Rvalue References & Move Semantics

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Motivation

Recall

- lvalues vs. rvalues
  - lvalues have identity
  - rvalues are temporary or literals
  - more detail: `http://en.cppreference.com/w/cpp/language/value_category`
- (lvalue) references
  - aliases
  - pass-by-reference function arguments

```
void foo( int & ) { std::cout << "int &\n"; }
void foo( const int & ) { std::cout << "const int &\n"; }
int x = 0;
int fun() { return x; }
foo( x );
foo( fun() );
foo( 7 );
```

Motivation

Lvalue references

- catch lvalues
- catch also rvalues when `const`
  - extend the lifetime of temporaries
  - cannot be modified

Motivation

Lvalue references

- catch lvalues
- catch also rvalues when `const`
  - extend the lifetime of temporaries
  - cannot be modified

Why modify rvalues (temporaries)?

- reuse the internals of a temporary object
- avoid (expensive or impossible) copies
  - `std::vector`
  - arithmetic with large objects
  - smart pointers (next seminar)

Rvalue References

Syntax: `type && var`

```
int foo();
int x = 3;

int && r1 = 5;
int && r2 = foo();
int && r3 = x; // error: cannot bind lvalue to int &&
```

- rvalue references only catch rvalues (temporaries)

Rvalue References

Syntax: `type && var`

```
int foo();
int x = 3;

int && r1 = 5;
int && r2 = foo();
int && r3 = x; // error: cannot bind lvalue to int &&
```

- rvalue references only catch rvalues (temporaries)
- lifetime of temporaries is extended by rvalue references

```
struct X { /* ... */ };
X createX();

X && r = createX();
```

Using Rvalue References

Rvalue references enable move semantics

```
class IntArray {
    int * data;
public:
    IntArray() : data( new int[ 32 ] ) {}
    IntArray( const IntArray & o ) : data( new int[ 32 ] )
        std::copy( o.data, o.data + 32, data );
    }
    IntArray( IntArray && o ) : data( o.data ) {
        o.data = nullptr; // data have been stolen
    }
    ~IntArray() { delete [] data; }

    IntArray &operator=( IntArray o ) { swap( o ); }

    void swap( IntArray & o ) { std::swap( data, o.data );
};
```

Interlude

Quiz: How many methods does this class have?

```
class Empty {};
```

Interlude

Quiz: How many methods does this class have?

```
class Empty {};
```

- in C++03, the answer is **four**:

```
Empty();
Empty( const Empty & );
Empty &operator=( const Empty & );
~Empty();
```

Interlude

Quiz: How many methods does this class have?

```
class Empty {};
```

- in C++03, the answer is **four**:

```
Empty();
Empty( const Empty & );
Empty &operator=( const Empty & );
~Empty();
```

- in C++11, the answer is **six**:

```
Empty( Empty && );
Empty &operator=( Empty && );
```

Rule of Three/Zero Revisited

Remember Rule of Three and Rule of Zero?

- copy constructor, copy assignment operator, destructor
- either implement all three or none of them

Rule of Five

- add move constructor, move assignment operator
- (if move semantics makes sense for your class)

Rule of Four and a Half

- only one assignment operator using copy-and-swap

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

Casting to Rvalue Reference

- sometimes we want to allow move from lvalues

```
void registerNewThing( int id ) {
    Thing thing( id );
    // some code that deals with thing
    thingStorage.push_back( thing ); // copy
    // but we don't need thing anymore
}
```

Casting to Rvalue Reference

- sometimes we want to allow move from lvalues

```
void registerNewThing( int id ) {
    Thing thing( id );
    // some code that deals with thing
    thingStorage.push_back( thing ); // copy
    // but we don't need thing anymore
}
```

- casting to rvalue reference: `std::move`

```
thingStorage.push_back( std::move( thing ) ); // move
```

Note: `std::move` does not move anything, it is just a cast!

Forwarding Rvalue References

```
template< typename T >
class Stack {
    std::vector< T > impl;
public:
    void push( T && t ) {
        impl.push_back( t ); // what happens here?
    }
// ...
};
```

Forwarding Rvalue References

```
template< typename T >
class Stack {
    std::vector< T > impl;
public:
    void push( T && t ) {
        impl.push_back( t ); // what happens here?
    }
// ...
};
```

- a rvalue reference variable is a lvalue; why?

Forwarding Rvalue References

```
template< typename T >
class Stack {
    std::vector< T > impl;
public:
    void push( T && t ) {
        impl.push_back( t ); // what happens here?
    }
// ...
};
```

- a rvalue reference variable is a lvalue; why?
- solution: use `std::move`

Move Semantics & Exceptions

`std::vector` uses move instead of copy when extending the vector.
What if the move constructor throws an exception?

Move Semantics & Exceptions

`std::vector` uses move instead of copy when extending the vector.
What if the move constructor throws an exception?

- cannot return to consistent state
  (`std::vector` promises strong exception guarantee)
- note: this problem does not arise with copy constructors

Move Semantics & Exceptions

`std::vector` uses move instead of copy when extending the vector.
What if the move constructor throws an exception?

- cannot return to consistent state
  (`std::vector` promises strong exception guarantee)
- note: this problem does not arise with copy constructors

Solution:

- `std::vector` only moves if the move constructor is `noexcept`
- using `std::move_if_noexcept`

Recommendation: make your move constructors `noexcept` if possible.

Universal References

Combining references

```
using LvRef = int &;
using RvRef = int &&;

using T1 = LvRef &;     // int &
using T2 = LvRef &&;    // int &
using T3 = RvRef &;     // int &
using T4 = RvRef &&;    // int &&
```

Universal References

```
template< typename T >
void foo( T && t ) {
    // what type is t here?
}
```

Universal References

```
template< typename T >
void foo( T && t ) {
    // what type is t here?
}
```

Universal reference

- `foo` accepts both lvalues and rvalues
- if `foo` is given a lvalue, `T` is `X &` and `T &&` is also `X &`
- if `foo` is given a rvalue, `T` is `X` and `T &&` is `X &&`

Universal References

```
template< typename T >
void foo( T && t ) {
    // what type is t here?
}
```

Universal reference

- `foo` accepts both lvalues and rvalues
- if `foo` is given a lvalue, `T` is `X &` and `T &&` is also `X &`
- if `foo` is given a rvalue, `T` is `X` and `T &&` is `X &&`

Consequences?

```
template< typename T >
void foo( T && t ) { cout << "T&&\n"; }
template< typename T >
void foo( const T & t ) { cout << "const T&\n"; }
```

`http://ericniebler.com/2013/08/07/universal-references-and-the-copy-constructo/`

Perfect Forwarding

Problem:

```
class Thing { /* ... */ };
class ThingFactory {
public:
    template< typename T >
    Thing createThing( T && t ) {
        return Thing( t );
    }
};
```

- we want: move if `t` is temporary, copy otherwise

Perfect Forwarding

Problem:

```
class Thing { /* ... */ };
class ThingFactory {
public:
    template< typename T >
    Thing createThing( T && t ) {
        return Thing( t );
    }
};
```

- we want: move if `t` is temporary, copy otherwise

Solution: `std::forward< T >( t )`

- more about perfect forwarding when we learn about variadic templates

Recommendation:

- `move` rvalue references, `forward` universal references

Perfect Forwarding

Solution to problem from slide 12

```
template< typename T >
void foo_impl( T && t, std::false_type ) {
    cout << "T&&\n";
}
template< typename T >
void foo_impl( const T & t, std::true_type ) {
    cout << "const T&\n";
}
template< typename T >
void foo( T && t ) {
    foo_impl( std::forward< T >( T ),
              std::is_lvalue_reference< T >() );
}
```

Other possible solution: `std::remove_reference_t`
(see study materials: `04_universalReferences.cpp`)

Copy Elision

Compilers may omit copies (or moves) in certain circumstances:

- `return` local object (Named Return Value Optimization)
- nameless temporary copied or moved to an object (RVO)
- in both cases, needs to be the same type

Returning function argument taken by value

- copy elision is not done
- but the object is automatically moved

Recommendation:

- do not write `return std::move( x )` if `x` is local or by-value argument
- (sometimes `return std::move( x )` may make sense, when?)

More information on copy elision: `http://en.cppreference.com/w/cpp/language/copy_elision`

Exercise

Exercise no. 1

- extend your `vector` implementation from the first seminar (or use the sample solution) to be move-aware
  - move constructor
  - move assignment operator (if not using copy-and-swap)
  - push_back with rvalue reference argument
  - extending the vector should move instead of copy if possible

Exercise no. 2

- Create a `Reporter` class that reports when its objects are being copied/moved/assigned/etc.
  - What happens if you `push_back` `Reporter` objects into `std::vector`?
  - What happens if you push them into your `vector`?
  - What happens if you create a new class that has a `Reporter` member variable and only default methods?
    Try copying/moving the objects of the class.
  - Try copy elision (NRVO, RVO).