

**CS100**

# **Introduction to Programming**

**Lecture 25. Rvalue references and their use**

# Today's learning objectives

- Leading Example: Smart pointers
- Move semantics
- Lvalues and Rvalues
- Rvalue references
- Move construction and move assignment
- `std::move`
- Efficiency considerations

# Today's learning objectives

- **Leading Example: Smart pointers**
- Move semantics
- Lvalues and Rvalues
- Rvalue references
- Move construction and move assignment
- `std::move`
- Efficiency considerations

# Smart pointers: motivation

- Consider the following function:

```
void someFunction()
{
    Resource *ptr = new Resource; // Resource is a struct or class

    // do stuff with ptr here

    delete ptr;
}
```

- Problems:
  - Easy to forget deallocating ptr
  - Even if we don't, early exit could lead to leak

# Smart pointers: motivation

- Example:

```
#include <iostream>

void someFunction()
{
    Resource *ptr = new Resource;

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        return; // function returns early, ptr won't be deleted!

    // do stuff with ptr here

    delete ptr;
}
```

# What are the good things of classes?

- Contain destructors
  - Get executed when object goes out of scope
  - Can deallocate memory that is owned by the class
  - Guarantees memory is free'd up
  - RAI idiom:
    - Resource Acquisition Is Initialization
    - Resources are allocated upon object construction
    - Resources are de-allocated upon object destruction

# Smart Pointer

- A class
- Manages and cleans up pointers
  - It “holds” the pointer
  - It lets us use the pointer
  - It automatically cleans up frees the allocated memory upon destruction

# Smart pointer

- Example: `#include <iostream>`

```
template<class T>
class Auto_ptr1 {
    T* m_ptr;
public:
    // Pass in a pointer to "own" via the constructor
    Auto_ptr1(T* ptr=nullptr) : m_ptr(ptr) {}

    // The destructor will make sure it gets deallocated
    ~Auto_ptr1() {
        delete m_ptr;
    }

    // Overload dereference and operator->
    // lets us use Auto_ptr1 like m_ptr.
    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};
```



# Smart pointer

- Usage:

```
// A sample class to prove the above works
class Resource {
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};
```

```
int main() {
    // Allocation of memory here:
    Auto_ptr1<Resource> res(new Resource);

    // ... but no explicit delete needed

    // Note: Resource in angled braces doesn't need
    // a * symbol, it is supplied by the template
```

Output:  
Resource acquired  
Resource destroyed

res goes out of scope and  
is destroyed  
Destructer of Auto\_ptr1  
will delete Resource

```
return 0;
} // res goes out of scope here, and destroys
// allocated Resource for us
```

# Smart pointer

- Similarly:

```
void someFunction()
{
    Auto_ptr1<Resource> ptr(new Resource); // ptr now owns the Resource
    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;
    if (x == 0)
        return; // the function returns early
    // do stuff with ptr here
    ptr->sayHi();
}

int main()
{
    someFunction();

    return 0;
}
```

- If x is not 0, output will be:  
Resource acquired  
Hi!  
Resource destroyed
- If x is 0, output will be:  
Resource acquired  
Resource destroyed

# Auto-pointer flaw

- Consider the following code

```
int main() {  
    Auto_ptr1<Resource> res1(new Resource);  
    Auto_ptr1<Resource> res2(res1);  
  
    return 0;  
}
```

or

```
int main() {  
    Auto_ptr1<Resource> res1(new Resource);  
    Auto_ptr1<Resource> res2;  
    res2 = res1;  
  
    return 0;  
}
```

- Hits

Resource acquired  
Resource destroyed  
Segmentation fault

# Auto-pointer flaw

- C++ automatically provides
  - copy-constructor
  - assignment operator
  - Are simply copying over the pointer inside Auto-pointer
    - shallow copy

# Auto-pointer flaw

- Consider the following code

```
int main() {  
    Auto_ptr1<Resource> res1(new Resource);  
    Auto_ptr1<Resource> res2(res1);  
  
    return 0;  
}
```

Both res1 and res2 are now pointing at the same resource

- ←
- res1 and res2 go out of scope
  - the first one deletes Resource already
  - the second one aims at doing so too → crash

or

```
int main() {  
    Auto_ptr1<Resource> res1(new Resource);  
    Auto_ptr1<Resource> res2;  
    res2 = res1;  
  
    return 0;  
}
```

# Auto-pointer flaw

- Similar problem is caused by this:

```
void passByValue (Auto_ptr1<Resource> res) {}
```

```
int main() {  
    Auto_ptr1<Resource> res1 (new Resource) ;  
    passByValue (res1)  
    return 0 ;  
}
```

- Call to passByValue will make a (shallow) copy of res1
- passByValue will use this copy
- passByValue will destroy this copy upon completion of the function body
- this will destroy the resource!

# Solution 1

- Prevent the copy constructor and the assignment operator to be “available”, thereby preventing copying and assignment altogether
- Solution 1, method 1 (the old way):
  - Explicitly declare them and make them private!


# Solution 1, method 1

```
template<class T>
class Auto_ptr1 {
T* m_ptr;
public:
    // Pass in a pointer to "own" via the constructor
    Auto_ptr1(T* ptr=nullptr) : m_ptr(ptr) {}

    // The destructor will make sure it gets deallocated
    ~Auto_ptr1() {
        delete m_ptr;
    }

    // ...

private:
    Auto_ptr1( const Auto_ptr1 & );
    Auto_ptr1& operator= ( const Auto_ptr1 & );
};
```



Explicit, private  
declaration



# Solution 1, method 1

- What are the problems with this approach?
  - Automatic generation of default constructor does not happen (not a problem here)
  - The explicitly defined default constructor is less efficient than the automatically provided default constructor
  - Member functions and friends can still call the privately defined copy-constructor etc.
    - → Linking error if declared but not defined
  - Unclear

# Solution 1

- Prevent the copy constructor and the assignment operator to be “available”, thereby preventing copying and assignment altogether
- Solution 1, method 2 (the C++11 way):

```
struct noncopyable {  
    noncopyable() = default;  
    noncopyable(const noncopyable&) = delete;  
    noncopyable& operator=(const noncopyable&) = delete;  
};
```

- Bringing back default constructor
- No performance loss!

- Explicitly deleted
- Calling them would lead to a compile time error!

# Solution 1, method2

```
template<class T>
class Auto_ptr1 {
T* m_ptr;
public:
    // Pass in a pointer to "own" via the constructor
    Auto_ptr1(T* ptr=nullptr) : m_ptr(ptr) {}

    // The destructor will make sure it gets deallocated
    ~Auto_ptr1() {
        delete m_ptr;
    }

    Auto_ptr1( const Auto_ptr1 & ) = delete;
    Auto_ptr1& operator= ( const Auto_ptr1 & ) = delete;

    ...
};
```

# Auto-pointer

- Consequence:
  - Pass by value is no longer possible  
→ Not a big problem, just pass by reference
  - However, how to do this?

```
??? generateResource() {  
    Resource *r = new Resource;  
    return Auto_ptr1(r);  
}
```

- Can't return by reference! → Object will be destroyed at the end of generateResource
- Can't return by value → copy-constructor is no longer available
- Can't just return pointer r, because that would defy the entire purpose of using the smart pointers

# Auto-pointer

- What about just doing deep copies?
  - Not desirable
  - We just want to return a pointer from a function

# Today's learning objectives

- Leading Example: Smart pointers
- **Move semantics**
- Lvalues and Rvalues
- Rvalue references
- Move construction and move assignment
- `std::move`
- Efficiency considerations

# What are move-semantics?

- Do not copy values!
- Simply move ownership!

# Example: Auto-pointer 2

```
#include <iostream>

template<class T>
class Auto_ptr2 {
T* m_ptr;
public:
    Auto_ptr2(T* ptr=nullptr) : m_ptr(ptr) {}
    ~Auto_ptr2() {
        delete m_ptr;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }

```

...



# Example: Auto-pointer 2

*// Copy constructor with move semantics*

```
Auto_ptr2(Auto_ptr2 &a) { // note: not const anymore!  
    m_ptr = a.m_ptr;           // transfer dumb pointer  
    a.m_ptr = nullptr;        // remove ownership from source  
}
```

*// Assignment operator with move semantics*

```
Auto_ptr2&  
operator=(Auto_ptr2& a) { // note: not const anymore!  
    if (&a == this)  
        return *this;  
  
    delete m_ptr;           // deallocate ptr "this" already holds  
    m_ptr = a.m_ptr;        // transfer dumb pointer  
    a.m_ptr = nullptr;      // remove ownership from source  
    return *this;  
}  
};
```

# Example: Auto-pointer 2

- Usage

```
class Resource {  
    public:  
        Resource() { std::cout << "Resource acquired\n"; }  
        ~Resource() { std::cout << "Resource destroyed\n"; }  
};
```

# Example: Auto-pointer 2

- Usage

```
int main() {  
    Auto_ptr2<Resource> res1(new Resource);  
    Auto_ptr2<Resource> res2; // Start as nullptr  
  
    std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");  
    std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");  
  
    res2 = res1; // res2 assumes ownership, res1 is set to null  
  
    std::cout << "Ownership transferred\n";  
    std::cout << "res1 is " << (res1.isNull() ? "null\n" : "not null\n");  
    std::cout << "res2 is " << (res2.isNull() ? "null\n" : "not null\n");  
  
    return 0;  
}
```

## Output:

Resource acquired  
res1 is not null  
res2 is null  
Ownership transferred  
res1 is null  
res2 is not null  
Resource destroyed

# Auto-pointer 2

- Remaining problems:
  - `std::auto_ptr` is implemented exactly like this in the original C++98 standard
  - Problems occur if passed by value
  - Deprecated and to be fully removed since C++17
- General problems with move-semantics pre C++11:
  - Copy semantics and move semantics can't be differentiated
  - Overwriting copy-constructor leads to weird behavior
  - `res1 = res2` is unclear

# Today's learning objectives

- Leading Example: Smart pointers
- Move semantics
- **Lvalues and Rvalues**
- Rvalue references
- Move construction and move assignment
- `std::move`
- Efficiency considerations

# Lvalues and Rvalues

- Every expression has two properties
  - A type
  - A “value” property
    - Checks for example whether an expression can be assigned to
- Do not denote properties of values per se, but properties of expressions

# Lvalues and Rvalues

type  
└─┬─  
`int thisNumber = 2 + anotherNumber;`  
└──────────┬──────────  
expression expression

- Historically:
  - Lvalues suitable for left side of assignment expression
  - Rvalues suitable for right side of assignment expression

# Distinguish Lvalues and Rvalues

- Every expression either an lvalue or an rvalue
  - Terms historical, now essentially arbitrary
- Lvalues:
  - Refer to objects accessible at more than one point in source code
    - Named objects + objects accessible via pointers/references (locator values)
    - **General rule: If you can take its address, it's an lvalue**
  - Conceptual motivation: Lvalues may not be moved from
- Rvalues:
  - Everything that is not an lvalue
  - Refer to objects accessible at exactly one point in source code
    - Conceptually: temporary objects, e.g., by-value function returns
      - Unnamed + pointers/references to them not possible
  - Conceptual motivation: Rvalues may be moved from



# Lvalues and Rvalues

```
int x;
```

```
x = 10;
```

- **x** is lvalue:
  - Named object, address can be taken
  - Could be referred to multiple times
- **10** is rvalue:
  - Can't take its address
  - All build-in numeric literals are rvalues

# Lvalues and Rvalues

```
template<typename T1, typename T2>           //return size
int sizeDiff(const T1& c1, const T2& c2)    //difference of
{return c1.size() - c2.size();}            //two containers
```

- `sizeDiff, c1, c2` are lvalues:
  - Named objects, addresses can be taken
  - Could be referred to multiple times
- Return value is rvalue
  - Address can't be taken.
  - No way to get pointer/reference to it

# Lvalues and Rvalues

```
int *px;  
std::vector<int> v;  
std::unordered_set<int> s;  
...  
*px = sizeDiff(v,s);
```

- **\*px** is an lvalue:
  - Address can be taken
  - Pointed-to object could be referred to multiple times
- **sizeDiff(v,s)** value is rvalue
  - Address can't be taken
  - No way to get pointer/reference to it

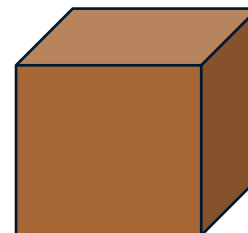
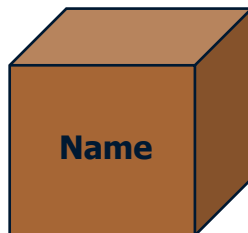
# Lvalues and Rvalues

```
std::ifstream myinput( std::string("/some/path.txt") );
```

- Anonymous objects are rvalues

# Lvalues and Rvalues

- A way to think about it:
  - Expressions are boxes with a certain content
    - Content has a type and a (real) value
    - Content may be a literal, an object, the result of another expression
  - In order to refer to a box (e.g. to modify it), the box needs to have a name
    - Lvalues: boxes with names
    - Rvalues: boxes without names



# Rvalues

- Have expression scope
  - They die at the end of an expression
- They are typically evaluated for their values
- Cannot be assigned to
  - Reason:
    - They are part of an expression
    - Assigning to them could change them before they have been used

# Lvalues and Rvalues

- Consider the following function

An lvalue (object with name and address)

```
void printSomething( const std::string & str ) {  
    std::cout << str << std::endl;  
}
```

and the following call

```
printSomething( std::string("Hello") );
```

**Special case:** local const reference can be used to prolong the lifetime of temporary value and refers to it until the end of the containing scope (does not incur the cost of a copy-construction!)

An rvalue!!

# Today's learning objectives

- Leading Example: Smart pointers
- Move semantics
- Lvalues and Rvalues
- **Rvalue references**
- Move construction and move assignment
- `std::move`
- Efficiency considerations



# C++11: lvalue and rvalue references

- Lvalue references are just “normal” reference as known pre-C++11
- 2 types:

L-value reference	Can be initialized with	Can modify
Modifiable Lvalues	Yes	Yes
Non-modifiable Lvalues	No	No
R-values	No	No

# C++11: lvalue and rvalue references

- Lvalue references are just “normal” reference as known pre-C++11
- 2 types:

L-value reference to const	Can be initialized with	Can modify
Modifiable Lvalues	Yes	No
Non-modifiable Lvalues	Yes	No
R-values	Yes	No

# C++11: lvalue and rvalue references

- What is an rvalue reference?
  - As the name says: a reference to an r-value
- Initialization:
  - Use double ampersand

```
int x = 5;
int &lref = x;    // l-value reference
                  // initialized with l-value x
int &&rref = 5;    // r-value reference initialized
                  // with r-value 5
```

# Rvalue references

- Again, 2 types

R-value reference	Can be initialized with	Can modify
Modifiable Lvalues	No	No
Non-modifiable Lvalues	No	No
R-values	Yes	Yes

# Rvalue references

- Again, 2 types

R-value reference to const	Can be initialized with	Can modify
Modifiable Lvalues	No	No
Non-modifiable Lvalues	No	No
R-values	Yes	No

# Rvalue references

- Allow us to:
  - Extend the lifespan of the (rvalue) object they are initialized with by the lifespan of the Rvalue reference
  - Modify the Rvalue!

# Rvalue references

- Example:

```
#include <iostream>

class Fraction {
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator) {}

    friend std::ostream& operator<< ( std::ostream& out,
                                      const Fraction &f1 ) {
        out << f1.m_numerator << "/" << f1.m_denominator;
        return out;
    }
};
```

# Rvalue references

- Usage:

```
int main() {  
    Fraction &&rref = Fraction(3, 5); // r-value reference  
                                     // to temporary Fraction  
    std::cout << rref << '\n';  
  
    return 0;  
} // rref (and the temporary Fraction) goes out of scope here
```

Output: 3/5



# Rvalue references

- May hide creation of temporaries!

```
#include <iostream>
```

```
int main() {  
    int &&rref = 5; // because we're initializing an  
                  // r-value reference with a literal,  
                  // a temporary with value 5 is  
                  // created here  
  
    rref = 10;  
    std::cout << rref;  
    return 0;  
}
```

Output: 10

# Rvalue references

- Typically used as function parameters:

```
void fun(const int &lref) { // l-value args select this function
    std::cout << "l-value reference to const\n";
}

void fun(int &&rref) { // r-value arguments will select this function
    std::cout << "r-value reference\n";
}

int main() {
    int x = 5;
    fun(x); // l-value argument calls l-value version of function
    fun(5); // r-value argument calls r-value version of function

    return 0;
}
```

l-value reference to const  
r-value reference

# Rvalue references

- temporaries can be passed as a const lvalue reference
- **However:** rvalue reference is considered a better match!
- Tricky:

```
int &&ref = 5;  
fun(ref);
```

- Has type rvalue reference
- Is an lvalue itself (it has a name)

- Calls the lvalue version!

# Today's learning objectives

- Leading Example: Smart pointers
- Move semantics
- Lvalues and Rvalues
- Rvalue references
- **Move construction and move assignment**
- `std::move`
- Efficiency considerations

# Back to our pointer example

- `Auto_ptr2`:
  - Overriding the default (shallow) copy constructor/assignment operator to implement move semantics
  - Unsafe
    - Example: pass by value to another function
- Let's make `Auto_ptr3`:
  - Simply does deep copy everytime
    - Guaranteed safe

# Auto\_ptr3

```
template<class T>
class Auto_ptr3 {
    T* m_ptr;
public:
    Auto_ptr3(T* ptr = nullptr) : m_ptr(ptr) {}
    ~Auto_ptr3() { delete m_ptr; }

    // Copy constructor: Do deep copy of a.m_ptr to m_ptr
    Auto_ptr3(const Auto_ptr3& a) {
        m_ptr = new T;
        *m_ptr = *a.m_ptr;
    }

    ...
}
```

# Auto\_ptr3 (continued)

...

```
// Copy assignment: Do deep copy of a.m_ptr to m_ptr
```

```
Auto_ptr3& operator=(const Auto_ptr3& a) {
```

```
    // Self-assignment detection
```

```
    if (&a == this)
```

```
        return *this;
```

```
    // Release any resource we're holding
```

```
    delete m_ptr;
```

```
    // Copy the resource
```

```
    m_ptr = new T;
```

```
    *m_ptr = *a.m_ptr;
```

```
    return *this;
```

```
}
```

```
T& operator*() const { return *m_ptr; }
```

```
T* operator->() const { return m_ptr; }
```

```
bool isNull() const { return m_ptr == nullptr; }
```

```
};
```

# Using Auto\_ptr3

Output:  
Resource acquired  
Resource acquired  
Resource destroyed  
Resource acquired  
Resource destroyed  
Resource destroyed

```
class Resource {  
    public:  
        Resource() { std::cout << "Resource acquired\n"; }  
        ~Resource() { std::cout << "Resource destroyed\n"; }  
};
```

```
Auto_ptr3<Resource> generateResource() {  
    Auto_ptr3<Resource> res(new Resource);  
    return res; // return value invokes copy constructor  
}
```

```
int main() {  
    Auto_ptr3<Resource> mainres;  
    mainres = generateResource(); // assign invokes copy assignment  
  
    return 0;  
}
```

Very inefficient!

- 3 Resource allocations just to create an object
- But at least it is safe!



# Move construction/assignment

- Role
  - Move ownership from one object to another
  - Use instead of copying to gain efficiency
- How to define?
  - Somewhat similar to regular copy constructor/assignment operator
  - **But:** Take non-const r-value reference instead of const l-value reference!

# Auto\_ptr4

- Beginning stays unchanged

```
#include <iostream>
```

```
template<class T>
```

```
class Auto_ptr4 {
```

```
    T* m_ptr;
```

```
public:
```

```
    Auto_ptr4(T* ptr = nullptr) : m_ptr(ptr) {}
```

```
    ~Auto_ptr4() {
```

```
        delete m_ptr;
```

```
    }
```

```
...
```

# Auto\_ptr4

- Move constructor:

...

// Copy constructor: Do deep copy of a.m\_ptr to m\_ptr

```
Auto_ptr4(const Auto_ptr4& a) {
```

```
    m_ptr = new T;
```

```
    *m_ptr = *a.m_ptr;
```

```
}
```

Important:

- Remove ownership
- Prevent a from destroying resource
- Prevent a from causing dangling pointers

// Move constructor: Transfer ownership of a.m\_ptr to m\_ptr

```
Auto_ptr4(Auto_ptr4&& a) : m_ptr(a.m_ptr) {
```

```
    a.m_ptr = nullptr;
```

```
}
```

// Copy assignment: Do deep copy of a.m\_ptr to m\_ptr

```
Auto_ptr4& operator=(const Auto_ptr4& a) {
```

```
    ...
```

```
} ...
```

# Auto\_ptr4

- Move assignment operator

...

```
// Move assignment: Transfer ownership of a.m_ptr to m_ptr
Auto_ptr4& operator=(Auto_ptr4&& a) {
    // Self-assignment detection
    if (&a == this)
        return *this;

    // Release any resource we're holding
    delete m_ptr;
```

```
    // Transfer ownership of a.m_ptr to m_ptr
    m_ptr = a.m_ptr;
    a.m_ptr = nullptr;
```

```
    return *this;
```

```
} ...
```

Important:

- Remove ownership
- Prevent **a** from destroying resource
- Prevent **a** from causing dangling pointers

# Auto\_ptr4: Usage

```
...  
Auto_ptr4<Resource> generateResource() {  
    Auto_ptr4<Resource> res(new Resource);  
    return res; // return value invokes move constructor  
}
```

When res goes out of scope, it has no more ownership.  
Return value was move constructed. **Why!?**

```
int main() {  
    Auto_ptr4<Resource> mainres;  
    mainres = generateResource(); // assign invokes move assignment  
  
    return 0;  
}
```

generateResource returns temporary, move assignment is chosen!

Output:  
Resource acquired  
Resource destroyed

# Move construction/assignment

- When?
  - When defined and argument is an r-value (literal, temporary)
  - **Exception: Automatic l-values returned by value!**
    - Special C++ specification
    - Makes sense: automatic l-values will be destroyed at end of function call anyway → just make r-values out of them!
- In most cases, they are not provided by default
  - → If you want them, you need to implement them

# Move construction/assignment

- Key insights:
  - If an l-value is used:
    - Copying is the only reasonable alternative
    - Example: **`a = b;`** should make a (deep) copy, and not alter **`b`**
  - If an r-value is used:
    - R-value is just a temporary about to be destroyed
    - It is reasonable to steal ownership and avoid copying! (which is more efficient)
    - Example: **`a = b + c;`**

# Disable copy constructor/assignment

- In some cases they are undesirable
  - Example:
    - Auto\_ptr4
    - Is move assigned/constructed if returned by another function
    - No need to copy, can be passed by value

```
...  
// Copy constructor -- no copying allowed!  
Auto_ptr4(const Auto_ptr4& a) = delete;  
  
// Copy assignment -- no copying allowed!  
Auto_ptr4& operator=(const Auto_ptr4& a) = delete;  
...
```



# Today's learning objectives

- Leading Example: Smart pointers
- Move semantics
- Lvalues and Rvalues
- Rvalue references
- Move construction and move assignment
- **std::move**
- Efficiency considerations

# std::move

- Extension of move semantics to l-values!
- Example:

```
int main() {  
    std::string x{ "abc" };  
    std::string y{ "de" };  
  
    std::cout << "x: " << x << '\n';  
    std::cout << "y: " << y << '\n';  
  
    swap(x, y);  
  
    std::cout << "x: " << x << '\n';  
    std::cout << "y: " << y << '\n';  
  
    return 0;  
}
```

Output:  
x: abc  
y: de  
x: de  
y: abc

# std::move

- Extension of move semantics to l-values!
- Example:

```
#include <iostream>
#include <string>

template<class T>
void swap(T& a, T& b) {
    T tmp { a }; // invokes copy constructor
    a = b;        // invokes copy assignment
    b = tmp;      // invokes copy assignment
}
```

Involves 3 copies!

# std::move

- Doing 3 copies is unnecessary
- Could be done with 3 moves instead!
- Solution: Cast the l-values to r-values
  - Done using **std::move**

```
#include <iostream>
#include <string>

template<class T>
void swap(T& a, T& b) {
    T tmp { std::move(a) }; // invokes move constructor
    a = std::move(b);       // invokes move assignment
    b = std::move(tmp);     // invokes move assignment
}
```

# **std::move**

- However:
  - What happens to the original l-value!?
  - Let's do another Example/test

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>
```

```
int main() {
    std::vector<std::string> v;
    std::string str = "Knock";
```

```
    std::cout << "Copying str\n";
```

```
    v.push_back(str); // calls l-value version of push_back
                     // which copies str to array
```

```
    std::cout << "str: " << str << '\n';
```

```
    std::cout << "vector: " << v[0] << '\n';
```

```
    std::cout << "\nMoving str\n";
```

```
    v.push_back(std::move(str)); // calls r-value version
                                // moving str into array
```

```
    std::cout << "str: " << str << '\n';
```

```
    std::cout << "vector:" << v[0] << ' ' << v[1] << '\n';
```

```
    return 0;
```

```
}
```

Output:  
Copying str  
str: Knock  
vector: Knock

Moving str  
str:  
vector: Knock Knock

# `std::move`

- What has happened to original l-value?
- It has changed!
  - Reset to null string
- This is good:
  - Objects that are being stolen from need to be left in a defined “null state”
  - They are not a temporary after all, and may be used again later!

# Today's learning objectives

- Leading Example: Smart pointers
- Move semantics
- Lvalues and Rvalues
- Rvalue references
- Move construction and move assignment
- `std::move`
- Efficiency considerations




# Efficiency considerations

- Consider the following example:

```
class MyInteger {  
public:  
    MyInteger( int value ) : _value(value) {};  
    virtual ~MyInteger() {};  
  
    MyInteger operator+( const MyInteger & op ) const {  
        return MyInteger( _value + op._value );  
    }  
  
private:  
    int _value;  
};  
  
int main() {  
    MyInteger val1(2);  
    MyInteger val2(3);  
    MyInteger sum = val1 + val2;  
    return 0;  
}
```

How many copies are  
being made?



# Efficiency considerations

- Consider the following example:

```
class MyInteger {  
public:  
    MyInteger( int value ) : _value(value) {};  
    virtual ~MyInteger() {};
```

Return by value: 1 copy

```
    ← MyInteger operator+( const MyInteger & op ) const {  
        return MyInteger( _value + op._value );  
    }
```

Pass by value: 2 copies

```
private:  
    int _value;  
};
```

```
int main() {  
    MyInteger val1(2);  
    MyInteger val2(3);  
    MyInteger sum = val1 + val2; ←  
    return 0;  
}
```

Copy-construction: 1 copy

How many copies are made in worst case?

# Efficiency considerations

- Why is this just a “worst-case”?
  - In reality, there is hardly any copy being done here!
- What is copy elision?
  - Copy elision happens if the compiler applies some optimization technique to bypass a copying process!

# Efficiency considerations

- Copy elision can and most likely will be applied in several occasions in our example:
  - If passing a temporary by value (an rvalue), why copy it and not immediately use that temporary inside the function? It is up for destruction after the function call anyway!
  - If returning a temporary or even a locally initialized lvalue from a function, why make a copy? Why not prevent destruction and immediately use that as the rvalue returned by the function?
    - This is called RVO (Return value optimization)
    - Compiler reserves space for that rvalue before function call

# Efficiency considerations

- Copy elision can and most likely will be applied in several occasions in our example:
  - If initializing an object using the copy constructor and from a temporary that is up for destruction, the compiler may choose automatically to install move semantics

# Why bother about move constructors etc.?

- RVO and copy elision will be done by the compiler automatically and only when it is sure to not alter the behavior of the code.
- With move constructors and move assignment operators, this behavior can be explicitly controlled.