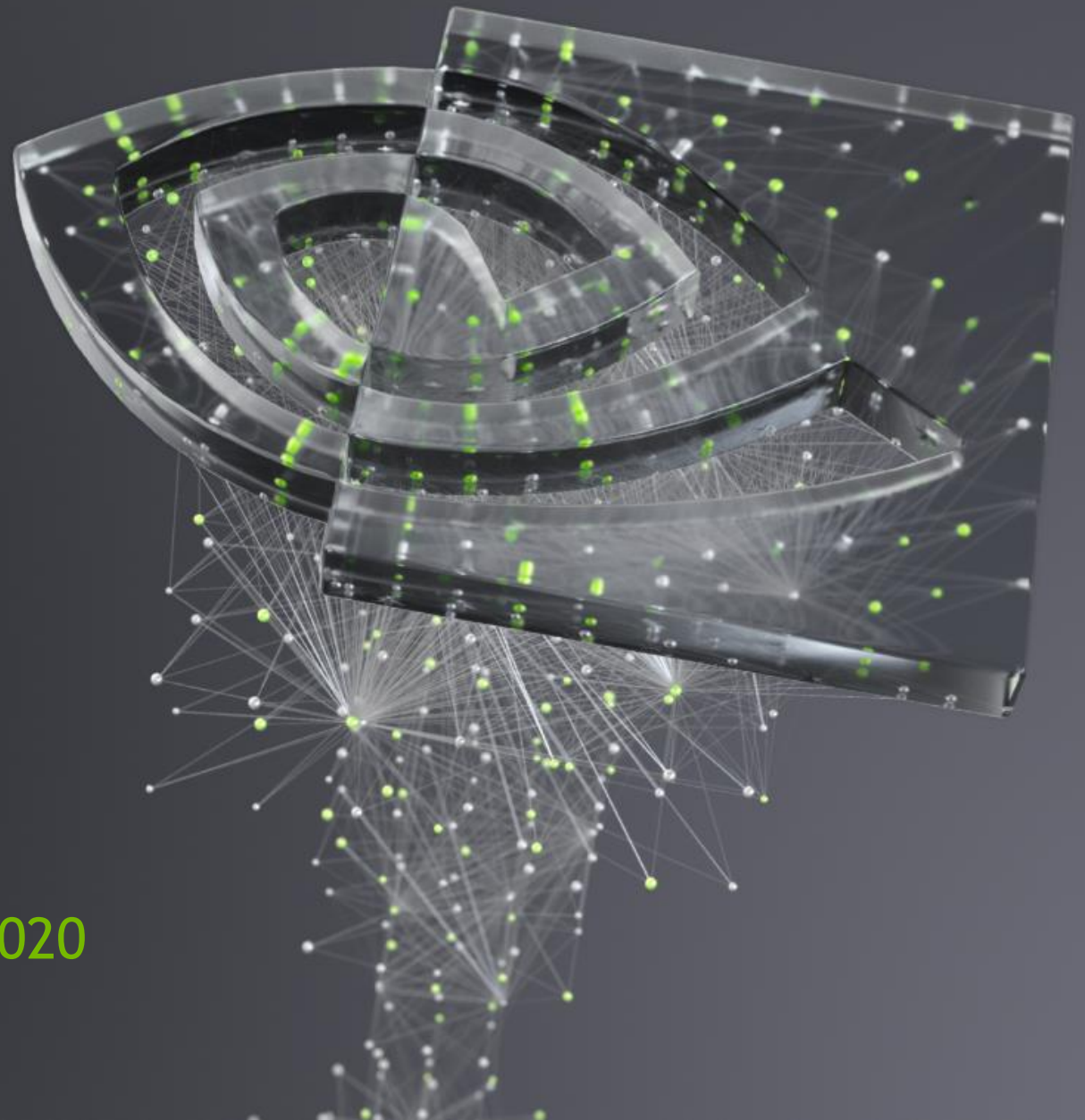


NVIDIA

Back to Basics: Move Semantics

David Olsen, CppCon, 17 Sep 2020

The Situation

It’s 2003…

You need to write code that creates and uses large maps of (string → string)

C++ has this std::map class that does just what you want

```
typedef std::map<std::string, std::string> dictionary_t;
```

```
typedef std::map<std::string, std::string> dictionary_t;
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

```
#include <C++>
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

One copy during function return

#include <C++>

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

One copy during function return

Another copy during assignment operator

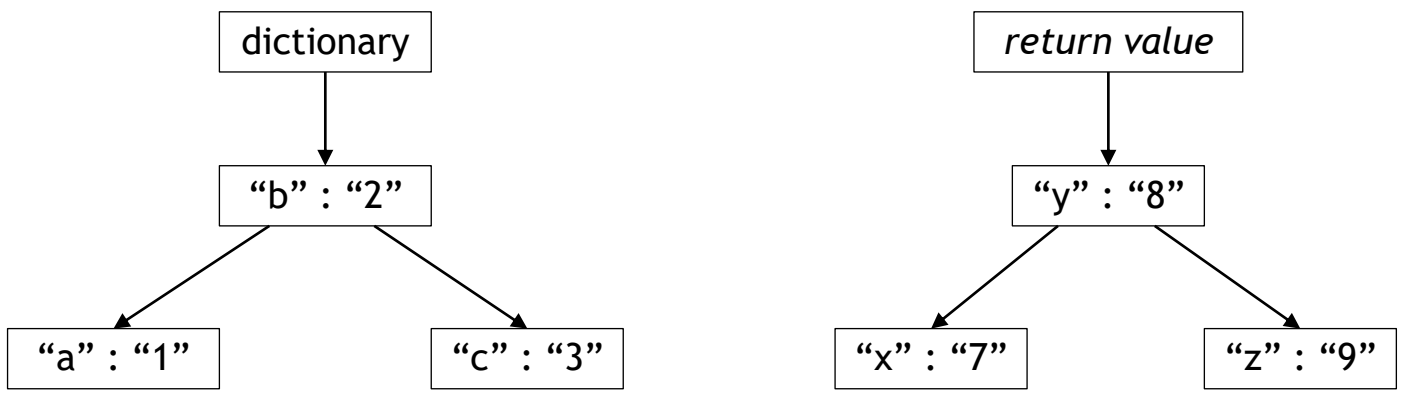
#include <C++>

Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Before assignment

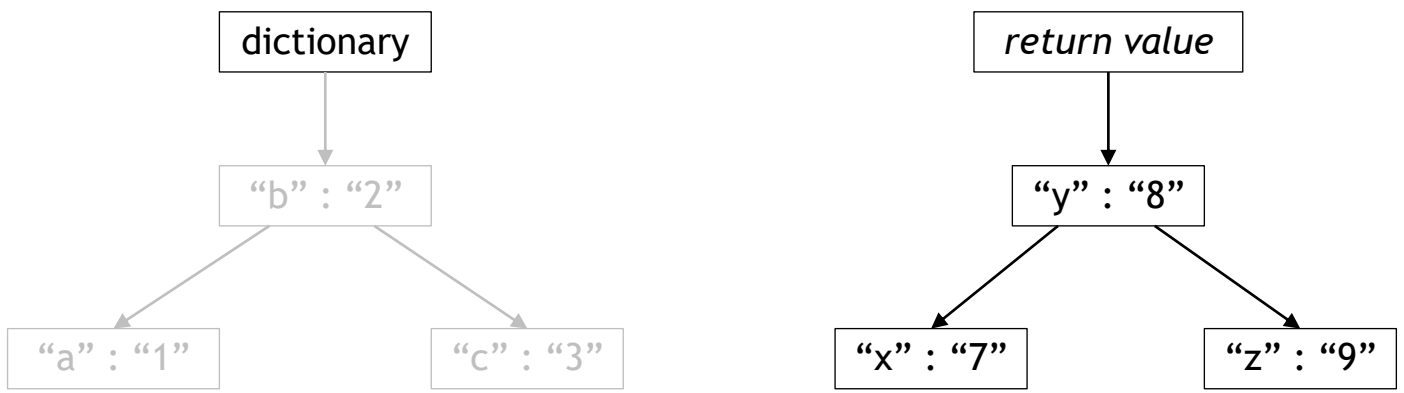


Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy old value

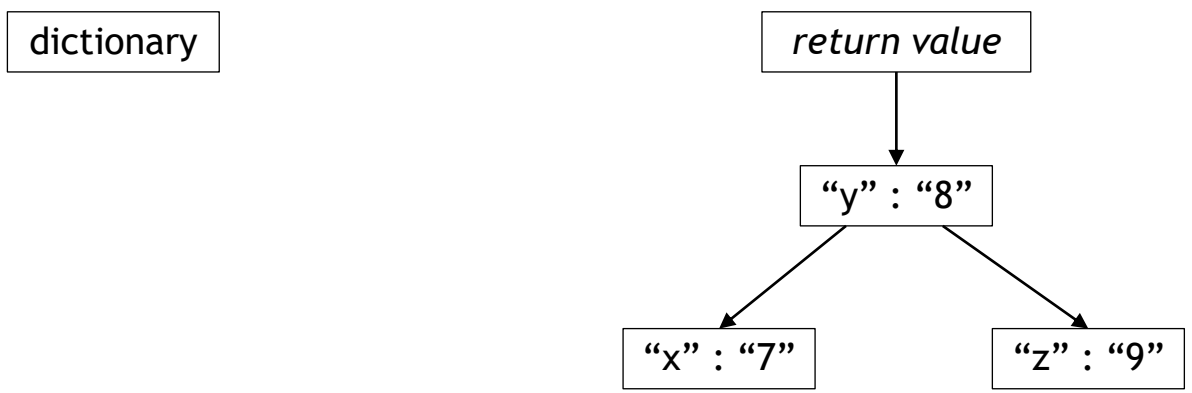


Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy old value

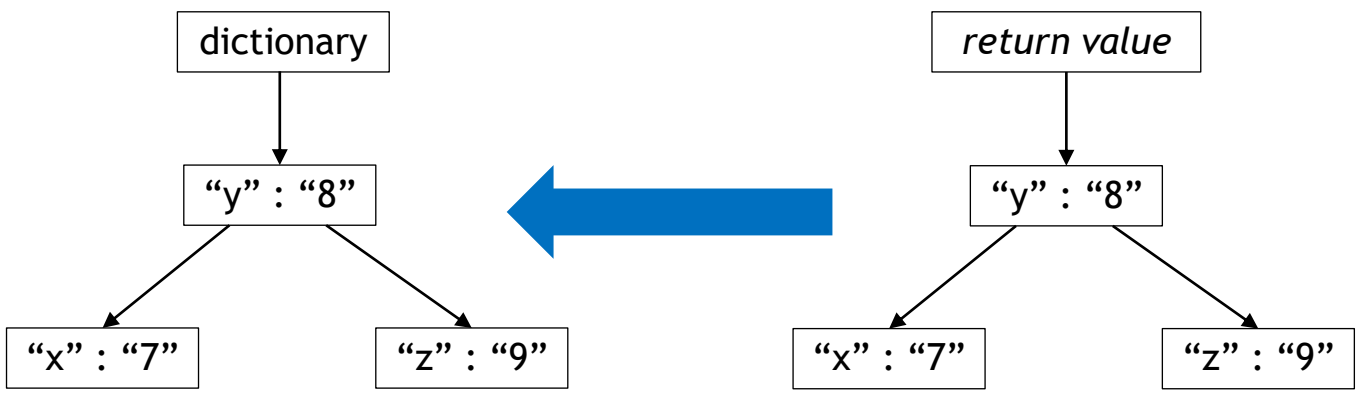


Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Copy the tree

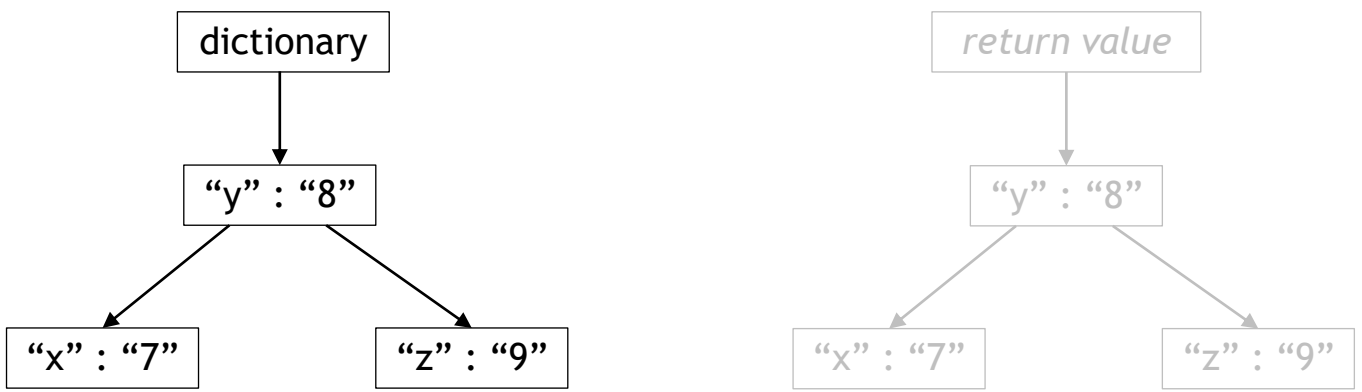


Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy the temporary

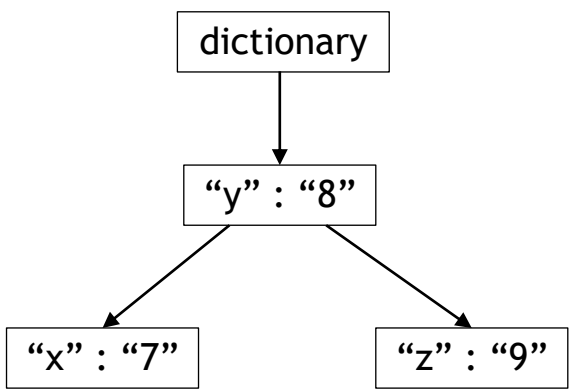


Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy the temporary

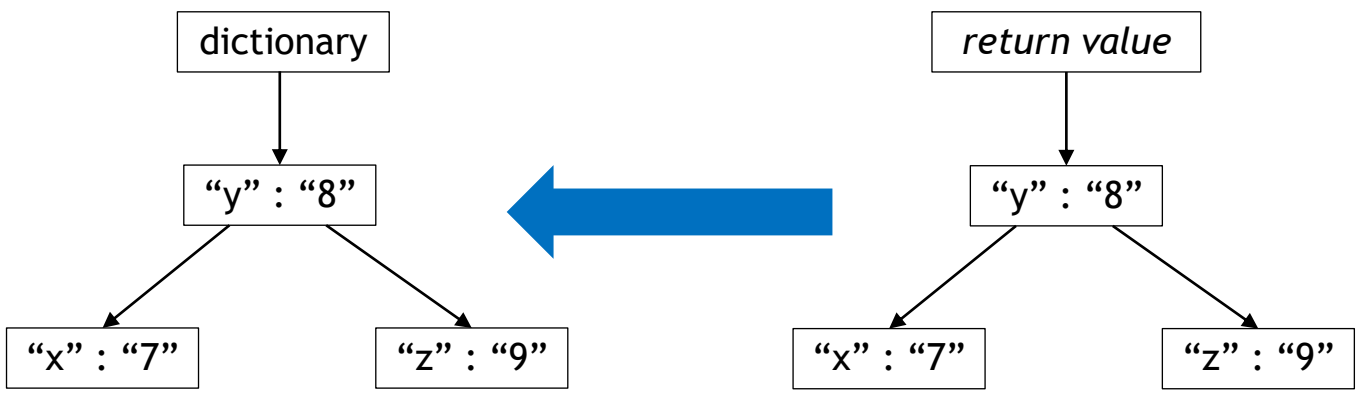


Assignment operator

Copy

```
dictionary = build_dictionary(getCustomerDb());
```

Copy the tree

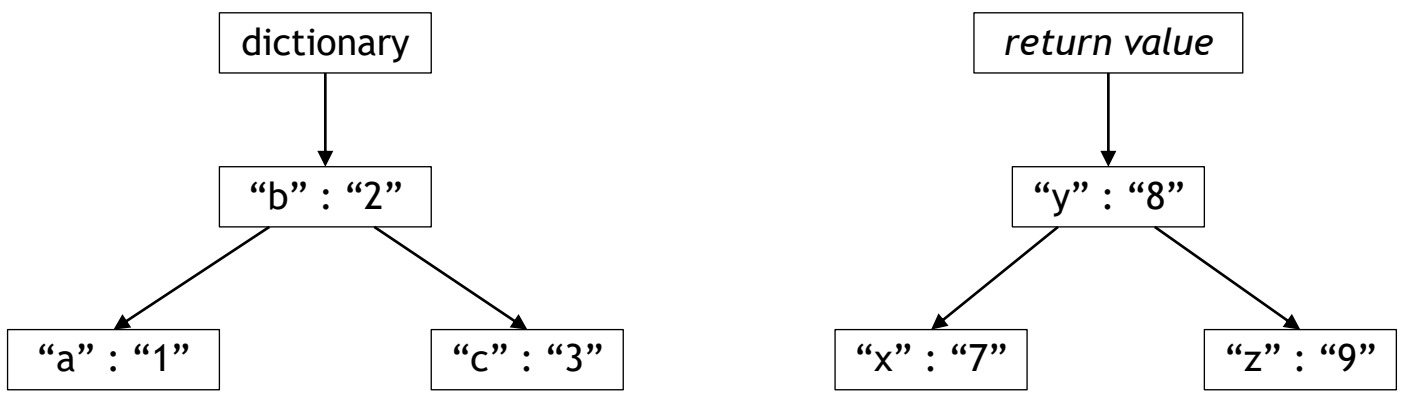


Assignment operator

Move

```
dictionary = build_dictionary(getCustomerDb());
```

Before assignment

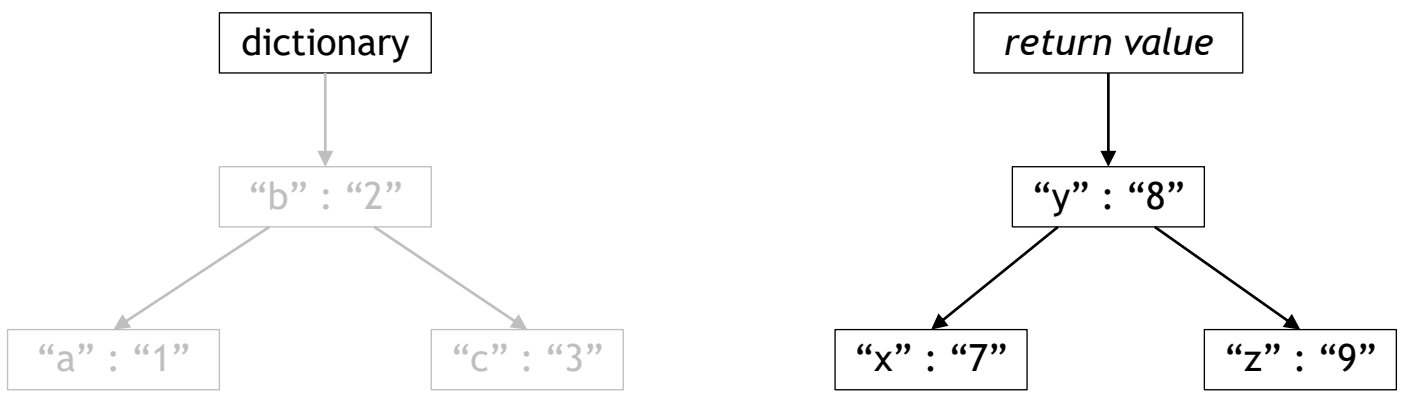


Assignment operator

Move

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy old value

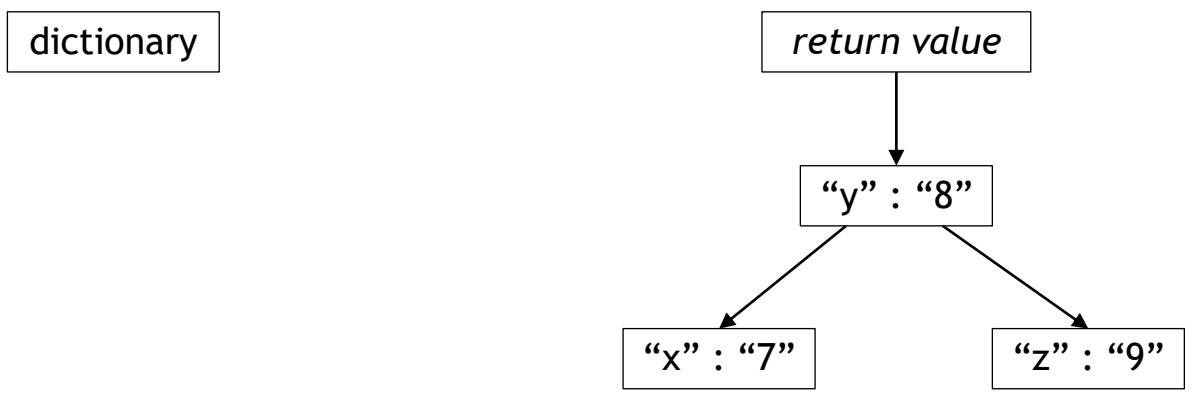


Assignment operator

Move

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy old value

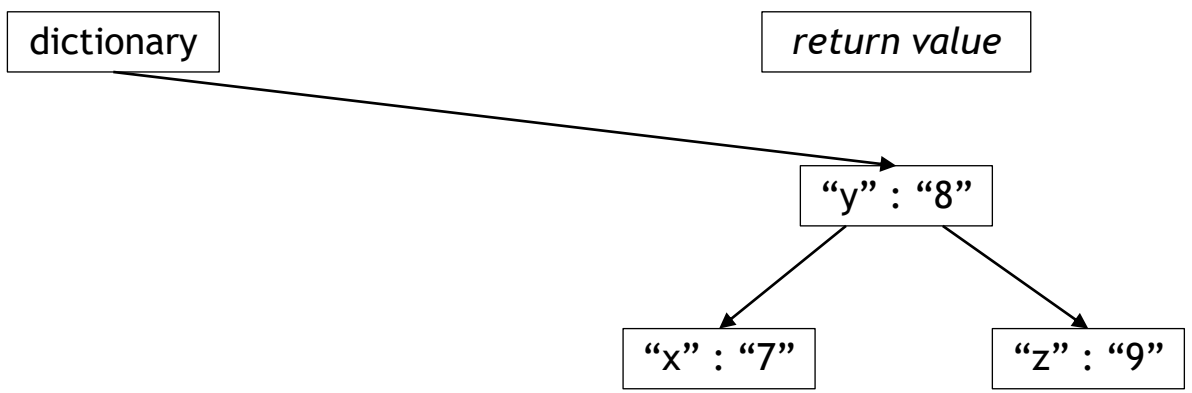


Assignment operator

Move

```
dictionary = build_dictionary(getCustomerDb());
```

Move the tree

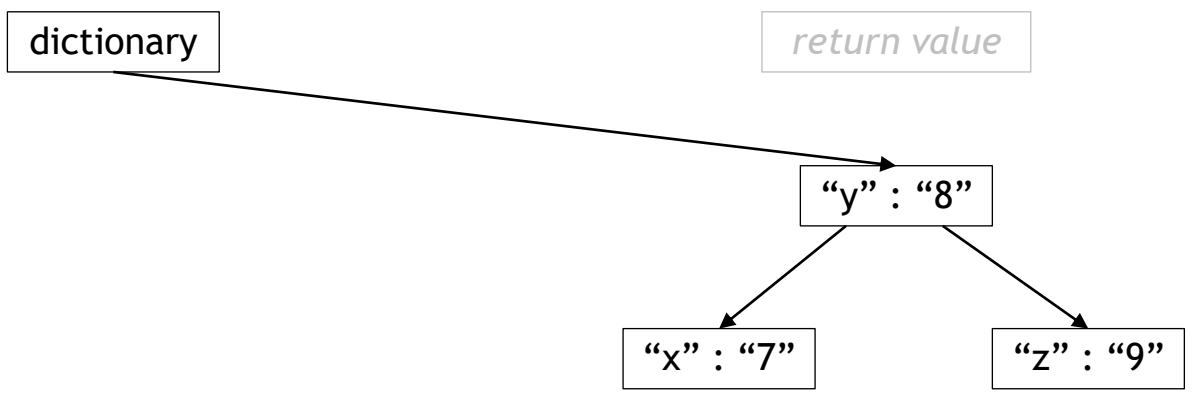


Assignment operator

Move

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy the temporary

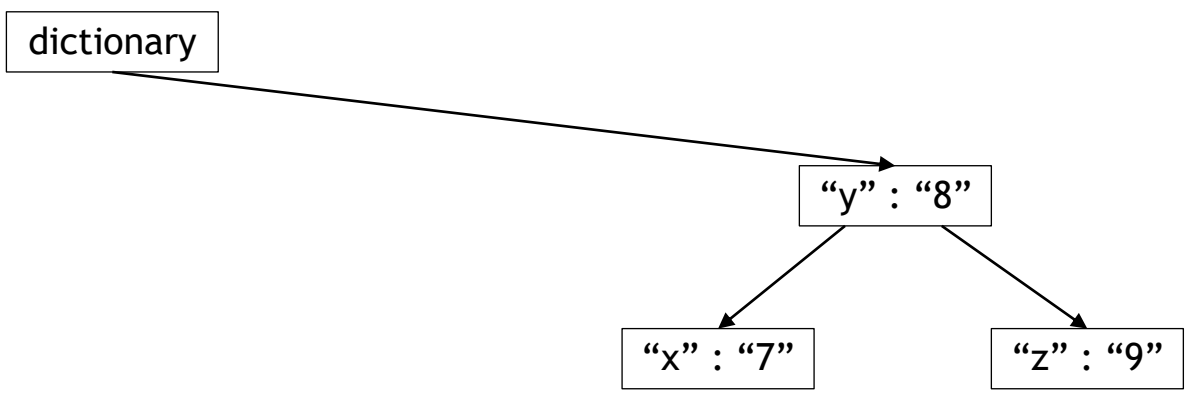


Assignment operator

Move

```
dictionary = build_dictionary(getCustomerDb());
```

Destroy the temporary



Expensive Copies

Some copies are expensive

C++98 has no mechanism to do a move rather than a copy

Move Semantics

Quick summary

C++11 introduced *move semantics*

Move constructor and move assignment operator:

- Transfer ownership, a.k.a. *move from*, rather than copy
- Used when the source object is an rvalue
- Can result in more efficient code

Move Semantics

Quick summary

Rvalue references, `T&&`, only bind to rvalues

`std::move` turns any expression into an rvalue

Move constructor / assignment:

- One rvalue reference parameter
- Transfer ownership of resources
- Leave the source object in a valid state

RVALUE REFERENCES

Lvalue

An lvalue is, roughly, something:

- that can appear on the left side of an assignment
- with a name
- with an address

Lvalue

Examples

```
int var;
var = 52;

const std::string hello = “Hello”;
std::string greeting = hello;

char* buffer = allocate_buffer();
*buffer = ‘\0’;

std::array<WidgetHandle, 300> widgets;
draw(widgets[123]);
```

Lvalue

Examples

```
int var;
var = 52;

const std::string hello = “Hello”;
std::string greeting = hello;

char* buffer = allocate_buffer();
*buffer = ‘\0’;

std::array<WidgetHandle, 300> widgets;
draw(widgets[123]);
```

An obvious lvalue

Lvalue

Examples

```
int var;
var = 52;

const std::string hello = “Hello”;
std::string greeting = hello;

char* buffer = allocate_buffer();
*buffer = ‘\0’;

std::array<WidgetHandle, 300> widgets;
draw(widgets[123]);
```


const lvalues are still lvalues

Lvalue

Examples

```
int var;
var = 52;

const std::string hello = “Hello”;
std::string greeting = hello;

char* buffer = allocate_buffer();
*buffer = ‘\0’;

std::array<WidgetHandle, 300> widgets;
draw(widgets[123]);
```

Rvalue

An rvalue is, roughly, anything that is not an lvalue

- Temporary objects
- Literal constants
- Function return values (that aren't lvalue references)
- Results of built-in operators (that aren't lvalues)

Rvalues usually have a short lifetime

Lvalue References

C++98 reference types are lvalue references

Non-const lvalue reference can only bind to an lvalue

```
int& r0 = page_count; // okay, variable is an lvalue
int& r1 = data[20];   // okay, array subscript is an lvalue
int& r2 = year % 100; // error, an rvalue
int& r3 = v.size();   // error, function return is often an rvalue
```

Const lvalue reference can bind to anything

```
const int& r2 = year % 100; // okay to bind to an rvalue
const int& r3 = v.size();   // okay to bind to an rvalue
```

Rvalue References

Syntax

Double ampersand, “&&”, in the declarator means rvalue reference type

```
void func( foo&& arg );
```

Type of “arg” is “foo&&”, or “rvalue reference to foo”, or “foo ref ref”

Rvalue References

Semantics

An rvalue reference can only bind to an rvalue

```
void func( int&& x );

int variable;
func( variable );

func( 42 );

func( v.size() );
```

Rvalue References

Semantics

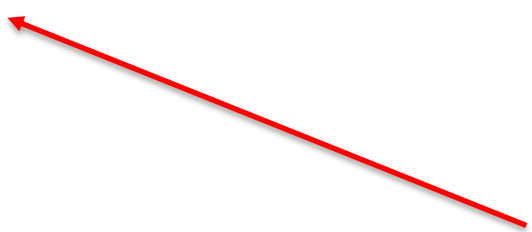
An rvalue reference can only bind to an rvalue

```
void func( int&& x );

int variable;
func( variable );

func( 42 );

func( v.size() );
```



Function parameter is rvalue reference to int

Rvalue References

Semantics

An rvalue reference can only bind to an rvalue

```
void func( int&& x );

int variable;
func( variable );

func( 42 );

func( v.size() );
```

Error: argument is an lvalue

Rvalue References

Semantics

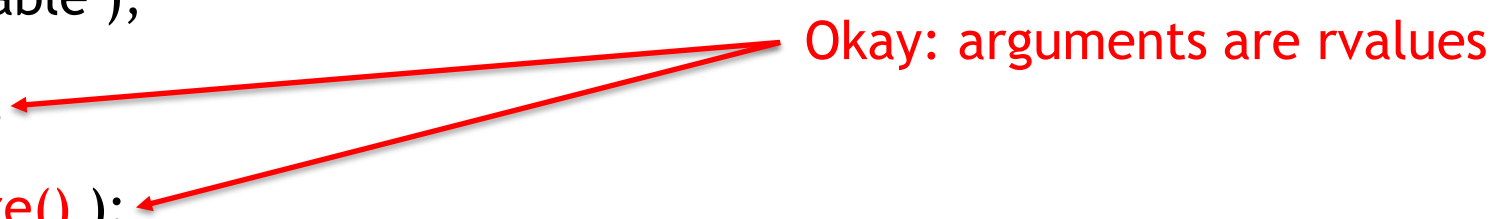
An rvalue reference can only bind to an rvalue

```
void func( int&& x );

int variable;
func( variable );

func( 42 );

func( v.size() );
```



Rvalue References

Semantics

The use of an rvalue reference is an lvalue

```
void f( int&& );
void g( int&& x ) {
  f(x);
}
void h() {
  g(42);
}
```

Rvalue References

Semantics

The use of an rvalue reference is an lvalue

```
void f( int&& );
void g( int&& x ) {
  f(x);
}
void h() {
  g(42);
}
```

Okay: 42 in an rvalue

Rvalue References

Semantics

The use of an rvalue reference is an lvalue

```
void f( int&& );
void g( int&& x ) {
  f(x);
}
void h() {
  g(42);
}
```

Error: Use of “x” is an lvalue

```
r.cpp: In function ‘void g(int&&)’:
r.cpp:3:5: error: cannot bind rvalue reference of type ‘int&&’ to lvalue of type ‘int’
   f(x);
     ^
r.cpp:1:6: note:   initializing argument 1 of ‘void f(int&&)’
 void f( int&& );
      ^
```

```
#include <C++>
```

Rvalue References

Guidelines

Guideline: **No rvalue reference to const type**

Use a non-const rvalue reference instead

Most uses of rvalue references modify the object being referenced

- Many examples of that later in the presentation

Most rvalues are not const

Rvalue References

Guidelines

Guideline: **No rvalue reference as function return type**

Core Guideline F.45: Don't return a T&&

Return by value instead

Rvalue references often bind to temporaries, which don't outlive the function

```
int&& func() { return 42; }
void test() {
  int a = func();
}
```

Rvalue References

Guidelines

Guideline: **No rvalue reference as function return type**

Core Guideline F.45: Don't return a T&&

Return by value instead

Rvalue references often bind to temporaries, which don't outlive the function

```
int&& func() { return 42; }
void test() {
  int a = func();
}
```

Rvalue References

Guidelines

Guideline: **No rvalue reference as function return type**

Core Guideline F.45: Don't return a T&&

Return by value instead

Rvalue references often bind to temporaries, which don't outlive the function

```
int&& func() { return 42; }
void test() {
  int a = func();
}
```

Temporary is destroyed before value is copied into "a"

Rvalue References

Guidelines

Guideline: **No rvalue reference as function return type**

Core Guideline F.45: Don't return a T&&

Return by value instead

Rvalue references often bind to temporaries, which don't outlive the function

```
int&& func() { return 42; }
void test() {
  int a = func();
}
r.cpp: In function ‘int&& func()’:
r.cpp:1:23: warning: returning reference to temporary [-Wreturn-local-addr]
```


STD::MOVE

std::move

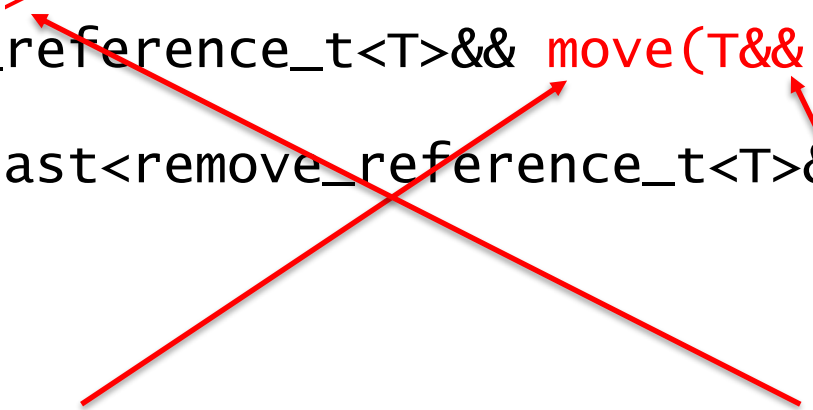
Definition

```
template <class T>
constexpr remove_reference_t<T>&& move(T&& t) noexcept
{
  return static_cast<remove_reference_t<T>&&>(t);
}
```

std::move

Definition

```
template <class T>
constexpr remove_reference_t<T>&& move(T&& t) noexcept
{
  return static_cast<remove_reference_t<T>&&>(t);
}
```



constexpr function taking one argument…

… of any type or value category

std::move

Definition

```
template <class T>
constexpr remove_reference_t<T>&& move(T&& t) noexcept
{
  return static_cast<remove_reference_t<T>&&>(t);
}
```


Returns an rvalue reference

std::move

Definition

```
template <class T>
constexpr remove_reference_t<T>&& move(T&& t) noexcept
{
  return static_cast<remove_reference_t<T>&&>(t);
}
```

static_cast the argument to its corresponding rvalue reference type

std::move

Definition

```
template <class T>
constexpr remove_reference_t<T>&& move(T&& t) noexcept
{
  return static_cast<remove_reference_t<T>&&>(t);
}
```

`std::move` doesn't move anything!

It converts any expression into an rvalue so it can be bound to an rvalue reference

std::move

Usage

Use std::move to convert an lvalue to an rvalue

- So it will bind to an rvalue reference
- So that object will be moved from rather than copied

std::move

Guidelines

Guideline: **Next operation after std::move is destruction or assignment**

or reset to a known value by other means, such as `vector<T>::clear()`

std::move

Guidelines

Guideline: **Next operation after std::move is destruction or assignment**

or reset to a known value by other means, such as `vector<T>::clear()`

```
container long_lived = ...;
{
  container possible = ...;
  if (...) {
    long_lived = std::move(possible);
  }
}
```

std::move

Guidelines

Guideline: **Next operation after std::move is destruction or assignment**

or reset to a known value by other means, such as `vector<T>::clear()`

```
std::vector<std::string> v;
std::string str = "Hello";

v.push_back(std::move(str));
str = "World";
```

std::move

Guidelines

Guideline: **Next operation after std::move is destruction or assignment**

or reset to a known value by other means, such as `vector<T>::clear()`

```
std::vector<std::string> v;
std::string str = "Hello";

v.push_back(std::move(str));
str += "World";
```

std::move

Guidelines

Guideline: **Next operation after std::move is destruction or assignment**

or reset to a known value by other means, such as `vector<T>::clear()`

```
std::vector<std::string> v;
std::string str = "Hello";

v.push_back(std::move(str));
str += "World";
```

Uses value after move

Don’t do this!

std::move

Guidelines

Guideline: **Next operation after std::move is destruction or assignment**

or reset to a known value by other means, such as `vector<T>::clear()`

Assume that object referred to by rvalue reference will be destroyed or assigned to

```
void maybe_insert(std::vector<std::string>& v,
                  std::string&& str) {
  if (some_condition()) {
    v.push_back(std::move(str));
  }
}
```

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

Core Guideline F.48: Don't `return std::move(local)`

C++ Standard has a special rule for this:

> The return expression is an rvalue if it is a local variable or parameter

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

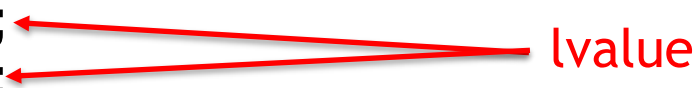
```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```

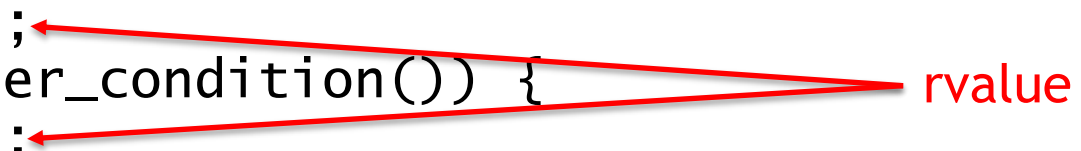


std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```



std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```

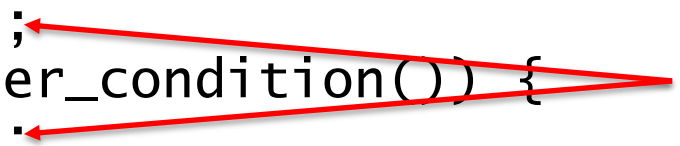
lvalue

std::move

Guidelines

Guideline: **Don't `std::move` the return of a local variable**

```
std::string func( std::string param, std::string* ptr ) {
  std::string local = "Hello"s;
  *ptr = param;
  *ptr = local;
  if (some_condition()) {
    return param;
  } else if (other_condition()) {
    return local;
  }
  return *ptr;
}
```



rvalue, because they are about to be destroyed

Move Constructor
Move Assignment

Move Constructor / Assignment

Similar to copy constructor/assignment, except parameter is an rvalue reference

```
struct foo {
  foo( foo&& ) noexcept;
  foo& operator=( foo&& ) noexcept;
  // ...
};
```

Move Constructor

Implicitly declared

Implicitly declared move constructor if there are no user-declared:

- destructor
- copy constructor
- copy assignment operator
- move assignment operator

Move Constructor

Default definition

Implicitly declared or `=default` move constructor

- Move constructs each base and non-static data member
- Deleted if any base or non-static data member cannot be move constructed

Move Assignment Operator

Implicitly declared

Implicitly declared move assignment operator if there are no user-declared:

- destructor
- copy constructor
- copy assignment operator
- move constructor

Move Assignment Operator

Default definition

Implicitly declared or `=default` move assignment operator

- Move assigns each base and non-static data member
- Deleted if any base or non-static data member cannot be move assigned

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Use subojbect's move constructor when possible

Explicit transfer of resources otherwise

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Use subojbect's move constructor when possible

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept ...
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Use subojbect's move constructor when possible

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept : a(std::move(other.a)),
                            b(std::move(other.b))
  { }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Use subojbect's move constructor when possible

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept : a(          other.a ),
                            b(std::move(other.b))
  { }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Use subojbect's move constructor when possible

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept : a(std::move(other.a)),
                            b(std::move(other.b))
  { }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Use subojbect's move constructor when possible

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept = default;


};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept ...
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept : data(std::move(other.data)) {
    ...
  }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept : data(          other.data ) {
    ...
  }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept : data(std::move(other.data)) {
    ...
  }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept : data(std::move(other.data)) {
    other.data = nullptr;
  }
};
```

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept
      : data(std::exchange(other.data, nullptr))
  { }
};
```

std::exchange

```
template <class T, class U = T>
T exchange( T& object, U&& value );
```

Assigns `value` to `object`

Returns the old value of `object`

Useful when implementing move operations

Move Constructor

How to write one

Transfer ownership of resources from existing object to object being constructed

Explicit transfer of resources otherwise

```
struct S {
  double* data;

  S( S&& other ) noexcept
      : data(std::exchange(other.data, nullptr))
  { }
};
```

Move Constructor

How to write one

```
struct S {
  double* data;

  S( S&& other ) noexcept
      : data(std::exchange(other.data, nullptr))
  { }
};
```

Assumes that `data == nullptr` is a valid state

If not, then a move constructor is not possible

Move Constructor

How to write one

```
struct S {
  double* data; // Invariant: data != nullptr

  S( S&& other ) noexcept = delete;


};
```

Assumes that `data == nullptr` is a valid state

If not, then a move constructor is not possible

Move Assignment Operator

How to write one

Free resources owned by assigned-to object

Transfer ownership of resources

- Use subobject's move assignment operator when possible

Move Assignment Operator

How to write one

```
struct S {
  int a;
  std::string b;

  S& operator=( S&& other ) noexcept {
    ...
    return *this;
  }
};
```

Move Assignment Operator

How to write one

```
struct S {
  int a;
  std::string b;

  S& operator=( S&& other ) noexcept {
    a = std::move(other.a);
    b = std::move(other.b);
    return *this;
  }
};
```

Move Assignment Operator

How to write one

```
struct S {
  int a;
  std::string b;

  S& operator=( S&& other ) noexcept = default;



};
```

Move Assignment Operator

How to write one

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    ...
    return *this;
  }
};
```

Move Assignment Operator

How to write one

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    ...
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};
```

Move Assignment Operator

How to write one

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};
```

Move Assignment Operator

How to write one

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};
```

Not quite done… We'll come back to this later.

Move Copy / Assignment

Guidelines

Guideline: **Move constructor / assignment should be explicitly noexcept**

Core Guideline C.66: Make move operations noexcept

Moves are supposed to transfer resources, not allocate or acquire resources

Other code depends on noexcept move to implement strong exception guarantee

Declare it `noexcept` even when it is defined as default

```
foo( foo&& ) noexcept = default;
```

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

Core Guideline C.64: A move operation should move and leave its source in a valid state

Prefer to leave it in the default constructed state

But that is not always practical

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

```
struct S {
  std::string str;
  std::size_t len;
  // Invariant: len == str.length()

  S( S&& other ) noexcept ...


};
```

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

```
struct S {
  std::string str;
  std::size_t len;
  // Invariant: len == str.length()

  S( S&& other ) noexcept : str(std::move(other.str)),
                            len(std::move(other.len))
  { }
};
```

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

```
struct S {
  std::string str;
  std::size_t len;
  // Invariant: len == str.length()

  S( S&& other ) noexcept : str(std::move(other.str)),
                            len(std::move(other.len))
  { }
};
```

`other.str` has unknown value after move. Might not match `other.len`

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

```
struct S {
  std::string str;
  std::size_t len;
  // Invariant: len == str.length()

  S( S&& other ) noexcept : str(std::move(other.str)),
                            len(std::move(other.len)) {
    other.len = other.str.length();
  }
};
```

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

```
struct S {
  std::string str;
  std::size_t len;
  // Invariant: len == str.length()

  S( S&& other ) noexcept : str(std::move(other.str)),
                            len(std::move(other.len)) {
    other.str.clear();
    other.len = 0;
  }
};
```

Move Copy / Assignment

Guidelines

Guideline: **Moved-from object must be left in a valid state**

```
struct S {
  std::string str;
  std::size_t len;
  // Invariant: len == str.length()

  S( S&& other ) noexcept
      : str(std::exchange(other.str, std::string())),
        len(std::exchange(other.len, 0))
  { }
};
```

Move Copy / Assignment

Guidelines

Guideline: **Use `=default` when possible**

Core Guideline C.80: Use `=default` if you have to be explicit about using the default semantics

Move Copy / Assignment

Guidelines

Guideline: **Use =default when possible**

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept : a(std::move(other.a)),
                            b(std::move(other.b))
  { }
};
```

Move Copy / Assignment

Guidelines

Guideline: **Use =default when possible**

```
struct S {
  int a;
  std::string b;

  S( S&& other ) noexcept = default;


};
```

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

Core Guideline C.65: Make move assignment safe for self-assignment

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};
```

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};

  s = std::move(s);
```

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};

  s = std::move(s);
```

*this and other are the same object

Delete the memory

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};

  s = std::move(s);
```

*this and other are the same object

data doesn't change

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};

  s = std::move(s);
```

*this and other are the same object

data points to delete memory

Move Assignment

Guidelines

Guideline: **Make move assignment safe for self-assignment**

```
struct S {
  double* data;

  S& operator=( S&& other ) noexcept {
    if (this == &other) return *this;
    delete[] data;
    data = std::exchange(other.data, nullptr);
    return *this;
  }
};
```

Move Copy / Assignment

Guidelines

Guideline: **Rule of 5 / Rule of 0**

Core Guideline C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all

(1) destructor, (2) copy constructor, (3) copy assignment operator, (4) move constructor, (5) move assignment operator

Rule of 0: If default behavior is correct for all five, let compiler do everything

Rule of 5: If you must define one of the five, declare all of them explicitly

Other Uses

Overload on Lvalue/Rvalue Reference

vector<T>::push_back has two overloads

```
void push_back( const T& value );
void push_back( T&& value );
```

Overload on Lvalue/Rvalue Reference

`vector<T>::push_back` has two overloads

```
void push_back( const T& value );
void push_back( T&& value );
```

This is a common pattern

Use it when parameter will be copied, and copies are more expensive than moves

Overload on Lvalue/Rvalue Reference

vector<T>::push_back has two overloads

```
void push_back( const T& value );

void push_back( T&& value );
```

```
std::vector<std::string> v;
std::string str = “Hello”;

v.push_back(str);

v.push_back(getMyString());
```

Overload on Lvalue/Rvalue Reference

vector<T>::push_back has two overloads

```
void push_back( const T& value );

void push_back( T&& value );
```

```
std::vector<std::string> v;
std::string str = “Hello”;

v.push_back(str);

v.push_back(getMyString());
```

Calls (const T&) version, copies into the vector

Overload on Lvalue/Rvalue Reference

vector<T>::push_back has two overloads

```
void push_back( const T& value );

void push_back( T&& value );
```

```
std::vector<std::string> v;
std::string str = “Hello”;

v.push_back(str);

v.push_back(getMyString());
```

Call (T&&) version,
moves into the vector

Perfect Forwarding

Universal reference / forwarding reference

```
template <class T> void f(T&& value)
{
  g(std::forward<T>(value));
}
```

Benefits of Move Semantics

Expensive Copies

Expensive copies can be replaced by inexpensive moves

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

move assignment

```
typedef std::map<std::string, std::string> dictionary_t;

dictionary_t build_dictionary( DbConnection db )
{
  dictionary_t dictionary;
  if (!db.is_open()) return dictionary_t();
  // ... Fill in thousands of entries from database ...
  return dictionary;
}

void business_logic()
{
  dictionary_t dictionary;

  dictionary = build_dictionary(getSupplierDb());
  // ... Create report about suppliers ...

  dictionary = build_dictionary(getCustomerDb());
  // ... Create report about customers ...
}
```

No copies!

```
#include <C++>
```

Move-only types

Some types cannot be copied, but can be moved safely

Not possible in C++98; easy to implement with move semantics

For example, `unique_ptr<T>`

- Has a move constructor and move assignment operator
- Copy constructor and copy assignment operator are deleted

std::unique_ptr

Not copyable

```
void f( std::unique_ptr<int> );
std::unique_ptr<int> g();

void h() {
  std::unique_ptr<int> a;
  std::unique_ptr<int> b{a}; // error: copy constructor is deleted
  b = a;                     // error: copy assignment is deleted
  f(a);                      // error: copy constructor is deleted
}
```

```
unique.cpp:8:27: error: use of deleted function ‘std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = int; _Dp =
std::default_delete<int>]’
```

std::unique_ptr

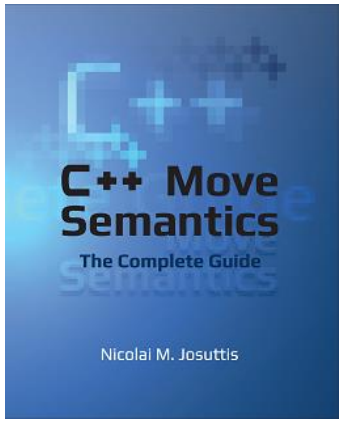
Movable

```
void f( std::unique_ptr<int> );
std::unique_ptr<int> g();

void h() {
  std::unique_ptr<int> a;
  std::unique_ptr<int> b{std::move(a)};
  b = std::move(a);
  f(std::move(a));
  f(g());
}
```

Summary

Resources



Nicolai M. Josuttis, *C++ Move Semantics: The Complete Guide*, http://www.cppmove.com/

C++ Core Guidelines
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html

Nicolai Josuttis, “The Hidden Secrets of Move Semantics”, CppCon 2020

Nicolai Josuttis, “The Nightmare of Move Semantics for Trivial Classes”, CppCon 2017
https://www.youtube.com/watch?v=PNRju6_yn3o

Move Semantics

Summary

Rvalue references, `T&&`, only bind to rvalues

`std::move` turns any expression into an rvalue

Move constructor / assignment:

- One rvalue reference parameter
- Transfer ownership of resources
- Leave the source object in a valid state

Move Semantics

Benefits

More efficient code: expensive copies → cheap moves

Move-only types are possible

Guidelines

No rvalue reference to const type

No rvalue reference as function return type

Next operation after `std::move` is destruction or assignment

Don't `std::move` the return of a local variable

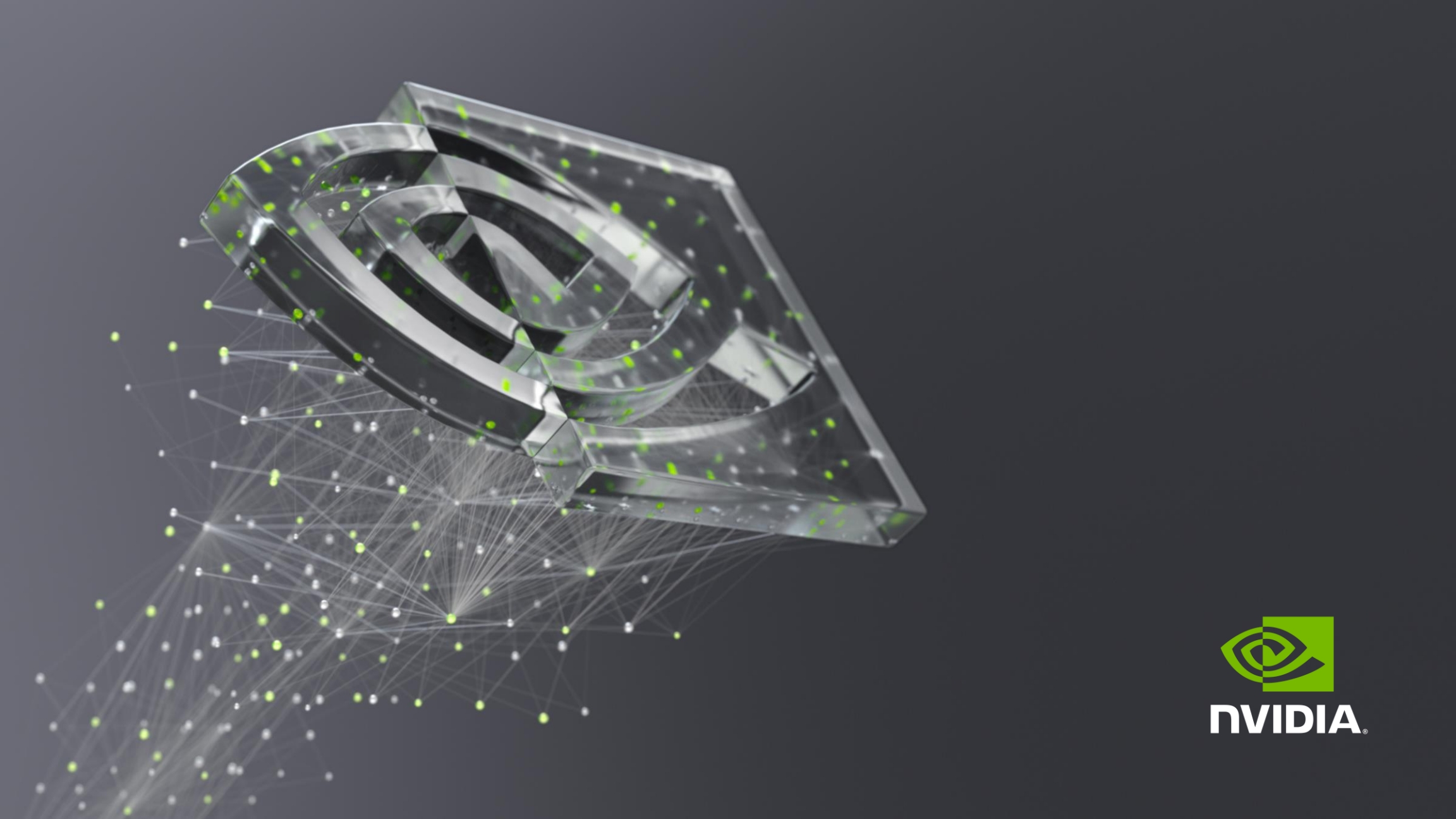
Move constructor / assignment should be explicitly `noexcept`

Moved-from object must be left in a valid state

Use `=default` when possible

Make move assignment safe for self-assignment

Rule of 5 / Rule of 0


NVIDIA