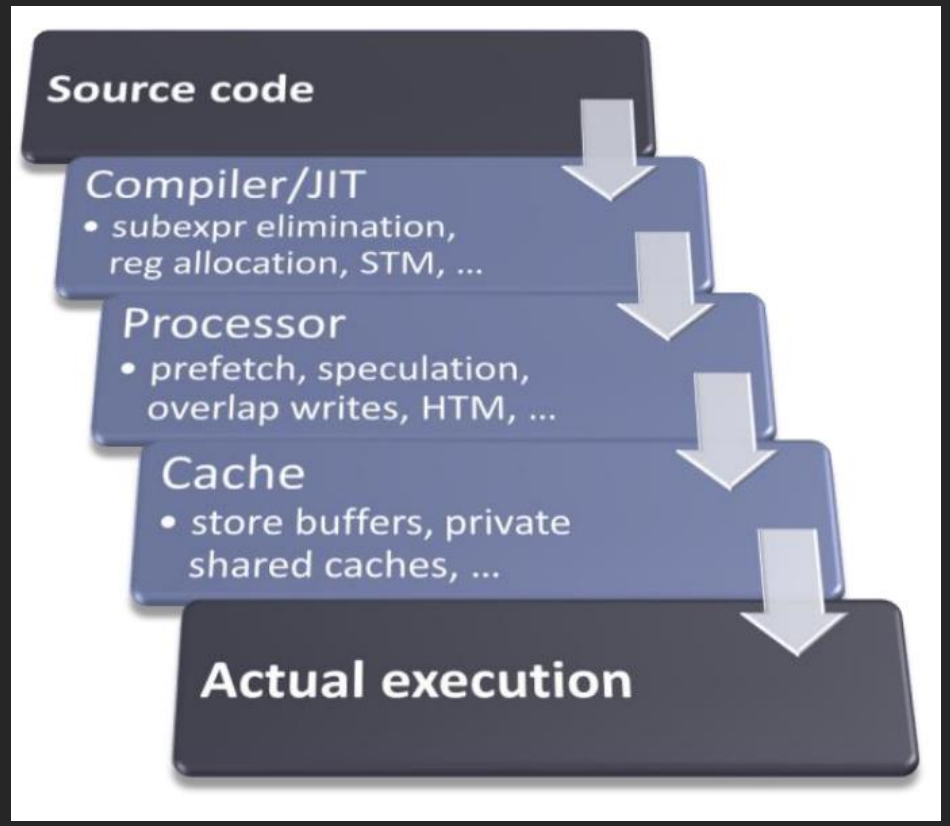
So what is memory model?

- an efficient low level-model of modern hardware as a foundation for concurrency
- Describes how memory reads and writes may appear to be executed relative to their program order
- Stroustrup: represents a contract between the implementers and the programmers to ensure that most programmers do not have to think about the details of modern computer hardware

```
typedef enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
```

Memory Model

What you see is not what you excute



C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2 - YouTube

Does your computer execute the program you wrote?
NO !!!!!!

Compiler optimization,
processor OoO execution
cache coherency

Compiler/processor/cache says:
“No, it’s much better to execute a different program.
Hey, don’t complain. It’s for your own good.
You really wouldn’t want to execute that dreck you actually wrote.”

As-if rule:
Thou shalt not modify the behavior of a single-threaded program.

Memory Model

Valid Single Thread Optimizition

```
//reordering
int A = 0, B = 0;
void foo()
{
    A = B + 1;
    B = 1;
}

// g++ -std=c++11 -O2 -S test.cpp
movl  B(%rip), %eax    // store B to eax
movl  $1, B(%rip)      // B = 1
addl  $1, %eax         // A = B + 1
movl  %eax, A(%rip)
```

```
// Speculative execution
if (cond)
    x = 42;

// code after optimizition (maybe)
r1 = x;      // read what's there
x = 42;      // oops: optimistic write is not
conditional
if (!cond)   // check if we guessed wrong
    x = r1; // oops: back-out write is not SC
```

Memory Model

Dekker’s and Peterson’s Algorithms

```
// Thread 1:
flag1 = 1; // a: declare intent

if( flag2 ≠ 0 ) // b
    // resolve contention
else
    // enter critical section
```

```
// Thread 2:
flag2 = 1; // c: declare intent

if( flag1 ≠ 0 ) // d
    // resolve contention
else
    // enter critical section
```

Can both threads enter the critical region?

- Maybe: If a can pass b, or c can pass d, this breaks. Also, write is slow, need flush to global memory
- Solution 1 (good): Use a suitable atomic type (e.g., Java/ .NET “volatile”, C++11 std::atomic<>) for the flag variables.
- Solution 2 (good?): Use system locks.

Memory Model

Optimizations

What the compiler knows:

- All memory operations in this thread and exactly what they do, including data dependencies.
- How to be conservative enough in the face of possible aliasing.

What the compiler doesn't know:

- Which memory locations are "mutable shared" variables and could change asynchronously due to memory operations in another thread.
- How to be conservative enough in the face of possible sharing.

Solution: Tell it.

- Somehow identify the operations on "mutable shared" locations (or equivalent information, but identifying shared variables is best).

Memory Model

One of the most important aspect of C++0x /C1x is almost invisible to most programmers

- Locks and atomic operations communicate non-atomic writes between two threads
  - Writes are explicitly communicated
  - The mechanism is acquire and release
- Data races cause undefined behavior
  - A non-atomic write to a memory location in one thread
  - AND a non-atomic read from or write to that same location in another thread
  - AND with no happens-before relations between them
- Some optimizations are no longer legal

Memory Model

Consistency models

Sequentially consistent

- What is observed is consistent with a sequential ordering of all events in the system
- the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program, but comes with a very heavy cost

Weaker models

- More complex to code for some, but very efficient

So

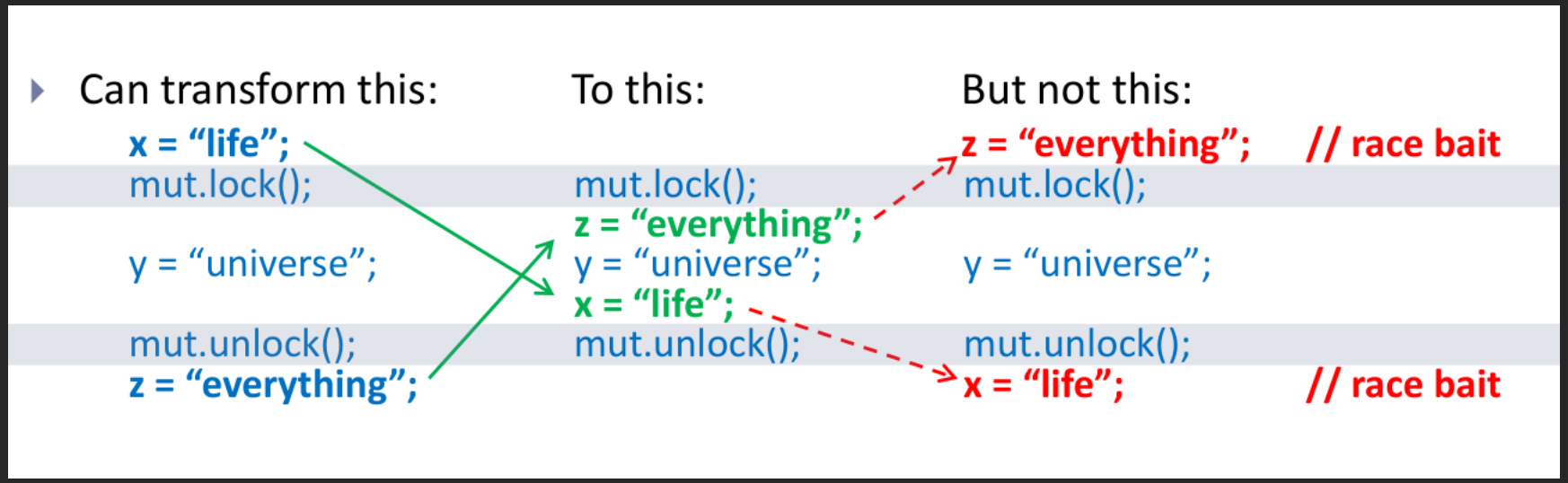
- Default is sequential consistency
- But allow weaker semantics explicitly

Memory Model

"One-way barriers": An "acquire barrier" and a "release barrier."

Acquire and Release

- More precisely: A release store makes its prior accesses visible to a thread performing an acquire load that sees (pairs with) that store.

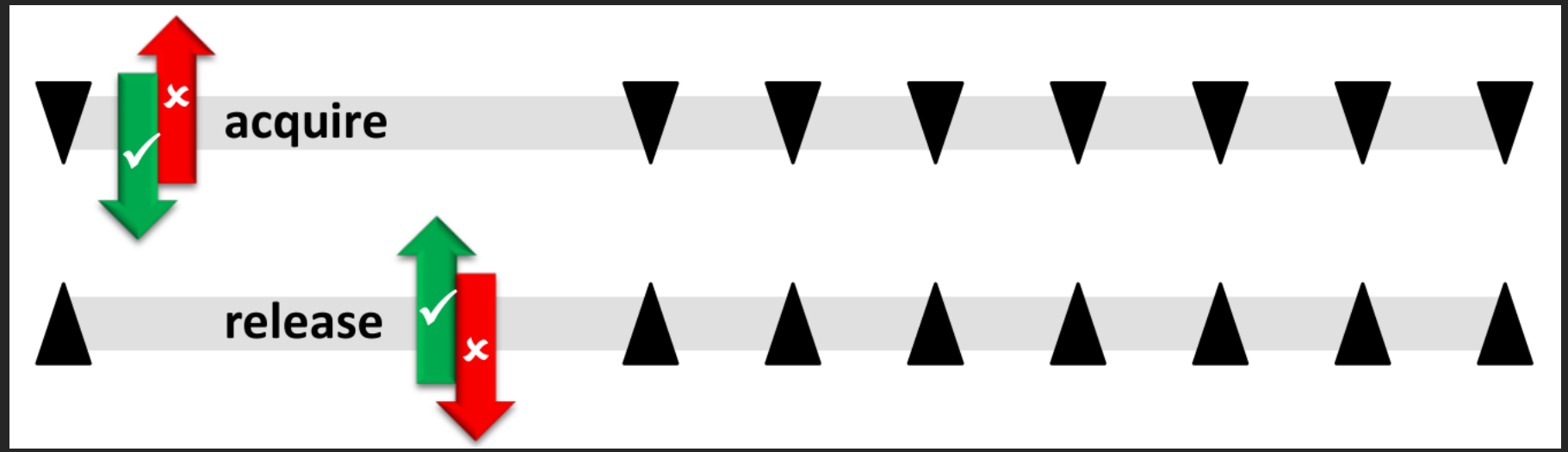


C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2 - YouTube

Memory Model

"One-way barriers": An "acquire barrier" and a "release barrier."

- Memory synchronization actively works against important modern hardware optimizations. Want to do as little as possible.(e.g. Cache coherence is expensive)



C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2 - YouTube

Memory Model

A simple and valid example

```
int x = 0;
atomic<int> y = 0;
```

```
// Thread 1 :
x = 17;
y.store(1, memory_order_release);
```

```
// Thread 2 :
while (y.load(memory_order_acquire) ≠ 1)
    continue;

assert(x == 17);
```

Memory Model

So what the hell happened?

Code Generation

	Load		Store		CAS
	Ordinary	SC Atomic	Ordinary	SC Atomic	
x86/x64	mov	**mov**	mov	**xchg**	**cmpxchg**
IA64	ld	**ld.acq**	st	**st.rel; mf**	**cmpxchg.rel; mf**
POWER	ld	**sync;** ld; cmp; bc; **isync**	st	**sync**; st	**sync; _loop: lwarx**; cmp; bc _exit; **stwcx.; bc _loop; isync;** _exit:
ARM v7	ldr	ldr; **dmb**	str	**dmb**; str; **dmb**	**dmb; _loop:** ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; **bne _loop;** isb
ARM v8	ldr	**ldra**	str	**strl**	

C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2 - YouTube

Memory Model

Don't try to use volatile for synchronization

- volatile does not provide atomicity, does not synchronize between threads, and does not prevent instruction reordering (neither compiler nor hardware). It simply has nothing to do with concurrency.
- Use atomic types where you might have used volatile in some other language. Use a mutex for more complicated examples.
- but, in some compiler(msvc) voliate has the sematic of acquire load and release store….

volatile only to talk to non-C++ memory

- volatile is used to refer to objects that are shared with "non-C++" code or hardware that does not follow the C++ memory model.

Memory Model

Key Takeaway

- Don’t write a race condition or use non-default atomics and your code will do what you think.
- volatile (Java, .NET) == atomic (C, C++) != volatile (C, C++)
- Relaxed: Don’t do it.

Guaranteed copy elision

This code can not compile before C++17

```
struct NonMoveable
{
    NonMoveable(int);
    // no copy or move constructor:
    NonMoveable(const NonMoveable&) = delete;
    NonMoveable(NonMoveable&&) = delete;

    std::array<int, 1024> arr;
};

NonMoveable make()
{
    return NonMoveable(42);
}

// construct the object:
auto largeNonMovableObj = make();
```

C++17 the constructors are not required - because the object largeNonMovableObj will be constructed in place.

allow returning objects that are not movable/copyable - because we could now skip copy/move constructors. Useful in factories.

improve code portability, support ‘return by value’ pattern rather than use ‘output params.’

Guaranteed copy elision

Speed up!

- Return containers by value (relying on move or copy elision for efficiency)
- For "out" output values, prefer return values to output parametersReason

```
vector<int> get_large_vector()
{
    return ...;
}

// return by value is ok,
// most modern compilers will do copy elision
auto v = get_large_vector();
```

```
// OK: return pointers to elements with the value x
vector<const int*> find_all(const vector<int>&, int x);

// Bad: place pointers to elements with value x in-out
void find_all(const vector<int>&, vector<const int*>& out, int x);
```

Guaranteed copy elision

Don’t try to be “smart”…

- Don't return std::move(local)

```
S f()
{
    S result;
    return std::move(result);
}

S f()
{
    S result;
    return result;
}
```

With guaranteed copy elision, it is now almost always a pessimization to expressly use std::move in a return statement.

Polymorphic memory resource

Better alloctor!

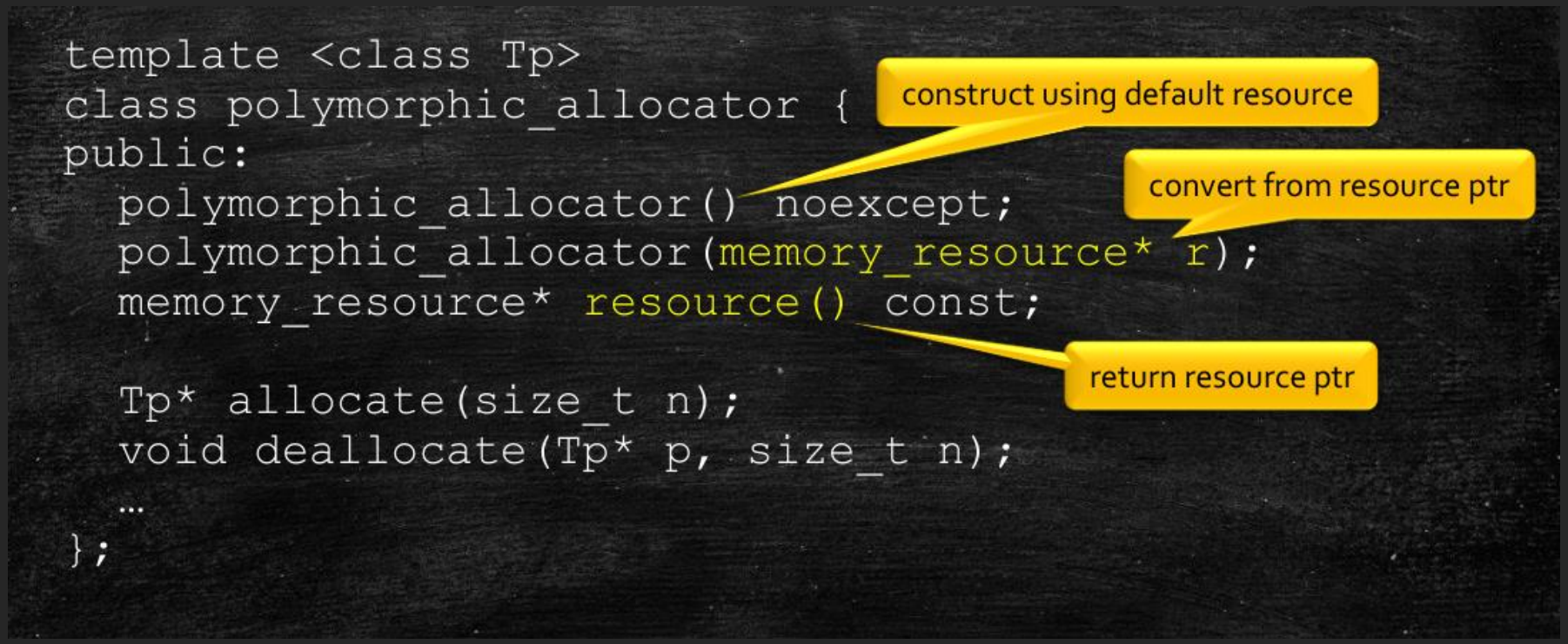
- Allocator in C++ is very complex and hard to use before C++17, C++17 supports a simpler allocator model

- **std::pmr::memory_resource** is a simple base class with allocate and deallocate member functions.
- **std::pmr::polymorphic_allocator** is a thin wrapper around a pointer to pmr::memory_resource for backwards-compatibility with the C++11 (and C++03) allocator model.
- std::pmr::list<T> is an alias for std::list<T,std::pmr::polymorphic_allocator<T>>. Similarly for the other allocator-aware standard containers.

```
namespace std::pmr
{
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

Polymorphic memory resource

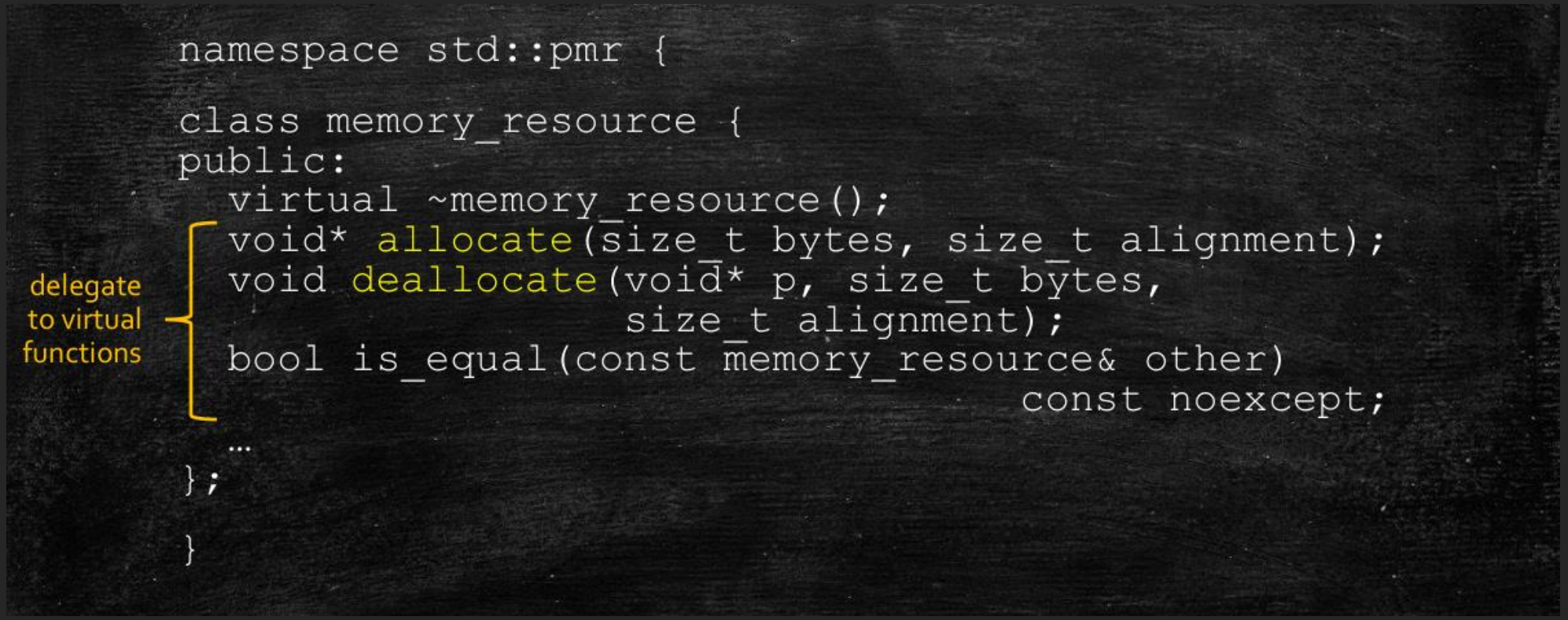
So what is a polymorphic_allocator



From [CppCon 2017: Pablo Halpern “Allocators: The Good Parts” - YouTube]

Polymorphic memory resource

So what is a memory_resource



From [CppCon 2017: Pablo Halpern “Allocators: The Good Parts” - YouTube]

Polymorphic memory resource

C++17 Defines several standard resources:

- new_delete_resource(): Allocates using ::operator new
- null_memory_resource(): Throws on allocation
- synchronized_pool_resource: Thread-safe pools of similar- sized memory blocks.
- unsynchronized_pool_resource: Non-thread-safe pools of similar-sized memory blocks.
- monotonic_buffer_resource: Super-fast, non-thread-safe allocation into a buffer with do-nothing deallocation.

Polymorphic memory resource

Getting it right – memory resources

- Creating a new memory resource is as easy as deriving from pmr::memory_resource.
- Override do_allocate() and do_deallocate() to do the real work.
- Override do_is_equal() to test for equality.
  - In most cases return this == &other (identity equality) is sufficient
  - If there is no resource-specific state, return true is correct.
  - In rare cases, something more complex is needed

Polymorphic memory resource

Let’s print all the allocate

```
class print_resource : public std::pmr::memory_resource
{
private:
    void *do_allocate(std::size_t bytes, std::size_t alignment) override{
        std::cout << "Allocating " << bytes << '\n';
        return std::pmr::new_delete_resource()->allocate(bytes, alignment);
    }
    void do_deallocate(void *p, std::size_t bytes, std::size_t alignment) override {
        std::cout << "Deallocating " << bytes << ": '";
        for (std::size_t i = 0; i < bytes; ++i)
        {
            std::cout << *(static_cast<char *>(p) + i);
        }
        std::cout << "'\n";
        return std::pmr::new_delete_resource()->deallocate(p, bytes, alignment);
    }
    bool do_is_equal(const std::pmr::memory_resource &other) const noexcept override {
        return std::pmr::new_delete_resource()->is_equal(other);
    }
};
```

Polymorphic memory resource

how to use it

```
int main()
{
    print_resource mem;
    std::pmr::set_default_resource(&mem);
    std::pmr::vector<char> a{'a', 'b', 'c', 'd'};
    std::cout << "exiting main\n";
}
```

```
Allocating 4
exiting main
Deallocating 4: 'abcd'
```

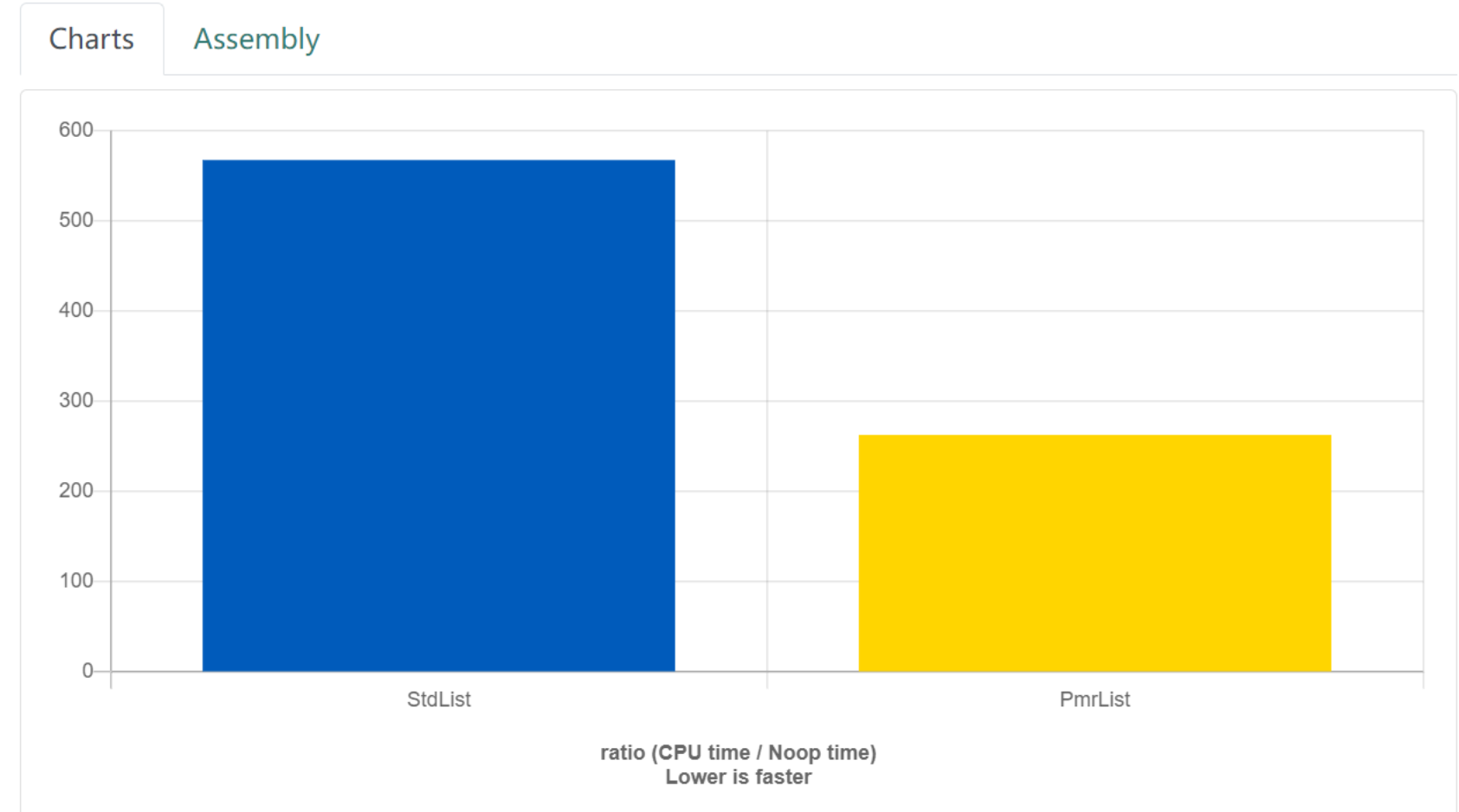
Polymorphic memory resource

Just a quick bench!

```
void StdList(benchmark::State &state)
{
    for (auto _ : state)
    {
        std::list<int> list{1, 2, 3, 4, 5, 6, 7, 8, 9,
10};
        benchmark::DoNotOptimize(list);
    }
}
void PmrList(benchmark::State &state)
{
    std::array<std::byte, 1024> buffer;
    for (auto _ : state)
    {
        std::pmr::monotonic_buffer_resource mem_resource(buffer.data(), buffer.size());
        std::pmr::list<int> list({1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, &mem_resource);
        benchmark::DoNotOptimize(list);
    }
}
```

Polymorphic memory resource

https://quick-bench.com/q/gyRE6p5EdZ-2P0XOAUwO99JRN7g#



Polymorphic memory resource

You can chain memory resources one by one

- Each memory resource has an upstream counterpart. If the resource runs out of memory, it tries to allocate more memory from upstream.
- Fixed-size vs. dynamic storage for allocators
- Customization of error-handling
- Combination of different allocation strategies
- Injection points for special purpose allocators for debugging and profiling

```
// fixed size buffer
std::aligned_storage_t<42> buffer;
std::pmr::monotonic_buffer_resource alloc(&buffer, 42, std::pmr::null_memory_resource());

// dynamically growing
std::aligned_storage_t<42> buffer;
std::pmr::monotonic_buffer_resource alloc(&buffer, 42, std::pmr::new_delete_resource());
```

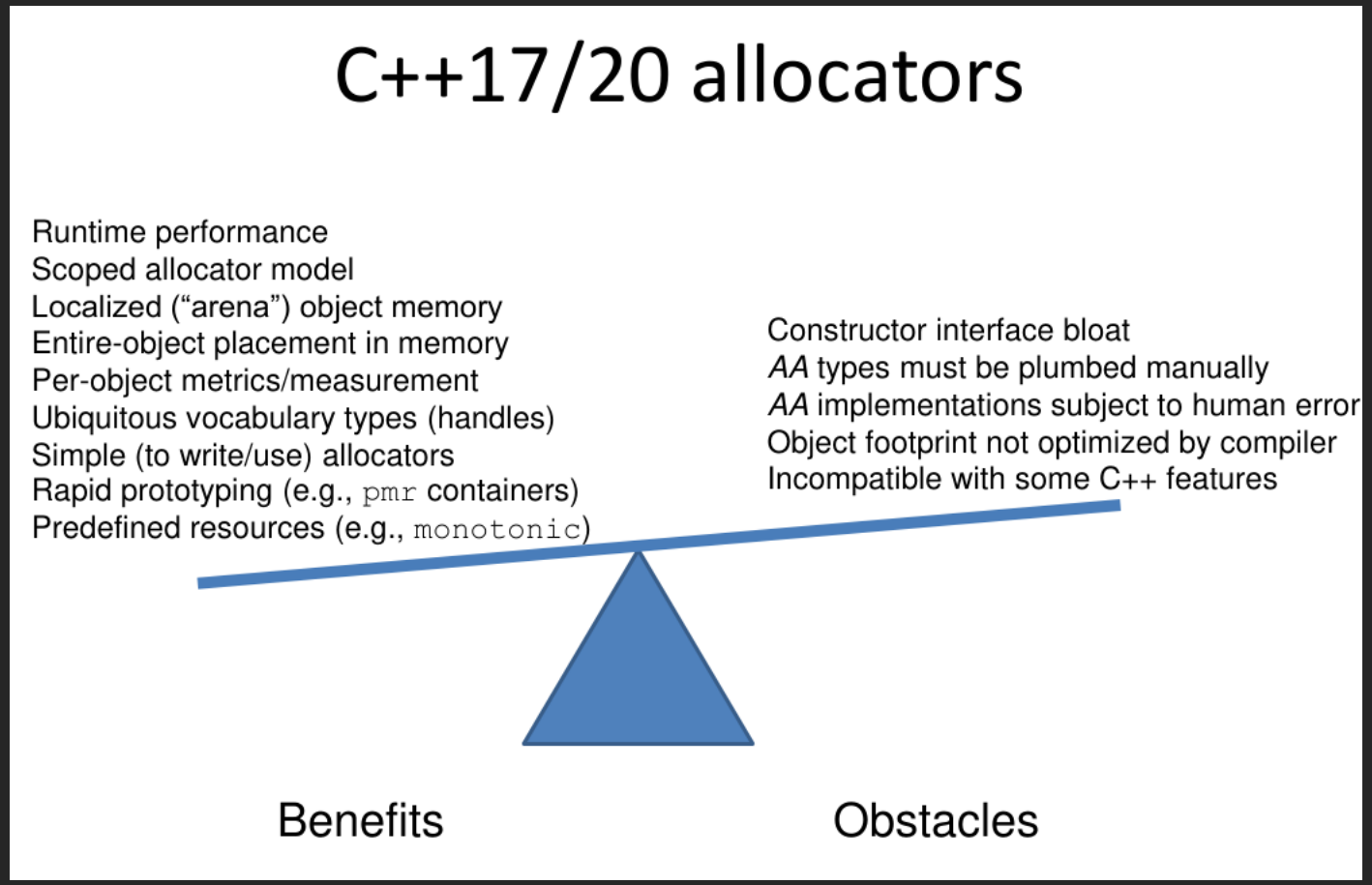
Polymorphic memory resource

Creating Allocator-Aware Types

```
struct S
{
    std::pmr::string str;
    using allocator_type = std::pmr::polymorphic_allocator<>;

    // default constructor, delegate to aa constructor
    S() : S(allocator_type{}) {}
    explicit S(allocator_type alloc)
        : str("Hello long string", alloc) {}

    S(const S &other, allocator_type alloc = {}): str(other.str, alloc) {}
    S(S &&) = default;
    S(S &&other, allocator_type alloc): str(std::move(other.str), alloc) {}
    S &operator=(const S &rhs) = default;
    S &operator=(S &&rhs) = default;

    ~S() = default;

    allocator_type get_allocator() const { return str.get_allocator(); }
};
```

Polymorphic memory resource

C++17/20 allocators



From [Value Proposition: Allocator Aware Software - John Lakos - Meeting C++ 2019 - YouTube]

Algebra data type

More convenient

- **std::pair** from c++98
- **std::tuple** from c++11
- **std::optional**
- **std::variant**

- **not std::any, it’s a type-erasued type, similar to std::function.**

Algebra data type

Why do we say “algebraic”?

It’s about the type’s number of possible values, a.k.a. the size of its domain.

```
char                        256 possible values
bool                        2 possible values (true and false)
pair<char,bool>             256 * 2 = 512 possible values
tuple<char,char,bool>       256 * 256 * 2 = 131072 possible values
variant<char,bool>          256 + 2 = 258 possible values
optional<char>              256 + 1 = 257 possible values
```

Algebra data type - optional

Use cases

- From the boost::optional documentation —— It is recommended to use optional<T> in situations where there is exactly one, clear (to all parties) reason for having no value of type T, and where the lack of value is as natural as having any regular value of

```
std::optional<std::string> TryParse(Input input)
{
    if (input.valid())
        return input.asString();

    return std::nullopt;
}
```

Algebra data type - optional

```cpp
// by operator*
std::optional<int> oint = 10;
std::cout << "oint " << *opt1 << '\n';

// by value()
std::optional<std::string> ostr("hello");
try
{
    std::cout << "ostr " << ostr.value() << '\n';
}
catch (const std::bad_optional_access &e)
{
    std::cout << e.what() << "\n";
}

// by value_or()
std::optional<double> odouble; // empty
std::cout << "odouble " << odouble.value_or(10.0) << '\n';
```

std::optional is a wrapper type to express “null-able” types.

std::optional won’t use any dynamic allocation

std::optional contains a value or it’s empty
use operator *, operator->, value() or value_or() to access the underlying value.

std::optional is implicitly converted to bool so that you can easily check if it contains a value or not.

Algebra data type - variant

What's the problem of union?

```
union S
{
    std::string str;
    std::vector<int> vec;
    ~S() { } // what to delete here?
};
```

```
int main()
{
    S s = {"Hello, world"};
    // at this point, reading from s.vec is undefined behavior
    std::cout << "s.str = " << s.str << '\n';

    // you have to call destructor of the contained objects!
    s.str.~basic_string<char>();

    // and a constructor!
    new (&s.vec) std::vector<int>;

    // now, s.vec is the active member of the union
    s.vec.push_back(10);
    std::cout << s.vec.size() << '\n';

    // another destructor
    s.vec.~vector<int>();
}
```

Algebra data type - variant

Variant is a typesafe union that will report errors when you want to access something that’s not currently inside the object.

```
std::variant<std::string, int> v{"Hello A Quite Long String"};
// v allocates some memory for the string
v = 10; // we call destructor for the string!
        // no memory leak
```

Algebra data type - variant

visit, the normal way

```cpp
std::variant<int, double, std::string> v;
if (int *pi = std::get_if<int>(&v)){
    std::cout << *pi << "\n";
}
else if (double *pd = std::get_if<double>(&v)){
    std::cout << *pd << "\n";
}
else if (auto *ps = std::get_if<std::string>(&v)){
    std::cout << *ps << "\n";
}
```

Algebra data type - variant

visit, the fancy way

```
std::variant<double, bool, std::string> var;
struct {
    void operator()(int) { std::cout << "int!\n"; }
    void operator()(std::string const&) { std::cout << "string!\n"; }
    void operator()(double const&) { std::cout << "double!\n"; }
} visitor;

std::visit(visitor, var);
```

```
std::variant<int, double, std::string> v;
auto printme = [](const auto &x)
{
    std::cout << x << "\n";
};
std::visit(printme, v);
```

Algebra data type - variant

overload pattern, the sexy way

```
template <typename ... Ts>
struct Overload : Ts ...
{
    using Ts::operator() ... ;
};
```

```
int main()
{

    std::variant<char, int, float> var = 2017;

    auto TypeOfIntegral = Overload{
        [](char) { return "char"; },
        [](int) { return "int"; },
        [](auto) { return "unknown type"; },
    };
    auto s = std::visit(TypeOfIntegral, var);
    return 0;
}
```

Algebra data type – tuple

Tuples are fixed-size heterogeneous containers.
They are a general-purpose utility, adding to the expressiveness of the language.

- Return types for functions that need to have more than one return type.
- Grouping related types or objects (such as entries in parameter lists) into single entities.
- Simultaneous assignment of multiple values.

```
auto SVD(const Matrix &A)
    -> tuple<Matrix, Vector, Matrix>
{
    Matrix U, V;
    Vector S;
    //  ...
    return {U, S, V};
};
```

```
void use()
{
    Matrix A;
    //  ...
    // using the tuple form and structured bindings
    auto [U, S, V] = SVD(A);
}
```

Algebra data type - tuple

Tuple is not great for public APIs

Let’s say I have a function that returns a hostname, a cert, and a time-to-live:

```
auto generateDefaultCert()
    -> std::tuple<std::string, Cert, double>;
// C++ before 17
auto hct = generateDefaultCert();
std::cout << "Made cert for host " << std::get<0>(hct) << "\n";
// C++17 structured binding
auto [host, cert, ttl] = generateDefaultCert();
std::cout << "Made cert for host " << host << "\n";
```

Algebra data type - tuple

Using a named class type, with named fields, is much friendlier.

```
struct CertInfo
{
    std::string host;
    Cert cert;
    double ttl;
};
CertInfo generateDefaultCert();
// C++ before 17
auto info = generateDefaultCert();
std::cout << "Made cert for host " << info.host << "\n";
// C++17 structured binding still permits this
auto [host, cert, ttl] = generateDefaultCert();
std::cout << "Made cert for host " << host << "\n";
```

string_view

Sometimes we just need a view, not a copy or a ref!

- Use std::string to own character sequences, since string correctly handles allocation, ownership, copying, gradual expansion, and offers a variety of useful operations.
- Use std::string when you need to perform locale-sensitive string operations

```
vector<string> read_until(string_view terminator)   // C++17
{
    vector<string> res;
    for (string s; cin >> s && s ≠ terminator; ) // read a word
        res.push_back(s);
    return res;
}
```

string_view

Sometimes we just need a view, not a copy or a ref!

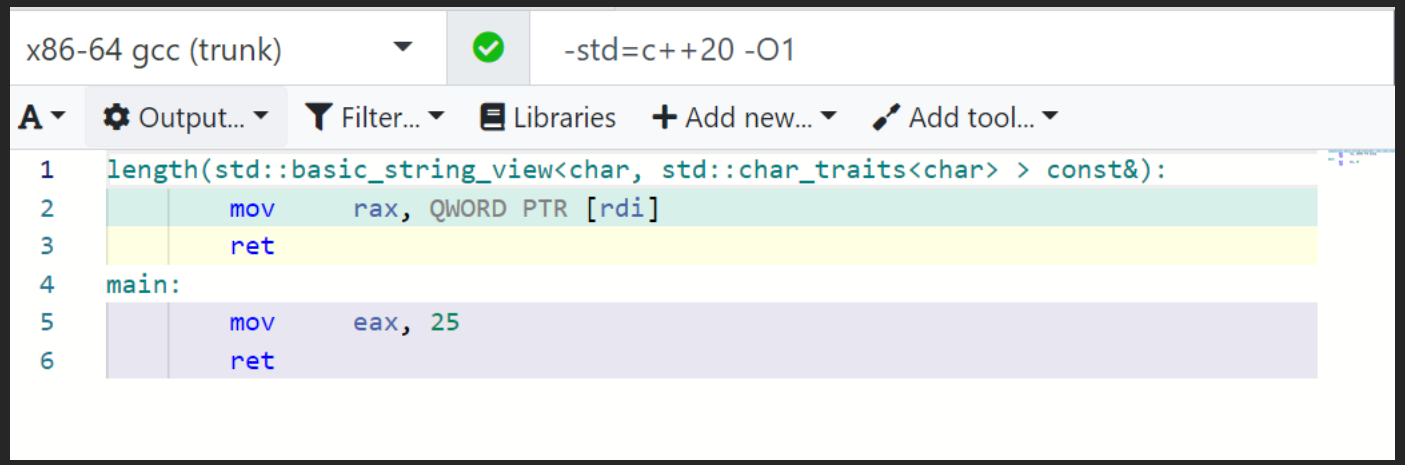
- string_view is a non-owning reference to an immutable sequence of characters
- Use std::string_view to refer to character sequences, it provides simple and (potentially) safe access to character sequences independently of how those sequences are allocated and stored.
- The std::string_view supports mostly all crucial function that is applied over the std::string like substr, compare, find, overloaded comparison operators (e.g., ==, <, >, !=).
- The std::string_view is better than the const std::string& because it removes the constraint of having a std::string object at the very beginning of the string as std::string_view is composed of two elements first one is const char* that points to the starting position of the array and the second is size.

- When strings are short (so they work nicely with SSO - Small String Optimization), and in that case, the perf boost might not be that visible.

string_view

An interesting example, string_view is quick !

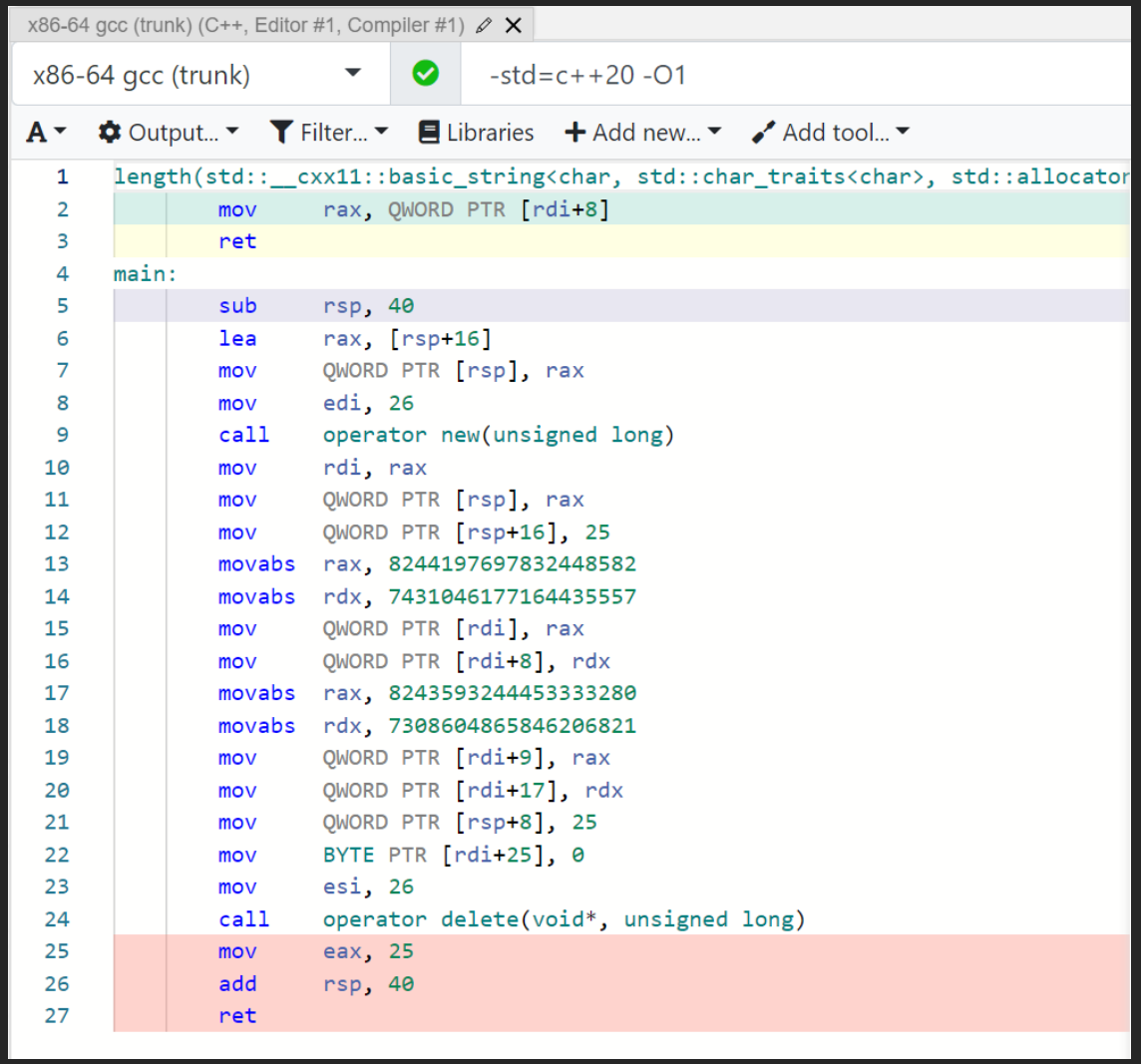
```
std::size_t length (
  const std::string_view &s)
{
    return s.size();
}

int main()
{
    return length("Free Fire is a
great game");
}
```



string_view

An interesting example, string is slow !

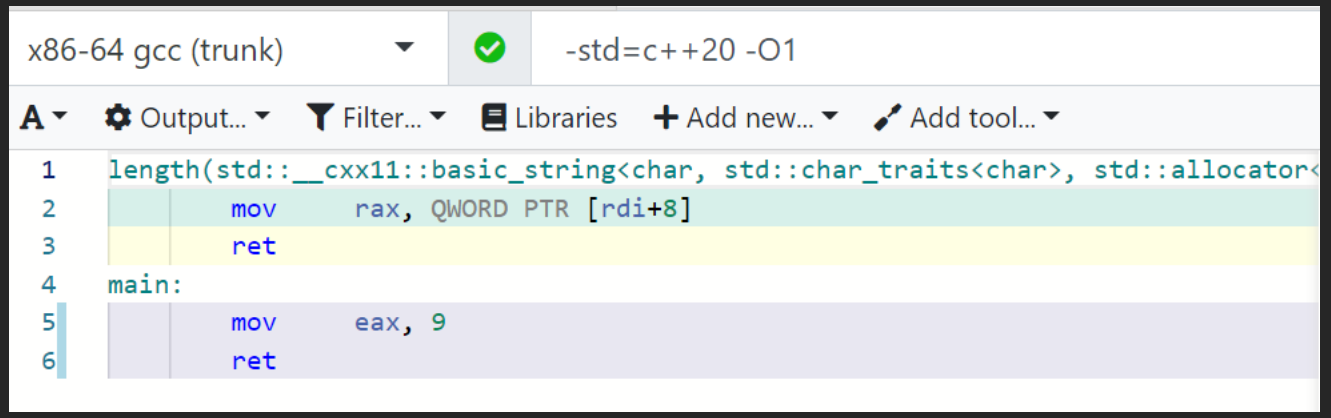
```
std::size_t length (
  const std::string &s)
{
    return s.size();
}

int main()
{
    return length("Free Fire is a
great game");
}
```



string_view

An interesting example, where SSO kicks in

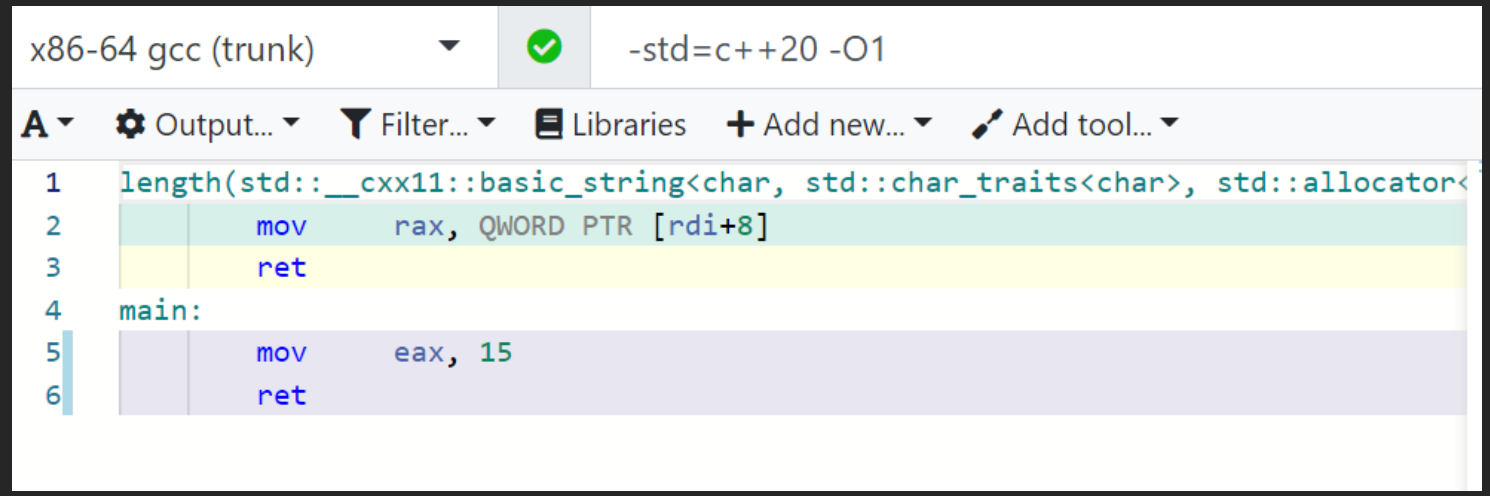
```
std::size_t length (
  const std::string &s)
{
    return s.size();
}

int main()
{
    return length("Free Fire");
}
```



string_view

An interesting example, where constexpr kicks in

```cpp
int main()
{
    constexpr auto s = std::string_view("Free Fire is a great game");
    constexpr auto t = s.find("great");
    return t;
}
```



inline

One Definition Rule

- There may be more than one definition of an inline function or variable (since C++17) in the program as long as each definition appears in a different translation unit and (for non-static inline functions and variables (since C++17)) all definitions are identical. For example, an inline function or an inline variable (since C++17) may be defined in a header file that is included in multiple source files.

The original intent of the inline keyword was to serve as an indicator to the optimizer that inline substitution of a function is preferred over function call

- Consider “**__forceinline**” if you really want to override the cost-benefit analysis and relies on the judgment of the programmer instead

What’s next ?

A glimpse of C++20, the big four

C++20

THE BIG FOUR

- Module
- Concept
- Range
- Coroutine

Module

Get rid of #include hell!

- Replace header files
- Modules explicitly state what should be exported (e.g. classes, functions, …)
- Separation into module interface files and module implementation files is possible but not needed
- Can be structured with submodules and module partitions
- No need for include guards
- No need to invent unique names, same name in multiple modules will not clash
- Modules are processed only once, faster build times
- Preprocessor macros have no effect on, and never leak from, modules
- Order of module imports is not important

Module

Get rid of #include hell!

- Create a module:

```
// freefire.cppm - Module Interface File
export module ff; // Module declaration

namespace FreeFire {
  auto GetWelcomeHelper() { return "Welcome to My Talk!"; }
  export auto GetWelcome() { return GetWelcomeHelper(); }
}
```

- Consume a module:

```
// main.cpp
import ff;
int main() {
  std::cout << FreeFire::GetWelcome();
}
```

Module

Get rid of #include hell!

- C++20 doesn’t specify if and how to modularize the Standard Library in bigger modules
- But, all C++ headers are “importable”, e.g.
  - import <version>;
  - Everything in <version> is exported, including macros
  - Improves build throughput; <version> will be processed only once
  - Comparable to precompiled header files (PCH)

Concept

No More enable_if!

- Named requirements to constrain template parameters
- Predicates evaluated at compile time

Concept

No More enable_if!

- Example of a concept definition:

```
template<typename T>
concept Incrementable = requires(T x) { x++; ++x; };
```

- Using this concept:

```
template<Incrementable T>
void Foo(T t);

template<typename T> requires Incrementable<T>
void Foo(T t);

template<typename T>
void Foo(T t) requires Incrementable<T>;

void Foo(Incrementable auto t);
```

Concept

No More enable_if!

- Combining concepts:

```
template<typename T> requires Incrementable<T> && Decrementable<T>
void Foo(T t);
```

- Or

```
template<typename T>
concept C = Incrementable<T> && Decrementable<T>;

void Foo(C auto t);
```

- The Standard defines a whole collection of standard concepts: same, derived_from, convertible_to, integral, constructible, … sortable, mergeable, permutable, …

Range

Abstraction!

What's a range?

- An object referring to a sequence/range of elements
- Similar to a begin/end iterator pair, but does not replace them

Why ranges ?

- nicer and easier to read syntax:
- Eliminate mismatching begin/end iterators
- Allows "range adaptors" to lazily transform/filter underlying sequences of elements

```
vector data { 11, 22, 33 };
sort(begin(data), end(data));
ranges::sort(data);
```

Range

Abstraction!

- Range: A concept defining iteration requirements
- Any container supporting begin()/end() is a valid range
- Range-based algorithms: all Standard Library algorithms accepting ranges instead of iterator pairs
- Projection: Transform elements before handing over to algorithm
- Views: transform/filter range: lazily evaluated, non-owning, non-mutating
- Range factories: construct views to produce values on demand
- E.g. sequence of integers
- Pipelining: Views can be chained using pipes -> |

Range

Abstraction!

Example of chaining views:

```
vector data { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto result { data
    | views::filter([](const auto& value) { return value % 2 == 0; })/* 2 4 6 8 10 */
    | views::transform([](const auto& value) { return value * 2.0; })/* 4 8 12 16 20 */
    | views::drop(2)                                                  /* 12 16 20 */
    | views::reverse                                                  /* 20 16 12 */
    | views::transform([](int i) { return to_string(i); }) };     /* "20" "16" "12" */
// Note: all lazily executed: nothing is done until you iterate over result
```

Coroutine

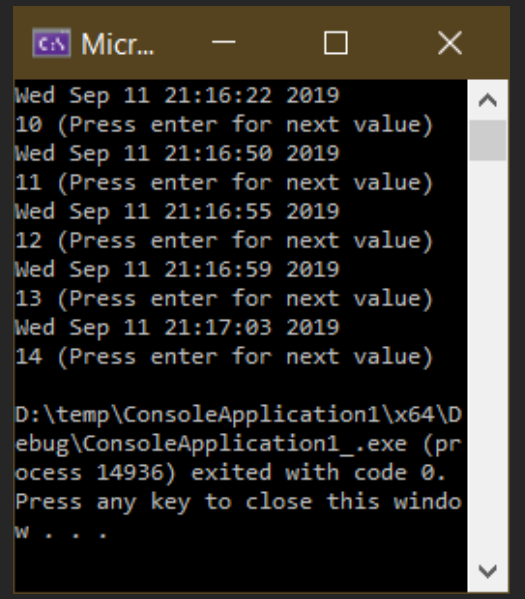
finally we have a coroutine in the language …
So what is a coroutine?

- A function
- with one of the following:
  - co_await: suspends coroutine while waiting for another computation to finish
  - co_yield: returns a value from a coroutine to the caller, and suspends the coroutine, subsequently calling the coroutine again continues its execution
  - co_return: returns from a coroutine (just return is not allowed)
- What are coroutines used for?
  - Generators
  - Asynchronous I/O
  - Lazy computations
  - Event driven applications

Coroutine

Feel it

```
experimental::generator<int> GetSequenceGenerator(
    int startValue, size_t numberOfValues)
{
    for (int i{startValue}; i < startValue + numberOfValues; ++i)
    {
        time_t t{system_clock::to_time_t(system_clock::now())};
        cout << std::ctime(&t);
        co_yield i;
    }
}

int main()
{
    auto gen{GetSequenceGenerator(10, 5)};
    for (const auto &value : gen)
    {
        cout << value << " (Press enter for next value)" << endl;
        cin.ignore();
    }
}
```



Tools we love

try clang-tidy!

Source code tools

clang is awesome!

- Prerequisite: compile your code with clang
- clang-format—— Five coding styles: LLVM, Google, Chromium, Mozilla, WebKit
- clang-tidy—— Linter with fixes
- clangd —— Intelligent code completion in your editor (VIM/Emacs)

Use and extend Clang-Tidy, modern C++ linter

Keep your codebase in a consistent style, save learning and maintenance cost.

- How to find similar misuses in your codebase? Regexp or machine learning?

```
const char *str = GetName();
// O(N^2) algorithm by accident if compiler can't
// hoist strlen() out of the loop
for (int i = 0; i < strlen(str); ++i)
{
    if ( ... )
    {
        // pass 'str' to another function
    }
}
```

Use and extend Clang-Tidy, modern C++ linter

Dump Clang AST (Abstract Syntax Tree)

- clang++ -fsyntax-only -Xclang -ast-dump slowloop.cc

```
-ForStmt 0xbea6d0 <line:7:3, line:9:9>
 |-DeclStmt 0xbea398 <line:7:8, col:17>
 | `-VarDecl 0xbea310 <col:8, col:16> col:12 used i 'int' cinit
 |   `-IntegerLiteral 0xbea378 <col:16> 'int' 0
 |-<<<NULL>>>
 |-BinaryOperator 0xbea4f8 <col:19, col:33> 'bool' '<'
 | |-ImplicitCastExpr 0xbea4e0 <col:19> 'int' <LValueToRValue>
 | | `-DeclRefExpr 0xbea3b0 <col:19> 'int' lvalue Var 0xbea310 'i' 'int'
 | `-CallExpr 0xbea4a0 <col:23, col:33> 'int'
 |   |-ImplicitCastExpr 0xbea488 <col:23> 'int (*)(const char *)' <FunctionToPointerDecay>
 |   | `-DeclRefExpr 0xbea438 <col:23> 'int (const char *)' lvalue Function 0xbe9f30 'strlen' 'int
 |   `-ImplicitCastExpr 0xbea4c8 <col:30> 'const char *' <LValueToRValue>
 |     `-DeclRefExpr 0xbea418 <col:30> 'const char *' lvalue ParmVar 0xbea038 'str' 'const char *'
 |-UnaryOperator 0xbea538 <col:36, col:38> 'int' lvalue prefix '++'
 | `-DeclRefExpr 0xbea518 <col:38> 'int' lvalue Var 0xbea310 'i' 'int'
 `-IfStmt 0xbea6b8 <line:8:5, line:9:9>
```

```
for (
  int i = 0;
  i < strlen(str);
  ++i) {
    if (...) {
```

Use and extend Clang-Tidy, modern C++ linter

Craft an AST matcher with clang-query

- ASTMatchers is a domain specific language to create predicates on Clang's AST

```
for (int i = 0; i < strlen(str); ++i)
```

```
forStmt(hasCondition(hasDescendant(
    callExpr(callee(functionDecl(hasName("strlen")))))))
```

Sanitizers: Address, Memory, Thread, etc.

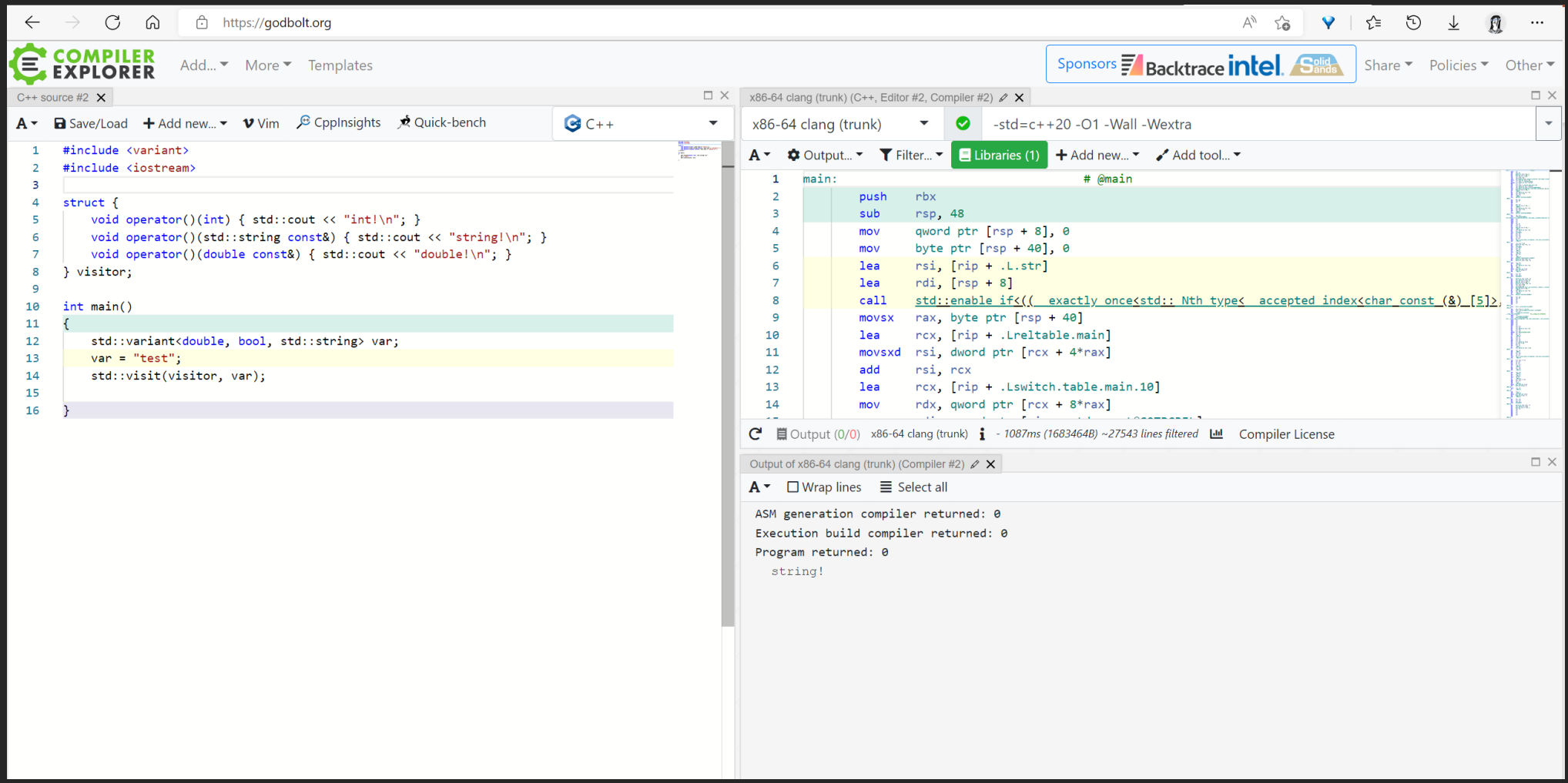
use it.

- Modern Valgrind memcheck, 2~3x slowdown, instead of 10~50x.
  - First added to gcc 4.8, mainstream maintained in clang/llvm.
- Common errors caught:
  - AddressSanitizer: use-after-free, out-of-bound read/write, double-free
  - MemorySanitizer: use-of-uninitialized-values
  - ThreadSanitizer: data race
  - UndefinedBehaviorSanitizer: integer overflow (could cause security bugs)

Learn to use the tools at your disposal

- Enable C++ compiler warnings
- Static code analysis tools (coverity, cppcheck, ...)
- Interactive debuggers (gdb, lldb, msvc, udb, ...)
- Time-travel debuggers (gdb, rr, liverecorder, udb, ...)
- Sanitizers (asan, tsan, ubsan, ...)
- Dynamic program analyzers (valgrind, callgrind, helgrind, ...)
- Call tracers and domain-specific diagnostic tools (strace, wireshark, SQL analyzers...)

Compiler Explorer!



Cpp Insight

```cpp
auto f(auto &&x)
{
    return x + x;
}

// x == 42
auto x = f(21);
// y == "hellohello"
auto y = f("hello"s);
```

```cpp
template <>
int f<int>(int &&x)
{
    return x + x;
}
#endif

template <>
std::basic_string<char, std::char_traits<char>, std::allocator<char>>
    f<std::basic_string<char, std::char_traits<char>, std::allocator<char>>>
    (std::basic_string<char, std::char_traits<char>, std::allocator<char>> &&x)
{
    return std::operator+(x, x);
}
int x = f(21);
std::basic_string<char, std::char_traits<char>, std::allocator<char>>
    y = f(std::operator""s("hello", 5UL));
```

Quick Bench

Quick C++ Benchmark

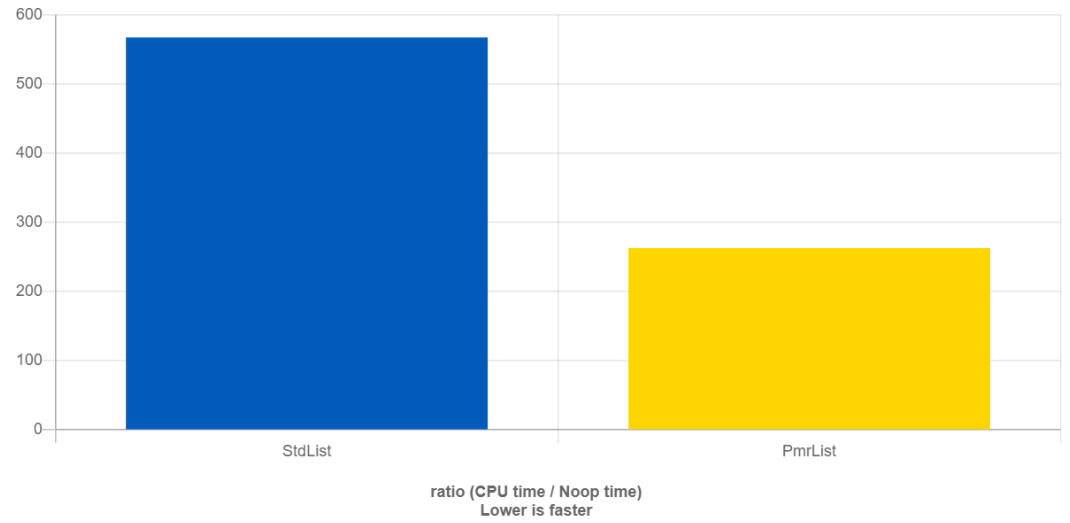
Run Quick Bench locally | Support Quick Bench Suite | More

```cpp
#include <memory_resource>
#include <list>

void StdList(benchmark::State& state) {
  for (auto _ : state) {
    std::list<int> list{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    benchmark::DoNotOptimize(list);
  }
}
// Register the function as a benchmark
BENCHMARK(StdList);

void PmrList(benchmark::State& state) {
  std::array<std::byte, 1024> buffer;
  for (auto _ : state) {
    std::pmr::monotonic_buffer_resource mem_resource(buffer.data(), buffer.size());
    std::pmr::list<int> list({1, 2, 3, 4 , 5, 6, 7, 8, 9, 10}, &mem_resource);
    benchmark::DoNotOptimize(list);
  }
}
// Register the function as a benchmark
BENCHMARK(PmrList);
```

compiler = GCC 10.1 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



Show Noop bar

APPENDIX

Useful resources

Useful Resources

- CppCon’s back to basics serious —— all of them !
- C++ Weekly With Jason Turner —— A “language lawyer” ’s perspective
- Thriving in a Crowded and Changing World: C++ 2006–2020 —— by BJARNE STROUSTRUP
- CppCoreGuidelines by BJARNE STROUSTRUP and Herb Sutter
- 《C++ Templates The Complete Guide》 if you want to learn some crazy stuff
- 《C++ Concurrency in action》 if you want to write some lock-free code (danger!)
- 《Hands-On Design Patterns with C++》 if you want to learn some modern cpp design pattern techniques
- Maybe, learn some Haskell or Rust


FREE FIRE
THANK YOU
Don’t hesitate to ask questions