Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

uOttawa.ca

This Lecture

No more Memory leaks

- **Smart pointers**
  - Understanding smart pointers, Ch. 12
  - C++11 Smart pointer library types
    - `shared_ptr`, Ch.12.1, 12.1.3
    - `unique_ptr`, Ch. 12.1.5 (similar to auto_ptr in C++98)
    - `weak_ptr`, Ch. 12.1.6
- **Move vs. Copy**
  - rvalue references, Ch. 13.6.1
  - move constructor and assignment operator, Ch. 13.6.2
  - rvalue reference and member functions

Understanding Smart Pointers

- **Idea: Encapsulate a pointer inside a class**
  - Overload the dereference operator *ptr
  - Overload the member access through pointer ptr->
- **Ownership management**
  - Deep copy – familiar from pointer class attributes
  - Copy on write
  - Reference counting – example
  - Reference linking
  - Destructive copy – example

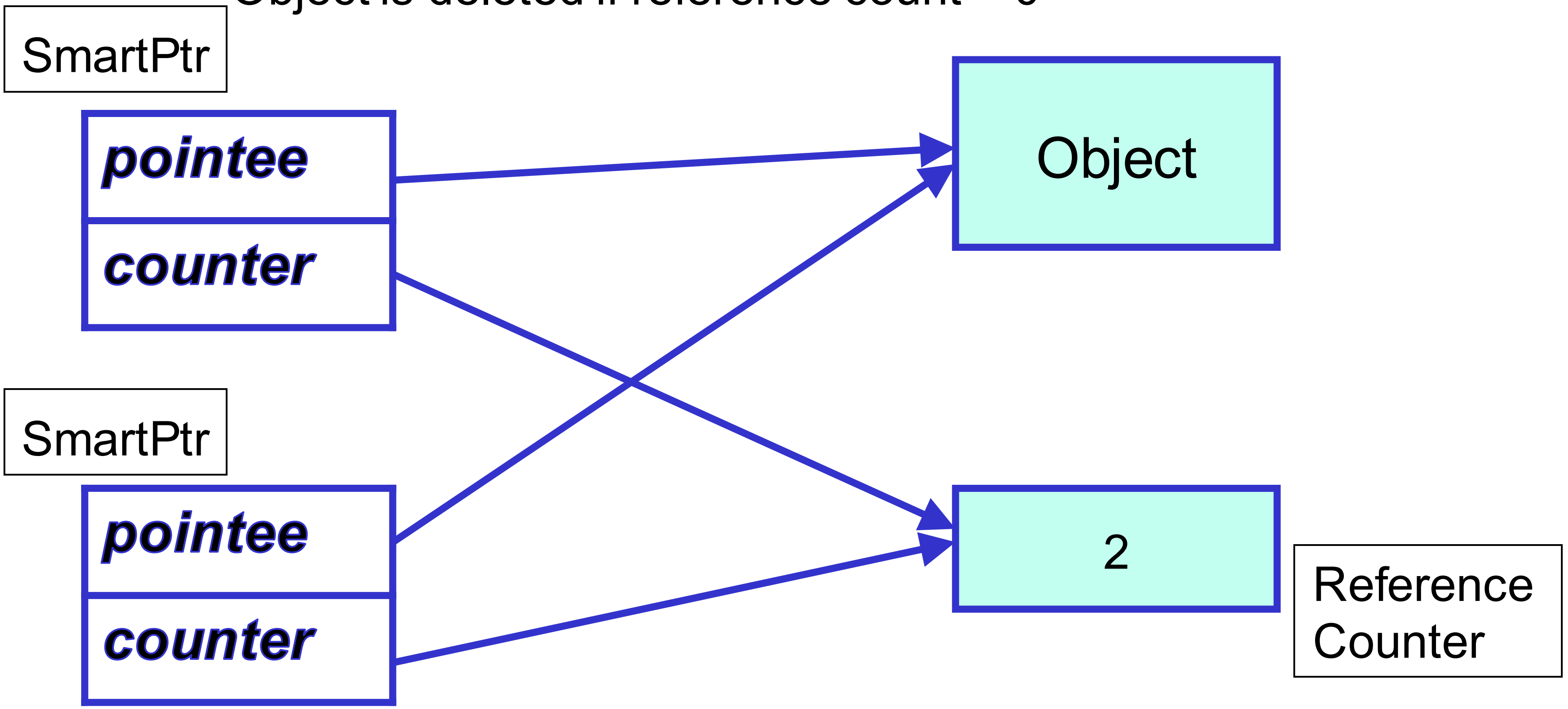
Basic Smart Pointer Layout

- Basic methods and operators
  - no ownership management yet

```
template <class T> class SmartPtr {
  T* d_pointee; // The object pointed to
public:
  // constructor from native pointer
  explicit SmartPtr( T* _pointee ) : d_pointee(_pointee)) {}
  // copy constructor from other smart pointer
  explicit SmartPtr(SmartPtr& _src);
   // delete this smart pointer
  ~SmartPtr();
  // assign a smart ptr to this smart ptr
  SmartPtr<T>& operator=(const SmartPtr<T>& _src);
  T& operator*(); // get the object
  T* operator->(); // pointer to be used in -> operator
…};
```

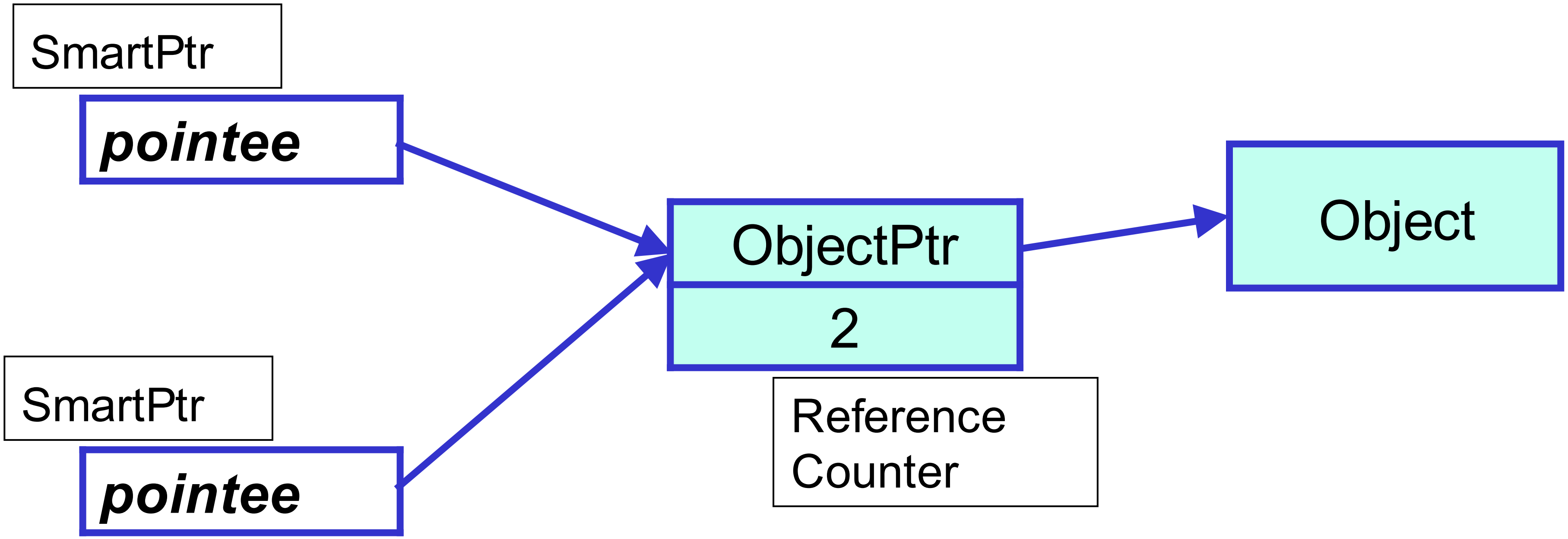
Reference Counting Concept

- **Dynamic object is paired with a counter**
  - Object is deleted if reference count = 0



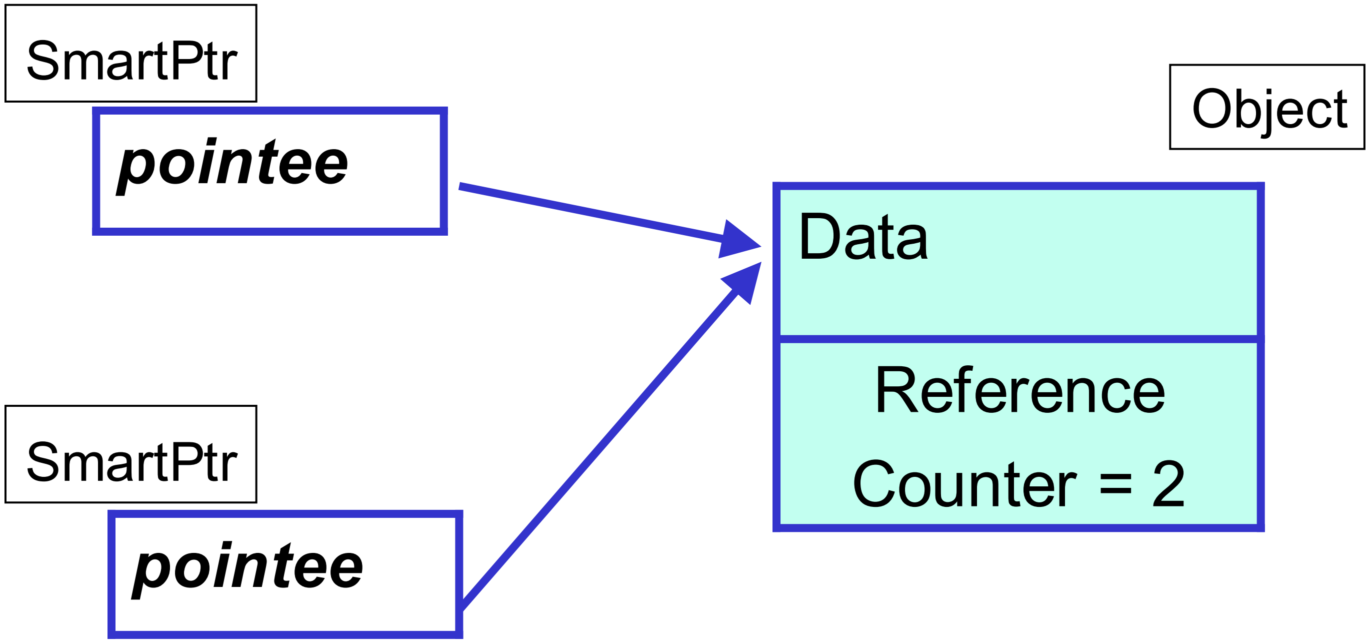
Alternative Reference Counting

- Reduced space by extra level of indirection
- Increased overhead for object access



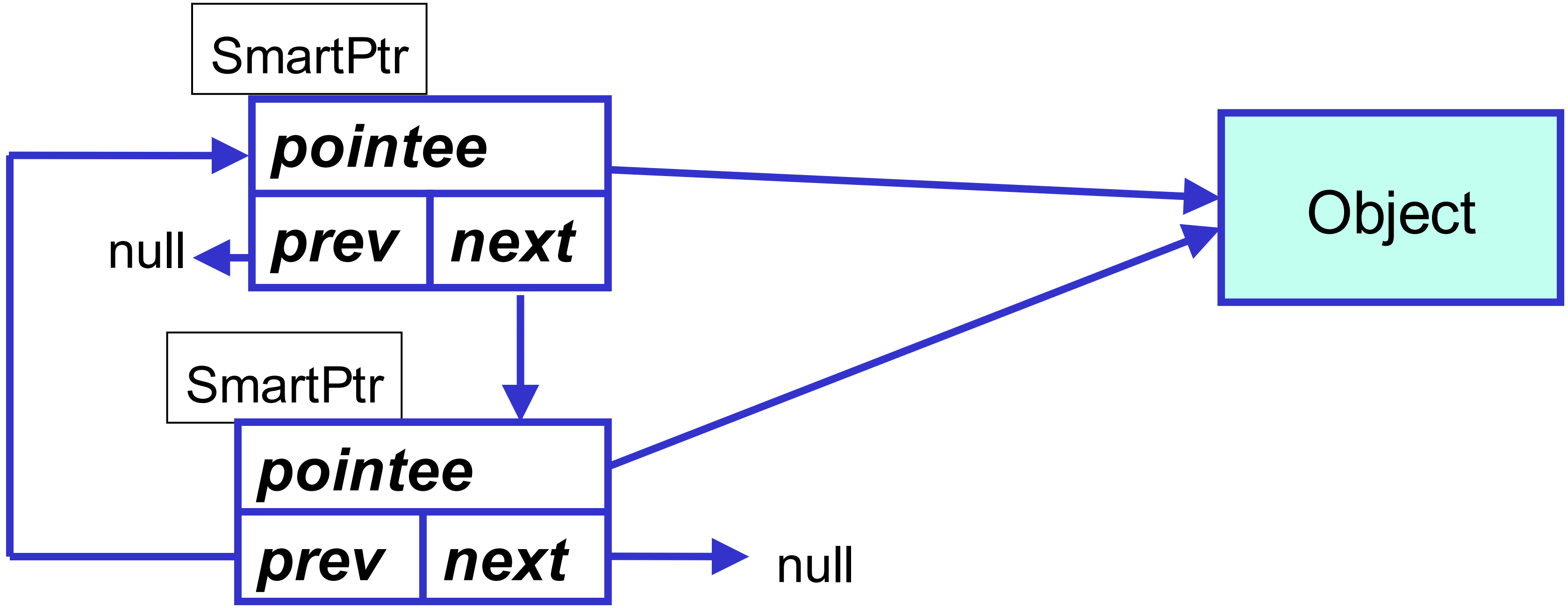
Intrusive Reference Counting

- Most effective if reference counter is part of object
- Objects must be designed for reference counting



Reference Linking

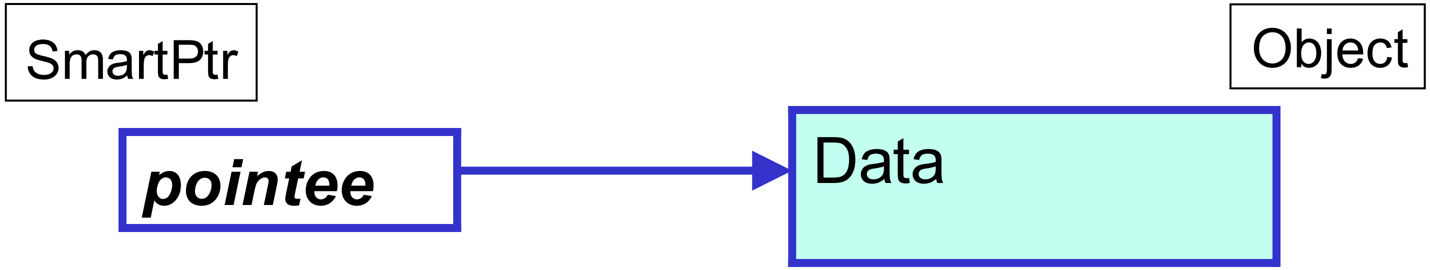
- **Ideas:**
  - Enough to know when the last object is dereferenced
  - Link all pointers into a doubly-linked list



uOttawa

Destructive Copy

- **Ideas:**
  - There is always only one valid ptr to an Object
  - Copying destroys the original source pointer
  - Pass by value acts as a sink
    - Implemented in std::auto_ptr in C++98 but deprecated in C++11
    - Instead use std::unique_ptr in C++11



Reference Counting Example

- Definition of a SmartPtr inspired by A. Alexandrescu, Modern C++ Design, Chpt. 7
- Implement reference counting with reference counter object
- Define comparison operators with raw pointers
- Simple design by not allowing null pointers

Reference Counting

```
template <class T> class SmartPtr {
  class RefCounter { // Helper class for reference counter
    unsigned int d_pCount;
  public:
    RefCounter() : d_pCount(1){};
    void clone() { ++d_pCount; return; }
    bool release() {
      if (!--d_pCount) return true;
      return false; } };
  T* d_pointee; // The object pointed to
  RefCounter* d_counter;
public:
  // constructor from native pointer - do not allow null
  explicit SmartPtr( T* _pointee )
    : d_pointee(_pointee), d_counter(new RefCounter()) {
    if (d_pointee == 0) { delete d_counter;
      throw std::runtime_error("Smart pointer = null");
    }
  }
```

Copying and Destruction

```
// copy constructor from other smart pointer
 SmartPtr(SmartPtr& _src)
 // share the object and the reference counter
   : d_pointee(_src.d_pointee), d_counter(_src.d_counter) {
   d_counter->clone(); // increase the counter
 }
 // delete this smart pointer
 ~SmartPtr() {
   // decrease ref count and check if last pointer to object
   if ( d_counter->release() ) {
     delete d_pointee; // delete object
     delete d_counter; // delete counter object
   }
 }
```

Smart Pointers in STL with C++11

- Different management strategies
  - `shared_ptr` treats the memory as shared ownership between the shared_ptr
  - `unique_ptr` is a bit like destructive copy and the old `std::auto_ptr`, assumes a single valid ptr
  - `weak_ptr` registers a pointer with memory but must be locked before use
- All smart pointer are defined `<memory>`

Operations with Smart Pointers in STL

- **Global operators: comparison and insertion**

```
template < class T, class U >
bool operator==( const shared_ptr<T> &lhs,
                 const shared_ptr<U> &rhs );
// Similar: operator!= operator< operator<= operator>
//          operator>=
template <class T, class U, class V>
basic_ostream<U,V>& operator<<(basic_ostream<U,V>& os,
                               const shared_ptr<T>& ptr );
```

- **Dereferencing**

```
// Derefernce
T& operator*() const;
// Access through pointer to member
T* operator->() const;
```

C++11 Aside: `nullptr`

- **Prior to C++11**
  - Null pointers were commonly written as `int *ptr = 0;`
  - Sometimes a C macro was adapted from <cstdlib> `int *ptr = NULL;`
  - With C++11 we can use use `int *ptr = nullptr;`
    - a new String literal
    - shields from size issues of the pointer type; special type convertible to any pointer type

Using `std::shared_ptr`

- C++11 implements sharing of a (reference counted) resource
- Memory is automatically deleted when last shared reference goes out of scape
- Shared pointer should be created by utility routine

```
// Constructor of resource and sharing overhead
std::shared_ptr<double> sptr = std::make_shared<double>(1.0);
// Share resource with other pointer - cctor
std::shared_ptr<double> sptr2 = std::shared_ptr<double>(sptr);
// Share resource with other pointer - assignment
std::shared_ptr<double> sptr3;
sptr3 = sptr; // Now 3 references
```

C++11 Using `std::unique_ptr`

- **Important:**
  - No copy for unique_ptr
  - No assignment for unique_ptr

```
#include <memory>
using std::unique_ptr;

unique_ptr<A> aPtr(new A(3));
aPtr->getValue(); // calling a function on A
cout << *aPtr << endl; // derefencing to get A

unique_ptr<A> aPtr2; // aPtr2 is a nullptr
aPtr2 = aPtr; // illegal assignment with unique_ptr
unique_ptr<A> aPtr3( aPtr ); // illegal copy with unique_ptr
// if an unique_ptr goes out of scopes it deletes the managed
object
```

C++11 `std::unique_ptr` object ownership

- **Two functionalities to manage ownership**
  - `reset()` and `release()`
  - also `get()` Use with caution! Underlying low-level pointer!

```
#include <memory>
using std::unique_ptr;

unique_ptr<A> aPtr(new A(3));
unique_ptr<A> aPtr2; // aPtr2 is a nullptr
aPtr2.reset(aPtr.release()); // transfer of ownership
aPtr2.reset(); // deletes object A and make aPtr2 a nullptr
aPtr2.reset(new A(4)); // resets aPtr2 to a new object A;
                // deletes any old object if it still exists
aPtr2.release(); // release the object A pointed to be aPtr2
```

Weak Pointers

- **Weak pointers can be initialized with shared pointers and once locked become a shared_ptr**

```
std::weak_ptr<int> wPtr;
{
  std::shared_ptr<double> sPtr=std::make_shared<double>(2.0);
  wPtr = sPtr; // Make weak ptr to same resource
  // Try to get a lock - on success lwPtr is a shared_ptr
  if ( auto lwPtr = wPtr.lock())
      cout << lwPtr << " : " *lwPtr;
}
// Try again
if ( auto lwPtr = wPtr.lock()) cout << lwPtr << " : " *lwPtr;
```

lvalue References

- **lvalue references**
  - We can name a reference to lvalue
  - We can also name a reference to a const rvalue
  - We can not have a lvalue reference to a temporary or result of a computation
    - rvalues are just temporary

```
int i=1;
int& a = i;
```

```
int i=1;
const int& b = i+5;
```

```
int foo();
int i = 2;
int& a=foo(); // Illegal
int& b = 5+3*i; // Illegal
```

rvalue References

- **rvalue references**
  - Using the move mechanism temporary rvalues can be turned in a rvalue reference
  - Replaces a copy with moving resources

```
int foo();
int i = 2;
int&& a = foo(); // Fine
int&& b = 5+3*i; // Fine
int&& c = 42; // Also ok.
```

Why move?

- **Optimization**
  - E.g., adding to a std::vector involves copying an object into the vector storage. But:
    - Usually the object is temporary and just used to pass it to the vector via push_back
    - Whenever the vector grows, we are guaranteed that objects are stored in contiguous memory:
      - Allocate k-times the current memory
      - Copy all the elements over
      - Delete the old elements
    - Moving would save the copies and with it the call to new

Move Constructor

- **Similar to copy constructor**
  - Also synthesized by the compiler
- **BUT**
  - Moves all the resources from the copied source object
  - Does not need to allocate new resources and can be made noexcept

```
class A {
  A( A&& _oA ) noexcept;
};
```

Move Assignment Operator

- **Similar to regular assignment operator**
  – Also synthesized by the compiler
- **BUT**
  – Moves all the resources from the assigned source object
  – Does not need to allocate new resources and can be made noexcept

```
class A {
  A& operator=(A&& _oA) noexcept;
};
```

Synthesized Move

- **Synthesized move constructor and move assignment by the compiler only if**
  - No definition of copy constructor, assignment operator or destructor
  - All class variables must be move constructible
    - Built-in types are ok
    - Own types must have a defined or synthesized move constructor

Defining your own Move

- **Whenever we need to apply the rule of three:**
  - Dynamically allocated resources (not managed with smart pointers) require us to define
    - copy constructor, assignment operator and destructor
  - We may want to upgrade to the rule of 5
    - Rule of 3 plus move constructor and move assignment operator
  - Must make sense:
    - moved from source object must be left in a destructible state
    - must be able to assign to moved from source object

A Worked Example

- **Class that holds its own dynamic array**

```
class ThumbNail {
  unsigned int d_size;
  unsigned char* d_pattern;
… };
```

  - The use of a pointer to a dynamically allocated array requires us to manage the copy control of objects of the class

- **Rule of 3 applies**

```
class ThumbNail {  …
public:
  ThumbNail(const ThumbNail& _otn);
  ~ThumbNail();
  ThumbNail& operator=(const ThumbNail& _otn);
};
```

Upgrade to the rule of 5

- Rule of three is sufficient to
  - prevent memory leaks, and
  - ensure that we can use our own type with the containers of the std library
- **But we can gain efficiency**

```
class ThumbNail {  …
public:
  ThumbNail(const ThumbNail& _otn);
  ~ThumbNail();
  ThumbNail& operator=(const ThumbNail& _otn);
 // Move cctor
  ThumbNail( ThumbNail&& _otn ) noexcept;
  // Move assignment
  ThumbNail& operator=(ThumbNail&& _otn) noexcept;
};
```

Example

- **The following code**

```
std::vector<ThumbNail> vec;
unsigned char res[] = "x1y2z1a3";
for ( int i=0; i<3; ++i ) {
  vec.push_back( ThumbNail(res,i*2+2));
}
```

  - calls the copy constructor for ThumbNail six times!
- **With the move constructor with noexcept, six calls to the move constructor are used (at least if the compiler cooperates)!**

rvalue reference functions

- We can define other class functions that take rvalues references
- Function overloading takes the lvalue/rvalue property into account
  - Requires that all overloaded functions define the reference qualifier

```
class ThumbNail {  …
  // Will steal resources and just update local variables
  // and return this
  ThumbNail& scramble() &&;
  // Will not change this and make a copy before changing
  ThumbNail scramble() const &;
};
```

Example rvalue reference functions

- **Example definition and call**

```
ThumbNail& ThumbNail::scramble() && {
  if (d_pattern)
    std::random_shuffle(d_pattern,d_pattern+d_size);
  return *this;
}
ThumbNail ThumbNail::scramble() const & {
  ThumbNail res(*this);
  if (res.d_pattern)
    std::random_shuffle(res.d_pattern,
                        res.d_pattern+res.d_size);
  return res;
}
…
tn = tn2.scramble();
tn = foo().scramble();
```

variable – uses lvalue

temporary – uses rvalue reference