Rvalue References, Move Semantics, Universal References

PV264 Advanced Programming in C++

Nikola Beneš   Jan Mrázek   Vladimír Štill

Faculty of Informatics, Masaryk University

autumn 2020

Motivation

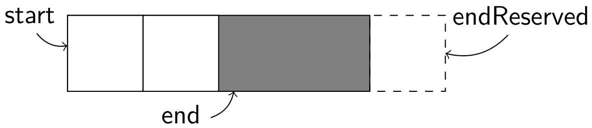
How does `std::vector` work?

Motivation

How does `std::vector` work?

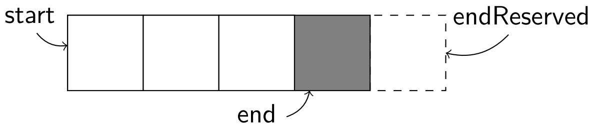
Motivation

How does `std::vector` work?



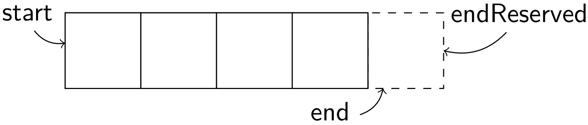
Motivation

How does `std::vector` work?



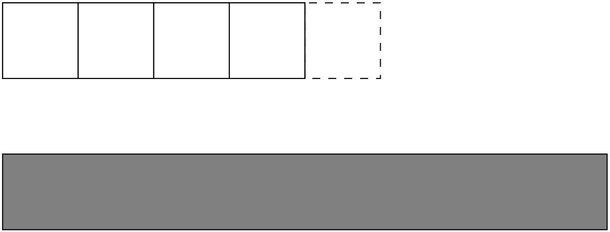
Motivation

How does `std::vector` work?



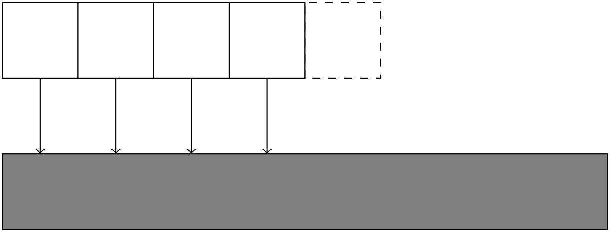
Motivation

How does `std::vector` work?



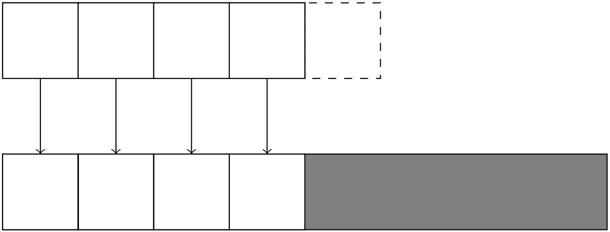
Motivation

How does `std::vector` work?



Motivation

How does `std::vector` work?



- pre-C++11: reallocation means copies
  - copies can be expensive (or even impossible)
  - *however*, we know that the copied elements will be destroyed *immediately afterwards*
  - **move semantics**: take advantage of this observation

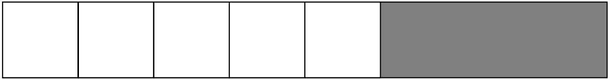
Motivation

How does `std::vector` work?

- pre-C++11: reallocation means copies
  - copies can be expensive (or even impossible)
  - *however*, we know that the copied elements will be destroyed *immediately afterwards*
  - **move semantics**: take advantage of this observation

Motivation

How does `std::vector` work?



- pre-C++11: reallocation means copies
  - copies can be expensive (or even impossible)
  - *however*, we know that the copied elements will be destroyed *immediately afterwards*
  - **move semantics**: take advantage of this observation

Motivation

Lvalues vs. rvalues (*very* simplified)

- lvalues have identity (name), are long-lived
- rvalues are temporaries or literals
- more details on cppreference

Motivation

Lvalues vs. rvalues (*very* simplified)

- lvalues have identity (name), are long-lived
- rvalues are temporaries or literals
- more details on cppreference

Lvalue references `&`

- bind to lvalues
- can sometimes also bind to rvalues, when?

Motivation

Lvalues vs. rvalues (*very* simplified)

- lvalues have identity (name), are long-lived
- rvalues are temporaries or literals
- more details on cppreference

Lvalue references &

- bind to lvalues
- can sometimes also bind to rvalues, when?
  - `const` (lvalue) references
  - extend the lifetime of temporaries

```
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

We want to have rvalue references, i.e. references that bind to temporaries.
Why is it useful?

Motivation

We want to have rvalue references, i.e. references that bind to temporaries. Why is it useful?

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

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** bind to 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** bind to rvalues (temporaries)
- lifetime of temporaries is extended by rvalue references

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

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

Move Semantics

- main reason for rvalue references
- idea: internals of temporary/moved object can be reused
- also transfer of ownership
- you have already seen: transfer of ownership of `unique_ptr` using `std::move`

Move Semantics

- main reason for rvalue references
- idea: internals of temporary/moved object can be reused
- also transfer of ownership
- you have already seen: transfer of ownership of `unique_ptr` using `std::move`
- move construction: like copy construction, but the moved-from object need not remain useful
    - can “steal” data from moved-from object, need not copy them
    - moved-from object has to remain in a valid state
    - what can be done with this object?
- move assignment: similar, but for assignment operator
- `std::move` is cast-to-rvalue

Move Constructor & Move Assignment Operator

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

    Array& operator=(const Array& o) {
        std::copy(o._data, o._data + 32, _data);
        return *this;
    }
    Array& operator=(Array&& o) {
        _data = o._data;
        o._data = nullptr;
        return *this;
    }
};
```

Move Constructor & Move Assignment Operator

Which constructor or assignment operator will be called?

```
Array foo(); // this is a function declaration

Array x(foo());         // Array x = foo();
Array y(x);             // Array y = x;
Array z(std::move(x));  // Array z = std::move(x);

x = foo();
y = x;
z = std::move(x);
```

Quiz

How many methods does this struct have?

```
struct Empty {};
```

Quiz

How many methods does this struct have?

```
struct Empty {};
```

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

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

Quiz

How many methods does this struct have?

```
struct 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 Five

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 and move assignment operator
- (only if move semantics is beneficial for your class)

Rule of Four and a Half

- only one assignment operator using the copy-and-swap idiom

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

Implicitly Defined Constructors and Operators

copy constructor `Object(const Object&)`

- calls copy constructors of attributes and bases
  (in the initialization order)

copy assignment operator `Object& operator=(const Object&)`

- calls copy assignment operators of attributes and bases
  (in the initialization order)

move constructor `Object(Object&&)`

- calls move constructors of attributes and bases
  (in the initialization order)

move assignment operator `Object& operator=(Object&&)`

- calls move assignment operators of attributes and bases
  (in the initialization order)

Hiding of Constructors and Operators (simplified)

default constructor `Object()`

- not implicitly defined when
  - other constructors are present
  - class contains something not default constructible

copy constructor `Object(const Object&)`

- not implicitly defined when
  - class has user-defined move constructor/operator=
  - class contains something not copyable

copy assignment operator `Object& operator=(const Object&)`

- not implicitly defined when
  - class has user-defined move constructor/operator=
  - class contains something not copy-assignable

Hiding of Constructors and Operators (simplified)

move constructor `Object(Object&&)`

- not implicitly defined when
    - class has user-defined move operator=
    - class has user-defined copy constructor/operator=
    - class has user-defined destructor
    - class contains something not movable

move assignment operator `Object& operator=(Object&&)`

- not implicitly defined when
    - class has user-defined move constructor
    - class has user-defined copy constructor/operator=
    - class has user-defined destructor
    - class contains something not move-assignable

Copy-and-Swap Idiom

```
struct Array {
    /* ...as before, but... */
    Array& operator=(Array o) { swap(o); return *this; }
    void swap(Array& o) {
        using std::swap;
        swap(_data, o._data);
    }
};
```

How does this work?

```
Array foo();

Array x, y;

x = foo();
x = y;
```

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
    storage.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
    storage.push_back(thing); // copy
    // but we don't need thing anymore
}
```

Casting to rvalue reference: `std::move`

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

Note: `std::move` does not really move anything, it is just a cast that *enables* move.

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?
    }
};
```

- an rvalue reference variable is an 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?
    }
};
```

- an rvalue reference variable is an lvalue; why?
    - it has an identity, it has a name
    - it can be used several times

Forwarding Rvalue References

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

- an rvalue reference variable is an lvalue; why?
    - it has an identity, it has a name
    - it can be used several times
- solution: use `std::move` here

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.

Initialization of Member Variables

```
class Person {
    std::string name;
public:
    // pre-C++11
    Person(const std::string& n) : name(n) {}
    // post-C++11
    Person(std::string n) : name(std::move(n)) {}
};
```

- advantage of the second approach?

Initialization of Member Variables

```
class Person {
    std::string name;
public:
    // pre-C++11
    Person(const std::string& n) : name(n) {}
    // post-C++11
    Person(std::string n) : name(std::move(n)) {}
};
```

- advantage of the second approach?
    - no copies if initialized with a temporary/moved value
- any disadvantages?

Initialization of Member Variables

```
class Person {
    std::string name;
public:
    // pre-C++11
    Person(const std::string& n) : name(n) {}
    // post-C++11
    Person(std::string n) : name(std::move(n)) {}
};
```

- advantage of the second approach?
    - no copies if initialized with a temporary/moved value
- any disadvantages?
    - if initialized with an lvalue, does copy + move instead of just copy
    - however, moves are typically very cheap
- prefer the new style

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 reference

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

Universal reference

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

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

Universal reference

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

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

Watch out! This is **not** a universal reference:

```
template <typename T>
struct Bar {
    void foo(T&& t); // What is the argument type?
};
```

Perfect Forwarding

The problem: our own `std::make_unique` taking just one argument

```
template <typename T, typename Arg>
std::unique_ptr<T> make_unique(Arg&& arg) {
    return std::unique_ptr<T>( new T(arg) );
}
```

We want to move `arg` if temporary, copy otherwise.

Perfect Forwarding

The problem: our own `std::make_unique` taking just one argument

```
template <typename T, typename Arg>
std::unique_ptr<T> make_unique(Arg&& arg) {
    return std::unique_ptr<T>( new T(arg) );
}
```

We want to move `arg` if temporary, copy otherwise.
Solution: `std::forward<Arg>(arg)`

```
template <typename T, typename Arg>
std::unique_ptr<T> make_unique(Arg&& arg) {
    return std::unique_ptr<T>(std::forward<Arg>(arg));
}
```

Question: Why do we need to write `std::forward<Arg>(arg)`?
Why is `std::forward(arg)` not enough?

Perfect Forwarding

Bad implementation of `std::forward` – what is wrong?

```
template <typename T>
T&& forward(T&& t) {
    return static_cast<T&&>(t);
}
```

Perfect Forwarding

Bad implementation of `std::forward` – what is wrong?

```
template <typename T>
T&& forward(T&& t) {
    return static_cast<T&&>(t);
}
```

Possible correct implementation of `std::forward` (libc++)

```
template <typename T>
T&& forward(std::remove_reference_t<T>& t) noexcept {
    return static_cast<T&&>(t);
}
template <typename T>
T&& forward(std::remove_reference_t<T>&& t) noexcept {
    static_assert(!std::is_lvalue_reference<T>::value,
                  "Cannot forward an rvalue as an lvalue.");
    return static_cast<T&&>(t);
}
```

Consequences of Universal References

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

What is the problem?

Consequences of Universal References

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

What is the problem?

```
int x;
foo(x); // which one is called?
```

Consequences of Universal References

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

What is the problem?

```
int x;
foo(x); // which one is called?
```

- calls the `T&&` version with `T = int&`

Possible Solution 1

One possible solution is using a technique called *tag dispatch*:

```
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>());
}
```

Possible Solution 2

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

- Why do we have to call `foo_impl<T>` and not just `foo_impl`?

Possible Solution 2

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

- Why do we have to call `foo_impl<T>` and not just `foo_impl`?
    - The compiler cannot deduce `T`.

. . . other possible solutions include SFINAE and C++17 constexpr-if (later in this course).

Copy Elision

Copy Elision

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

- `return` local object – Named Return Value Optimisation (NRVO)
- nameless temporary copied or moved to an object or `return`ed – Return Value Optimisation (RVO)
- in both cases, needs to be the same type
- copy elision is the only[1] optimisation which is allowed to change the outcome of a sequential program! (how?)

---

[1] until C++14, since C++14 there are two, see cppreference

Copy Elision

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

- `return` local object – Named Return Value Optimisation (NRVO)
- nameless temporary copied or moved to an object or `return`ed – Return Value Optimisation (RVO)
- in both cases, needs to be the same type
- copy elision is the only[1] optimisation which is allowed to change the outcome of a sequential program! (how?)
  - side effects in elided move/copy constructor

Returning function argument taken by value

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

---

[1] until C++14, since C++14 there are two, see cppreference

Consequences of Copy Elision

```
struct Array { /* ... */ };
Array foo() {
    Array a;
    // do something with
    return a;
}

Array x = foo(); // What methods of Array are called?
```

- move/copy constructor still needs to exist
  (different in C++17, see next slide)

Consequences of Copy Elision

```
struct Array { /* ... */ };
Array foo() {
    Array a;
    // do something with
    return a;
}

Array x = foo(); // What methods of Array are called?
```

- move/copy constructor still needs to exist (different in C++17, see next slide)
- **do not** write `return std::move(x)` if `x` is local or a by-value argument
- *note:* sometimes `return std::move(x)` may make sense; when?

Consequences of Copy Elision

```
struct Array { /* ... */ };
Array foo() {
    Array a;
    // do something with
    return a;
}

Array x = foo(); // What methods of Array are called?
```

- move/copy constructor still needs to exist
  (different in C++17, see next slide)
- **do not** write `return std::move(x)` if `x` is local or a by-value argument
- *note:* sometimes `return std::move(x)` may make sense; when?
  - non-value (e.g. rvalue ref) function arguments
  - complicated expression after `return`

Copy Elision in C++17

C++17

- copy elision is guaranteed in these cases:
  - object initialized by temporary
  - `return` temporary from function (RVO)
- in these cases, move/copy constructors do not need to exist

- copy elision not guaranteed, but allowed:
  - `return` local object (NRVO)

- more information:
  `http://en.cppreference.com/w/cpp/language/copy_elision`