Back To Basics
Rvalues and Move Semantics

AMIR KIRSH

CppCon
The C++ Conference

2024
September 15 - 20

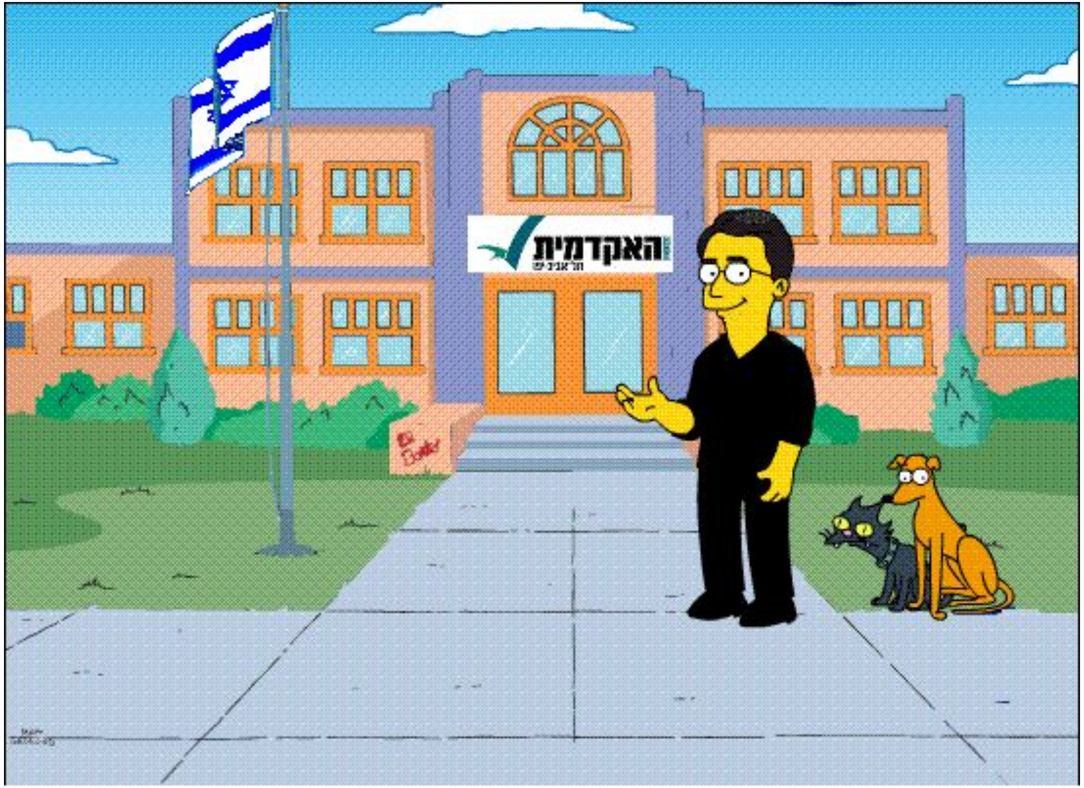
About me

Lecturer
Academic College of Tel-Aviv-Yaffo
Tel-Aviv University

Member of the Israeli ISO C++ NB

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

Trainer and Advisor
(C++, but not only)



The motivation for Move Semantics

The motivation for Move Semantics

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

The motivation for Move Semantics

Avoid redundant copying when we can steal from a dead object!

The motivation for Move Semantics

an almost dead object

Avoid redundant copying when we can ~~steal~~ from ~~a dead object!~~

move

The motivation for Move Semantics

an almost dead object

Avoid redundant copying when we can ~~steal~~ from ~~a dead object!~~

move

* Some of the redundant inefficient copies can be avoided by the compiler, using RVO (Return Value Optimization) or NRVO (Named Return Value Optimization). But not all.

Recap questions

Recap question (1)

assume str1 is another std::string

```
std::string str2 = str1;
```

Should we ~~steal~~ move here?

Recap question (2)

```
std::string str3 = str2 + str1;
```

Should we ~~steal~~ move here?

assume str1 and str2 are both
objects of type std::string

So, how can we distinguish between the cases?

```
std::string str2 = str1;          // (a) do not move
std::string str3 = str2 + str1;  // (b) move
```

So, how can we distinguish between the cases?

```
std::string str2 = str1;          // (a) do not move
std::string str3 = str2 + str1;   // (b) move
```

So, how can we distinguish between the cases?

```
std::string str2 = str1;          // (a) do not move
std::string str3 = str2 + str1;   // (b) move
```

- Both assigned expressions are references

So, how can we distinguish between the cases?

```
std::string str2 = str1;            // (a) do not move
std::string str3 = str2 + str1;  // (b) move
```

- Both assigned expressions are references
- Currently, both lines would call the copy constructor

So, how can we distinguish between the cases?

```
std::string str2 = str1;            // (a) do not move
std::string str3 = str2 + str1;  // (b) move
```

- Both assigned expressions are references
- Before C++11, both lines would call the copy constructor
- *But they are kind of a **different references**:*
    - The reference at **(a)** ***would still be alive after the end of the statement***
    - The reference at **(b)** ***would not, it is a temporary object***

Let’s name them

Let’s name them

```
std::string str2 = str1;
```

Let’s call a reference that would *still be alive* after the end of the statement.

Lvalue reference

Let’s name them

```
std::string str2 = str2 + str1;
```

Let’s call a reference that *would be dead* by the end of the statement.

Rvalue reference

Now, we need a language symbol for each...

Lvalue reference

```
std::string str2 = str1;
```

We already have a language symbol for this one:

Lvalue reference

```
std::string str2 = str1;
```

We already have a language symbol for this one:

&

Rvalue reference

```
std::string str2 = str2 + str1;
```

But what about this? Which symbol is “free” for use?

?

Rvalue reference

```
std::string str2 = str2 + str1;
```

But what about this? Which symbol is “free” for use?

&&

OK, so what can we do now with & and &&?

We can overload functions!

Let’s try a toy example

Let’s try a toy example

```
void foo(int& lvalueref) { cout << "in Lvalueref foo!\n"; }
void foo(int&& rvalueref) { cout << "in Rvalueref foo!\n"; }

int main() {
  int i = 3;
  foo(i);
  foo(3);
  const int j = i;
  foo(j);  // which one is called here?
}
```

Code: http://coliru.stacked-crooked.com/a/0e0244f1dc5ee040

Exercise

Now that we know the difference between A& and A&& - let's implement a simple MyString class with:

- **Copy ctor**
- **Move ctor ("steals" the other instead of copying)**
- **Copy Assignment operator**
- **Move assignment operator (again, "steals")**

Skeleton: https://coliru.stacked-crooked.com/a/b9e793be2a5f220b

Solution: https://coliru.stacked-crooked.com/a/e03c79803a0c4ffd

Reference 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: https://stackoverflow.com/questions/47734382/47736813#47736813

* Row 4 is rare - rationale for 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: https://stackoverflow.com/questions/4938875

** 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).

Some additional rules

If it has a name…

```
void foo(int& lvalueref) { cout << "in Lvalueref foo!\n"; }
void foo(int&& rvalueref) { cout << "in Rvalueref foo!\n"; }

void bar(int&& i) {
  foo(i); // which foo will be called from here?
}

int main() {
  bar(3);
}
```

The need for *std::move*

```
void foo(int& lvalueref) { cout << "in Lvalueref foo!\n"; }
void foo(int&& rvalueref) { cout << "in Rvalueref foo!\n"; }

void bar(int&& i) {
  foo(std::move(i)); // which foo will be called now?
}

int main() {
  bar(3);
}
```

std::move

std::move is a casting to rvalue ref, preserving const-volatile ("cv") type-qualifiers.

std::move doesn't move

std::move prepares its argument to be moved, but it doesn't actually perform any move on its own!

```
std::string s2 = std::move(s1);
```

The "move" is performed here,
calling the move ctor in this case

A question

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);
}
```

The rule of implicit move on return

The rule of implicit move on return

```
struct Moo {
  std::string s;
  // ...
};

Moo foo() {
  Moo moo {"hello"};
  return moo; // implicitly moved (do not call std::move!)
}
```

So, when do we need to implement move?

So, when do we need to implement move?

Do we need to implement move for this class:

```
class Point {
  int x, y;
public:
    // ctor, methods ...
};
```

So, when do we need to implement move?

Do we need to implement move for this class:

```
class Point {
  int x, y;
public:
    // ctor, methods ...
};
```

No need for move!

So, when do we need to implement move?

And what about this class:

```
class Person {
  long id;
  std::string name;
public:
    // ctor, methods ...
};
```

So, when do we need to implement move?

And what about this class:

```
class Person {
  long id;
  std::string name;
public:
    // ctor, methods ...
};
```

No need for move!

Rule of Zero

It is the best if your class doesn't need any resource management

- no need for dtor, copy ctor, assignment operator
- defaults do the job
- that includes defaults for move operations

To achieve that, use properly managed data members:
std::string, std containers, std::unique_ptr, std::shared_ptr

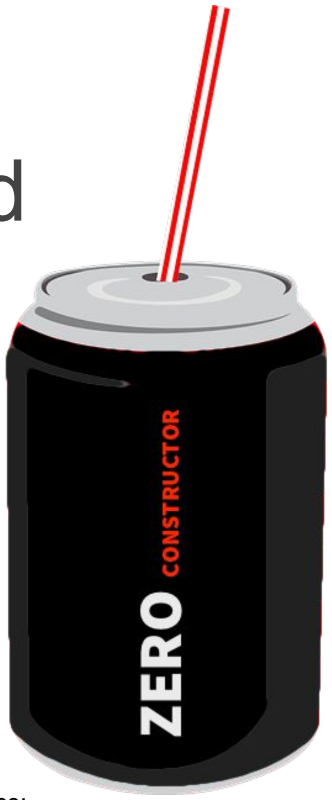


Image Source:
https://www.fluentcpp.com/2019/04/23/the-rule-of-zero-zero-constructor-zero-calorie/

What about this class?

```
class NamedArray {
  size_t size;
  int* arr;
  std::string name;
public:
    // ctor, methods ...
};
```

What about this class?

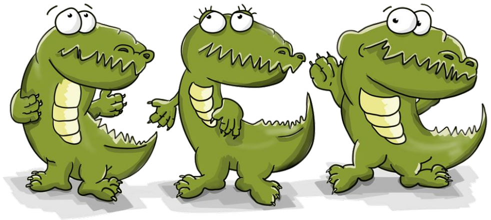
```
class NamedArray {
  size_t size;
  int* arr;
  std::string name;
public:
    // ctor, methods ...
};
```

Skeleton for the solution:
https://coliru.stacked-crooked.com/a/1e5bac3dcd358417

Solution 1 - with a need for std::move:
https://coliru.stacked-crooked.com/a/06907d732f61dbca

Solution 2 - better design:
https://coliru.stacked-crooked.com/a/3b9c4f0b05fc7241

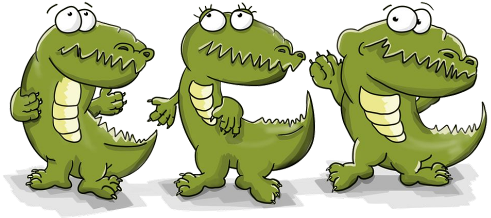
Rule of Three (1)



Implementing any of the three: dtor, copy ctor, assignment operator, will (*may*) warn on usage of the compiler provided default copy ctor / assignment operator.

See: http://en.cppreference.com/w/cpp/language/rule_of_three

Rule of Three (2)



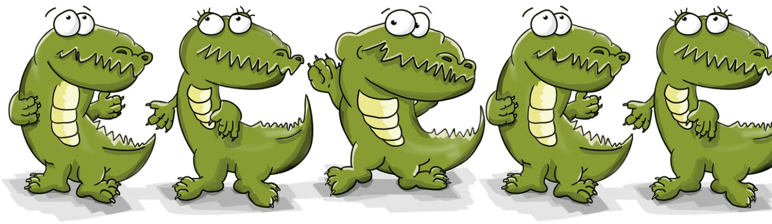
If you need a destructor, first thing block the copy ctor and assignment operator

- No TODO, no let's check if we need to implement them, BLOCK NOW

```
MyClass(const MyClass&) = delete;
MyClass& operator=(const MyClass&) = delete;
```

- If you need them later => implement

Rule of Five (1)

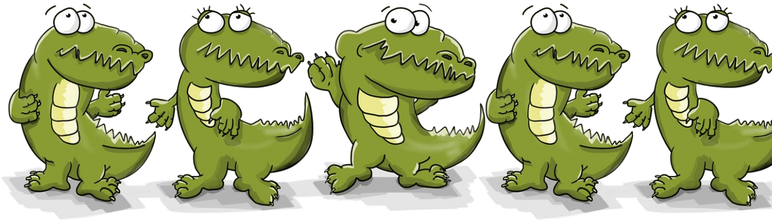


If you declare *any one of the five*, you lose the defaults provided by the compiler, for the move operations.

- Make sure to ask back for the defaults if they are fine

```
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
```

Rule of Five (2)



If you declare *any of the move operations*, all the other operations (copy/move ctor, copy/move assignment operator) are not default provided by the compiler any more.

- Make sure to ask back the defaults if they are fine, e.g.

```
MyClass(const MyClass&) = default;
MyClass& operator=(const MyClass&) = default;
MyClass(MyClass&&) = default;
MyClass& operator=(MyClass&&) = default;
```

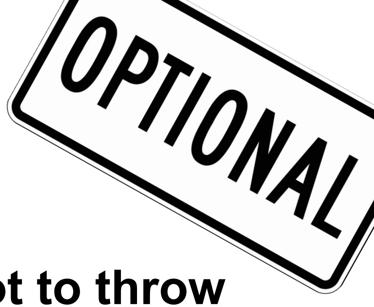
When do we use rvalue ref?

When do we use rvalue ref?

<u>Mostly</u> - on parameters, to allow overload for “Move” operations

<u>In rare cases</u> - on variable definition
usually you would just create a “value type” if it’s not an lvalue ref

<u>In very rare cases</u> - on function return value: http://stackoverflow.com/a/5770888



move if no except

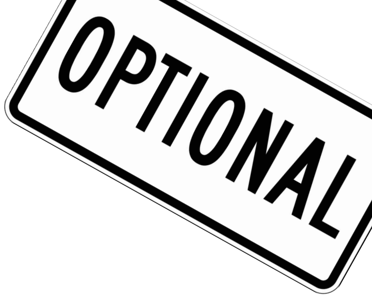
There are cases where move can be used only if it promises not to throw any exception:

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

Scenario:

- we call push_back to add a Godzilla to vector<Godzilla>
- capacity of vector is exhausted, so vector capacity shall be enlarged to allow insertion
- new bigger allocation is made, all old Godzillas shall be moved / copied to the new place
- vector is allowed to use move, to move the elements from the old location to the new one, but only if the move constructor of Godzilla is declared as noexcept, to avoid the bad case of "partial work done - there is no good vector but two broken ones..."

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*

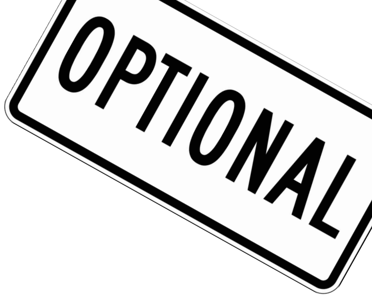
Don’t believe there is a difference?

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

vs.

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

Implementing *move* forgetting *noexcept*



Don’t believe there is a difference?

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

vs.

```
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

forwarding reference (AKA “universal reference”)

forwarding reference (AKA “universal reference”)

```
template<typename T>
void bar(T&& t) {
  // t is not an rvalue ref, it’s a forwarding reference
  // to pass it on as it was passed to us, we use std::forward
  foo(std::forward<T>(t));
}
```

&& on direct template argument or on auto, is a forwarding reference and not rvalue ref (based on reference collapsing rules)

forwarding reference - usage examples

forwarding reference - usage example 1

```
template<typename T, std::size_t SIZE> class Array {
    T arr[SIZE] = {};
public:
    // ...
    template<typename... Ts>
    void emplace_at(std::size_t index, Ts&&... ts) {
        arr[index].~T();
        new (&arr[index]) T(std::forward<Ts>(ts)...); // placement new
    }
    // ...
};
```

Code: https://coliru.stacked-crooked.com/a/6f85b23484f8ba46

forwarding reference - usage example 2

```
vector<bool> vb = {true, true, false};
vector<int> vi = {1, 1, 0};

template<class T> void reset(vector<T>& vec) {
    for(auto&& item : vec) // auto& wouldn't work for vector<bool>!
        item = {};
}
// auto&& is ok with all kinds: lvalue-ref, rvalue-ref and simple values
// thus ok with returned item for vector<int> which is int&
// and also ok with returned item for vector<bool> which is a proxy object ByVal!
```

Code: https://coliru.stacked-crooked.com/a/dd957e786fc0241d

Quiz Part - if time permits

Skip if no time … :-(

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++?

If we forget to use move, would clang-tidy warn on the inefficiency?
Currently no… https://clang-tidy.godbolt.org/z/b36coxdn4

Summary

Summary (1)

Rvalues and move semantics are here to improve our code performance

Stick to Rule of Zero when you can!
You will get the default move operations for your data members and base classes.

Summary (2)

If your class do manage resources and you need to implement move operations

Narrow the class to the minimum required to manage the resource.

Use the class by others, sticking to the rule of zero.

Summary (3)

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

Don’t std::move local variables on return (pessimization).

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

Don’t move something twice.

Summary (4)

Don’t waive std::move *when needed*

You may need to move an rvalue that has a name, and you know you won’t be using it anymore, and thus can move from it.

You may move lvalues, if the moved object would not be used.

Summary (5)

Remember that forwarding references need to use std::forward, not move

This is relevant when you implement template functions.

Any questions before we conclude?

```cpp
class Greetings {
   std::string greetings[] =
         {"Thank you!"s, "תודה לכולם"s};
public:
    void greet() const;
};
```