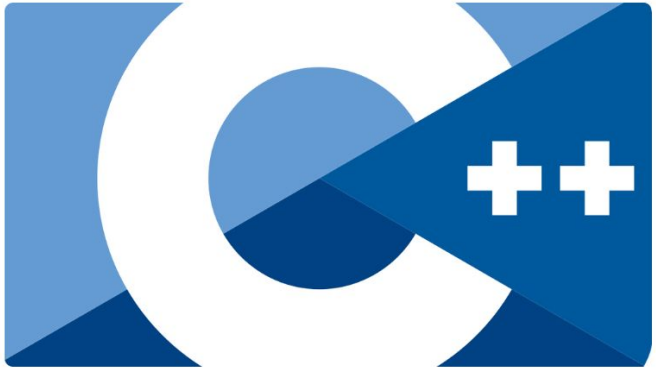


Aachen C++ Meetup

C++ User Gruppe Aachen

To Move or Not to Move

an interactive analysis

Amir Kirsh

About me

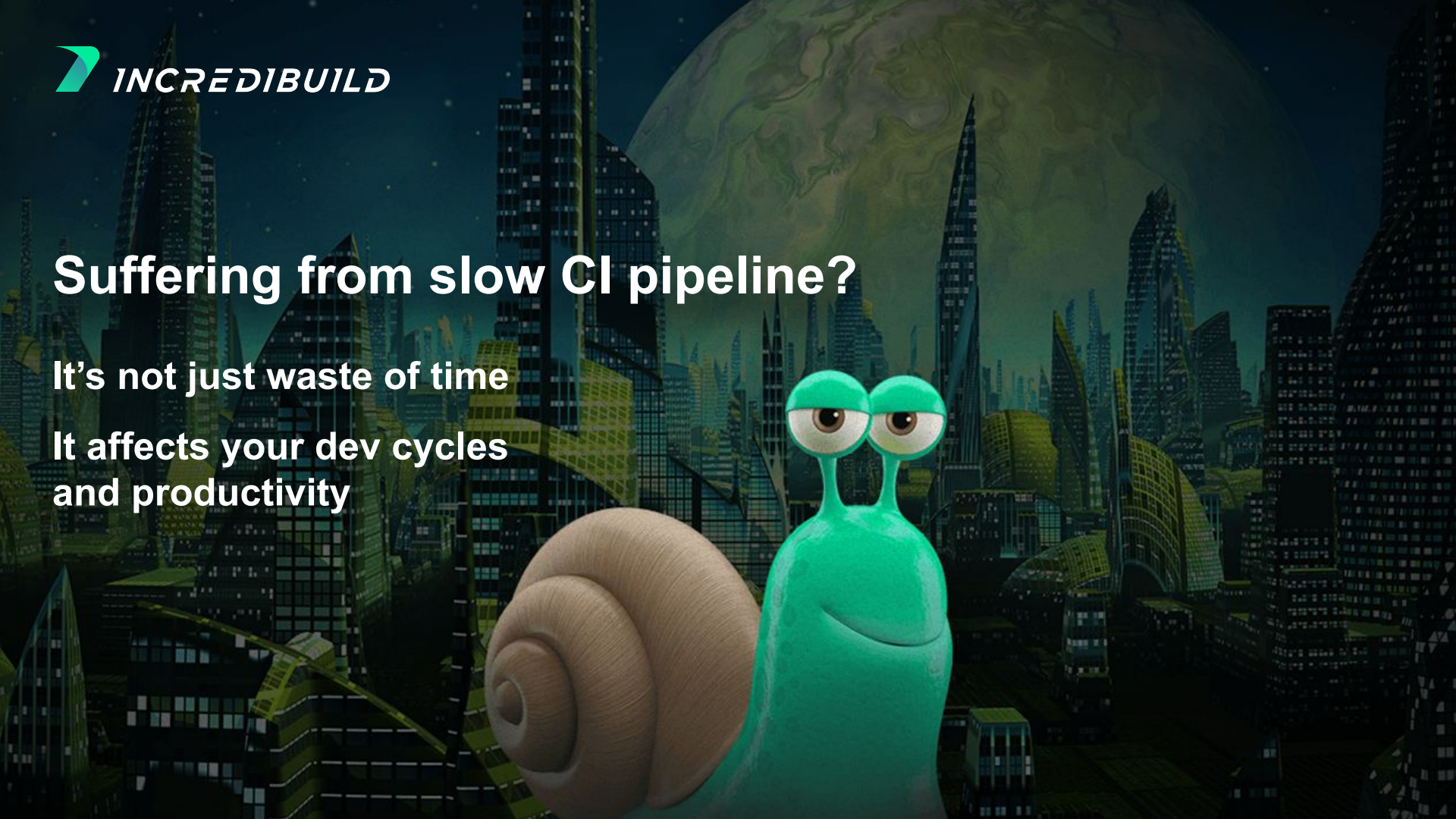
Lecturer
Academic College of Tel-Aviv-Yaffo
Tel-Aviv University
Visiting lecturer @ Stony Brook University

Developer Advocate at

INCREDIBUILD

Member of the Israeli ISO C++ NB

Co-Organizer of the **CoreCpp** conference and meetup group

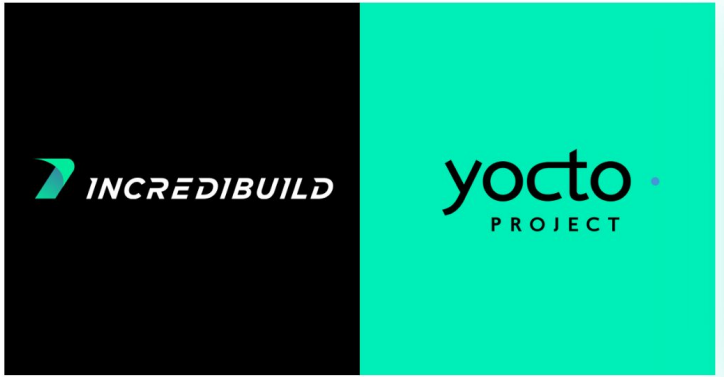

INCREDIBUILD
Suffering from slow CI pipeline?
It’s not just waste of time
It affects your dev cycles
and productivity

We also accelerate Yocto builds!

Our recent talks at Yocto Project Summit:

https://bit.ly/YPS-2022_IB_bitbake

https://bit.ly/YPS-2022_IB_Cache



Incredibuild + Yocto:

https://www.incredibuild.com/blog/announcing-incredibuild-support-for-yocto
https://www.incredibuild.com/lp/yocto

Incredibuild for Automotive

Relevant Sub-Sectors:

- Infotainment
- Instrument cluster
- Heads-up-display (HUD)
- Telematics/connected car
- Advanced driver assistance systems (ADAS)
- Functional safety and autonomous driving

Jaguar Land Rover, Nissan, Toyota, DENSO Corporation, Fujitsu, HARMAN, NVIDIA, Renesas, Samsung

146 members

Relevant Linux OS's / distribution collaborations:

Yocto, QNX, AOSP, Bazel, AGL

The motivation for Move Semantics

```
Godzilla g1 = factory.createFrighteningGodzilla();
Godzilla g2;
g2 = factory.createSpookyGodzilla();
list<Godzilla> godzillas;
godzillas.push_back(Godzilla("sweety"));
```

What is being called?

What is being called?

```
A::A(std::string a_name): name(a_name) {}

B::B(std::string&& b_name): name(b_name) {}
```

A copy in A, move in B

B copy in B, move in A

C copy in both

D move in both

What is being called?

```
A::A(std::string a_name): name(a_name) {}

B::B(std::string&& b_name): name(b_name) {}
```

A copy in A, move in B

B copy in B, move in A

C copy in both

D move in both

The need for std::move

```
A::A(std::string a_name): name(std::move(a_name)) {}

B::B(std::string&& b_name): name(std::move(b_name)) {}
```

What is being called?

What is being called?

```
A::A(const std::string& a_name): name(std::move(a_name)) {}
// note that above is bad, you should not move here
// but what would happen if you accidentally do?
```

A move

B copy

C the code would not compile

D it’s compiler dependent

What is being called?

```
A::A(const std::string& a_name): name(std::move(a_name)) {}
// note that above is bad, you should not move here
// but what would happen if you accidentally do?
```

A move

B copy

C the code would not compile

D it's compiler dependent

reference type overload resolution

		A	B	C	D**
	Who is sent => Candidate Functions	lvalue	const lvalue	rvalue	const rvalue
1	f(X& x)	(1) ✔			
2	f(const X& x)	(2) ✔	✔	(3) ✔	(2) ✔
3	f(X&& x)			(1) ✔	
4*	f(const X&& x)			(2) ✔	(1) ✔

Table source:

C++ - How does the compiler decide between overloaded functions with reference types as parameter? - Stack Overflow

* Row 4 is rare - rationale in handling rvalue reference is to “steal” it.
But you cannot steal if it is a const rvalue reference, so usually you will not implement row 4.
Still, possible use case: Do rvalue references to const have any use? - Stack Overflow

** Column D is also rare and mostly irrelevant - if it happens, would be usually handled by row 2 and not by row 4 (-- as row 4 is most probably not implemented).

Is it valid to std::move an lvalue reference?

```
template<class T>
void foo(T& a, T& b) {
    T temp = std::move(a);
    // do some more stuff
}
```

Is it valid to std::move an lvalue reference?

```
template<class T>
void foo(T& a, T& b) {
    T temp = std::move(a);
    // do some more stuff
}
```

A it will not compile

B compiles with a warning

C it will compile but it's undefined behavior

D yes, it can be legit in some cases

Is it valid to std::move an lvalue reference?

```
template<class T>
void foo(T& a, T& b) {
    T temp = std::move(a);
    // do some more stuff
}
```

A it will not compile

B compiles with a warning

C it will compile but it's undefined behavior

D yes, it can be legit in some cases

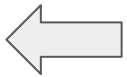
Is it valid to std::move an lvalue reference? YES

```
template<class T>
void swap(T& lhv, T& rhv) {
    T temp = std::move(lhv);  // we steal from lvalue
    lhv = std::move(rhv);     // because we override it here
    rhv = std::move(temp);
}
```

Be cautious with passing by value

Be cautious with passing by value

```
std::set<string> long_strings;
void store(string s) {
  long_strings.insert(std::move(s));
}
```



we copy even if not needed

the rule of “if you need to copy pass by value” needs great care

See: https://stackoverflow.com/questions/10231349/are-the-days-of-passing-const-stdstring-as-a-parameter-over

Related:

The *copy and swap idiom* is elegant (maybe) but *inefficient*…

http://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

https://stackoverflow.com/questions/24014130/should-the-copy-and-swap-idiom-become-the-copy-and-move-idiom-in-c11/24018053#24018053

Alternatives

```
void store(const string& s) {
  long_strings.insert(s);
}

void store(string&& s) {
  long_strings.insert(std::move(s));
}
```

OR

```
template<typename T> requires
std::convertible_to<T, std::string>
void store(T&& s) {
  long_strings
     .insert(std::forward<T>(s));
}
```

Alternatives

Inserting *existing item* into std::set via our *store* function

	byval	const ref	const ref + rval	forwarding ref
lvalue	**copy**	**---**	**---**	**---**
rvalue	**move**	**copy**	**move**	**move**

GCC (with libstdc++) and Clang (with libc++) both with -O3
https://godbolt.org/z/954KeM

Alternatives

Inserting *existing item* into std::set via our *store* function

	byval	const ref	const ref + rval	forwarding ref
lvalue	**copy**	**---**	**---**	**---**
rvalue	**move**	**copy**	**move**	**move**

better

GCC (with libstdc++) and Clang (with libc++) both with -O3
https://godbolt.org/z/954KeM

What’s wrong here?

```
friend MyString&& operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return std::move(concat);
}
```

What's wrong here?

```
friend MyString&& operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return std::move(concat);
}
```

A nothing, code is fine

B returning a dangling ref

C a performance issue

D code doesn't compile

What's wrong here?

Code: https://godbolt.org/z/856PTjKdd
See also, Stack Overflow:
Is there any case where a return of a RValue Reference (&&) is useful?

```
friend MyString&& operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return std::move(concat);
}
```

A nothing, code is fine

B returning a dangling ref

C a performance issue

D code doesn't compile

What’s wrong here?

```
friend MyString operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return std::move(concat);
}
```

What's wrong here?

```
friend MyString operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return std::move(concat);
}
```

A nothing, code is fine

B returning a dangling ref

C a performance issue

D code doesn't compile

What's wrong here?

Code:
https://godbolt.org/z/WxzExM5ef
See also, Stack Overflow:
C++11 rvalues and move semantics - return statement

```
friend MyString operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return std::move(concat);
}
```

A nothing, code is fine

B returning a dangling ref

C a performance issue

D code doesn't compile

The proper way

```
friend MyString operator+(const MyString& s1, const MyString& s2) {
    MyString concat /* = do concatenation*/;
    return concat; // implicit move on return of a local variable
}
```

What's wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::forward<T>(t));
    }
    // ...
};
```

What's wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::forward<T>(t));
    }
    // ...
};
```

A **T&&** in **push** is NOT a forwarding reference, thus **compilation error**

B **T&&** in **push** is NOT a forwarding reference, thus we support only push of rvalues

C **push** may add to the vector a dangling ref

D **push** may inefficiently copy when it can move an item into the vector

What's wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::forward<T>(t));
    }
    // ...
};
```

A **T&&** in **push** is NOT a forwarding reference, thus **compilation error**

B **T&&** in **push** is NOT a forwarding reference, thus we support only push of rvalues

C **push** may add to the vector a dangling ref

D **push** may inefficiently copy when it can move an item into the vector

The proper way - option 1

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    void push(T&& t) {
        vec.push_back(std::move(t));
    }
    void push(const T& t) {
        vec.push_back(t);
    }
    // ...
};
```

The proper way - option 2

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    template<typename U> requires std::convertible_to<U, T>
    void push(U&& u) {
        vec.push_back(std::forward<U>(u));
    }
    // ...
};
```

What's wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    // ...
    T pop() {
        T& e = vec.back();
        vec.pop_back();
        return std::move(e);
    }
};
```

What's wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    // ...
    T pop() {
        T& e = vec.back();
        vec.pop_back();
        return std::move(e);
    }
};
```

A **pop** returns a dangling reference

B **pop** moves from a dangling reference (code would be OK without the call to **`std::move`**)

C **pop** has UB: "moving out" from a vector is impossible

D the reference **e** is being invalidated once we call **`pop_back`**

What's wrong here?

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    // ...
    T pop() {
        T& e = vec.back();
        vec.pop_back(); // e's dtor called
        return std::move(e);
    }
};
```

A **pop** returns a dangling reference

B **pop** moves from a dangling reference (code would be OK without the call to **`std::move`**)

C **pop** has UB: "moving out" from a vector is impossible

D the reference **`e`** is being invalidated once we call **`pop_back`**

The proper way

```
template<typename T>
class Stack {
    std::vector<T> vec;
public:
    T pop() {
        T e = std::move(vec.back());
        vec.pop_back();
        return e;
    }
    // ...
};
```

Code:
http://coliru.stacked-crooked.com/a/b339af287c876ec4

See also - Stack Overflow:
- Iterator invalidation rules for C++ containers
- pop_back() return value?
- How to store a value obtained from a vector `pop_back()` in C++?

A side note

Implementing *move* forgetting *noexcept*

Implementing *move* forgetting *noexcept*

vector's push_back implementation is allowed to use *move ctor* only if it is declared as noexcept:

```
A(A&& a) noexcept {
  // code
}
```

Why? to avoid possible bad scenario of exception during move

- we call push_back to add a Godzilla to vector<Godzilla>
- capacity of vector is exhausted, so vector capacity shall be enlarged
- new bigger allocation is made, old Godzillas shall be moved
- while moving Godzilla at index N an exception is thrown
- we have now two broken vectors and cannot rollback

Read: https://en.cppreference.com/w/cpp/utility/move_if_noexcept
https://stackoverflow.com/questions/28627348/noexcept-and-copy-move-constructors

Implementing *move* forgetting *noexcept*

Don’t believe there is a difference?

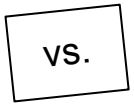
```
A(A&& a) noexcept {
  // code
}
```


```
A(A&& a) /* oops forgot */ {
    // code
}
```

Implementing *move* forgetting *noexcept*

Don’t believe there is a difference?

```
A(A&& a) noexcept {
  // code
}
```




```
A(A&& a) /* oops forgot */ {
    // code
}
```

```
in A's empty ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
in A's move ctor
...
```

```
in A's empty ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
in A's copy ctor
...
```

http://coliru.stacked-crooked.com/a/15a89b45b0dcfedd

Last One

Last One

Are you ready?

What's wrong here?

```
template<typename T>
T my_max(T&& a, T&& b) {
    return std::forward<T>(b) < std::forward<T>(a) ?
        std::forward<T>(a) : std::forward<T>(b);
}
```

Code: https://godbolt.org/z/rx4xq7Ksx

Summary

Summary (1)

Don’t std::move *anything* without thinking

Don’t std::move local variables on return

Don’t move something that is still in use by you or others

Don’t move something twice

Summary (2)

Don’t waive std::move *when needed*

You should std::move an rvalue that has a name, and you know you won’t be using it anymore and thus can move from it

You can std::move lvalues, if the moved value would not be used

A bonus slide

```
auto max = strs[0];
// we put into the lambda intentionally (for this example) something that cannot be copied
auto ptr = std::make_unique<int>(0);
auto callback =
    [max, ptr = std::move(ptr)](auto&& s) mutable {  // moving into lambda (C++14)
        if (s > max) {
            ++(*ptr);
            std::cout << *ptr << '\n';
            using s_type = decltype(s);    // rvalue maybe
            max = std::forward<s_type>(s); // forwarding an auto&& param
        }
        return max;
    };
```

Code: https://godbolt.org/z/YqxoY3o55

Thank you!

```
void conclude(auto&& greetings) {
    while(still_time() && have_questions()) {
        ask();
    }
    greetings();
}

conclude([]{ std::cout << "Thank you!"; });


// Comments, feedback: kirshamir@gmail.com
// let me help you accelerate you builds: amir.kirsh@incredibuild.com
```