C++11 Forward

last week: move

```
// constructor
vector(): data_(nullptr)
        , size_(0)
        , capacity_(0)
{}

// move-constructor (note: rhs is not const)
vector(vector && rhs):
                     data_(rhs.data_)
                   , size_(rhs.size_)
                   , capacity_(rhs.capacity_)
{
    // rhs's invariants must not be broken!
    // without the following lines we only have a shallow copy...
    rhs.data_ = nullptr;
    rhs.size_ = 0;
    rhs.capacity_ = 0;
}
```

move operation are almost always (much) faster than their corresponding copy operation!

last week: improve `reserve`

```
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
void check_capacity() {
    if(size() == capacity()) {
        if(capacity() == 0)
            reserve(1);
        else
            reserve(2*capacity());
    }
}
void reserve(size_type const & new_cap) {
    value_type * new_data = Alloc::allocate(new_cap);

    for(size_type i = 0; i < size(); ++i)
        Alloc::construct(new_data + i, std::move_if_noexcept(data_[i]));

    for(size_type i = 0; i < size(); ++i)
        Alloc::destruct(data_ + i);

    Alloc::deallocate(data_, capacity());
    data_ = new_data;
    capacity_ = new_cap;
}
```

assume `data_[i]` supports move

last week: improve `allocator`

```
template<typename  T>
struct allocator {

    static T * allocate(std::size_t  const  & n);
    static void deallocate(T  * p, std::size_t const  & n);

    static void construct(T  * p, T const  & t) {
        // this is called a „placement new“. It doesn‘t allocate any memory
        new (p) T(t);
    }
    static void construct(T  * p, T && t) {
        new (p) T(std::move(t));
    }

    static void destruct(T  * p) { p->~T(); }
};
```

why do we need `std::move` again? `t` is already an rvalue reference `T&&`

r-value references

```
int main() {

    // the new c++11 rvalue reference
    int && rref = 10;
    // is rref an rvalue or an lvalue?

    return 0;
}
```

the valueness is **independent** of the type!
together with `&` and `&&` this can lead to initial confusion:
There are:

- **l**-value refs that are **l**-values (`int & a = b;`)
- **r**-value refs that are **l**-values (`int && a = b;`)
- **r**-value refs that are **r**-values (`std::move(b);`)
- ~~**l**-value refs that are **r**-values~~

“Blood Group” compatibility chart

r-value references only accept r-values! That’s why we need `std::move` again

		parameter								
		l-value						r-value		
		`int`	`int const`	`int &`	`int const &`	`int &&`	`int const &&`	`int`	`int &&`	`int const &&`
function	`void fct1(int)`	Y	Y	Y	Y	Y	Y	Y	Y	Y
	`void fct2(int const)`	Y	Y	Y	Y	Y	Y	Y	Y	Y
	`void fct3(int &)`	Y	N	Y	N	Y	N	N	N	N
	`void fct4(int const &)`	Y	Y	Y	Y	Y	Y	Y	Y	Y
	`void fct5(int &&)`	N	N	N	N	N	N	Y	Y	N
	`void fct6(int const &&)`	N	N	N	N	N	N	Y	Y	Y

“Blood Group” overload chart

if multiple overloads of fct exist, constructing this table is more of a challenge…
(we don’t show it here)

		parameter								
		l-value						r-value		
		`int`	`int const`	`int &`	`int const &`	`int &&`	`int const &&`	`int`	`int &&`	`int const &&`
function	`void fct(int)`									
	`void fct(int const)`									
	`void fct(int &)`									
	`void fct(int const &)`									
	`void fct(int &&)`									
	`void fct(int const &&)`									

additional move in `allocator`

```
template<typename  T>
struct allocator {

    static T * allocate(std::size_t const & n);
    static void deallocate(T  * p, std::size_t const & n);

    static void construct(T  * p, T const & t) {
        // this is called a „placement new“. It doesn‘t allocate any memory
        new (p) T(t);
    }
    static void construct(T  * p, T && t) {
        new (p) T(std::move(t));
    }

    static void destruct(T  * p) { p->~T(); }
};
```

code-duplication?

we need `std::move` again to cast `t` to a r-value

problem with move-code?

```
struct my_large_type {
    my_large_type(int const & a);
    // copy
    my_large_type(my_large_type const &) noexcept;
    my_large_type & operator=(my_large_type const &) noexcept;

    // move
    my_large_type(my_large_type &&) noexcept;
    my_large_type & operator=(my_large_type &&) noexcept;
};


int main() {
    my_large_type a(10);

    database db;
    db.add(a, ”admin");                    // user “admin” wants to add by copy
    db.add(my_large_type(10), ”admin"); // user “admin” wants to add by move

    return 0;
}
```

code-duplication

```
class database {
public:
    void logger(my_large_type const & a);
    void access_control(my_large_type const & a, std::string const & user);

    void add(my_large_type const & a, std::string const & user) {
        logger(a);
        access_control(a, user);
        add_helper(a);
    }
    void add(my_large_type && a, std::string const & user) {
        logger(a);
        access_control(a, user);
        add_helper(std::move(a));
    }
private:
    void add_helper(my_large_type const & a) {
        data_.push_back(a);
    }
    void add_helper(my_large_type && a) {
        data_.push_back(std::move(a)); // yes, std::vector can move-construct
    }

    std::vector<my_large_type> data_;
};
```

Universal References

- used to distinguish between l-values and r-values
- whenever the compiler encounters this expression:

```
template<…, typename T, …>
…(…, T && t, …)
```

- it will treat it specially (as a so called universal reference):
  - if what's passed in is a **l-value** of type X
    `t` will be of type **X&**
  - if what's passed in is a **r-value** of type X
    `t` will be of type **X&&**
- This allows us to use the same function for l-values and r-values while still knowing the valueness of what entered

not with universal references

```
class database {
public:
    void logger(my_large_type const & a);
    void access_control(my_large_type const & a, std::string const & user);

    template<typename T>
    void add(T && a, std::string const & user) {
        logger(a);
        access_control(a, user);
        add_helper(std::forward<T>(a));
    }

private:
    template<typename T>
    void add_helper(T && a) {
        data_.push_back(std::forward<T>(a));
    }

    std::vector<my_large_type> data_;
};
```

`std::forward` is a `std::move` for r-values and does nothing for l-values

Universal Reference Gottcha

```
template<typename T>
struct allocator {

    static T * allocate(std::size_t const & n);
    static void deallocate(T * p, std::size_t const & n);

    // this fails: T && is not an universal reference
    // in this context, just a normal r-value reference
    // static void construct(T * p, T && t) {
    //     new (p) T(std::forward<T>(t));
    // }
    // T && (or here U &&) is considered as an universal reference iff
    // T (here U) is a direct template argument of the function
    template<typename U>
    static void construct(T * p, U && t) {
        new (p) T(std::forward<U>(t));
    }

    static void destruct(
};
```

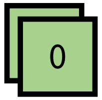
`construct` takes now many more types than just `T` since it is a template…

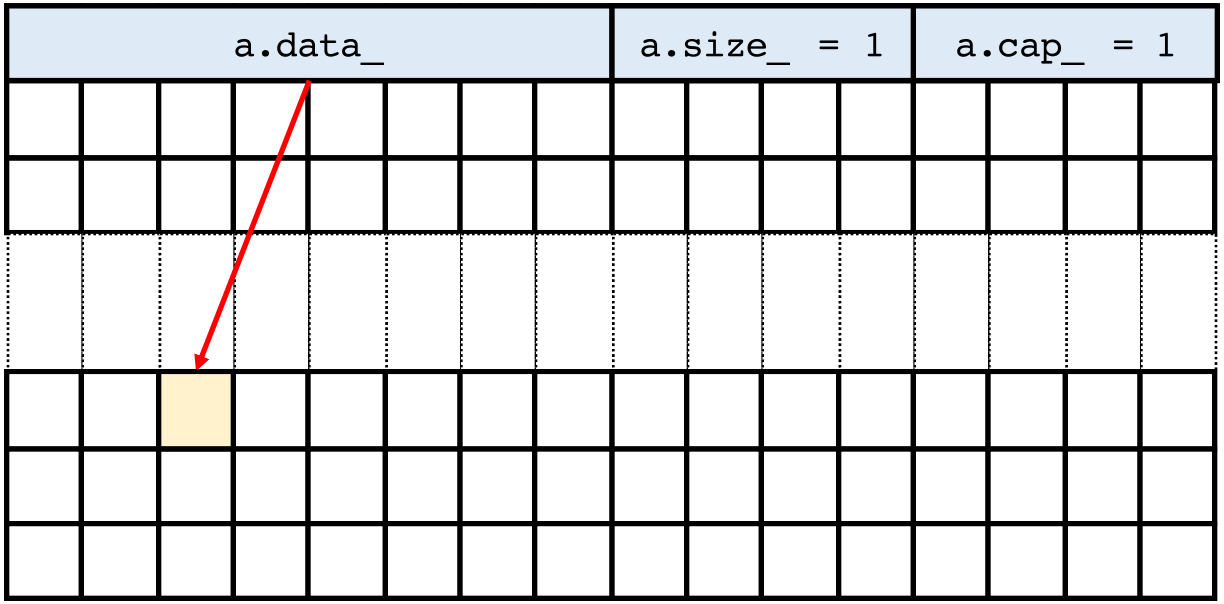
we can disable this easily (later)

additional temporary copy

```
int main() {
    vector<myint> a;

    a.push_back(0);




    return 0;
}
```

```
// if an int is passed, val is a temporary object
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
```



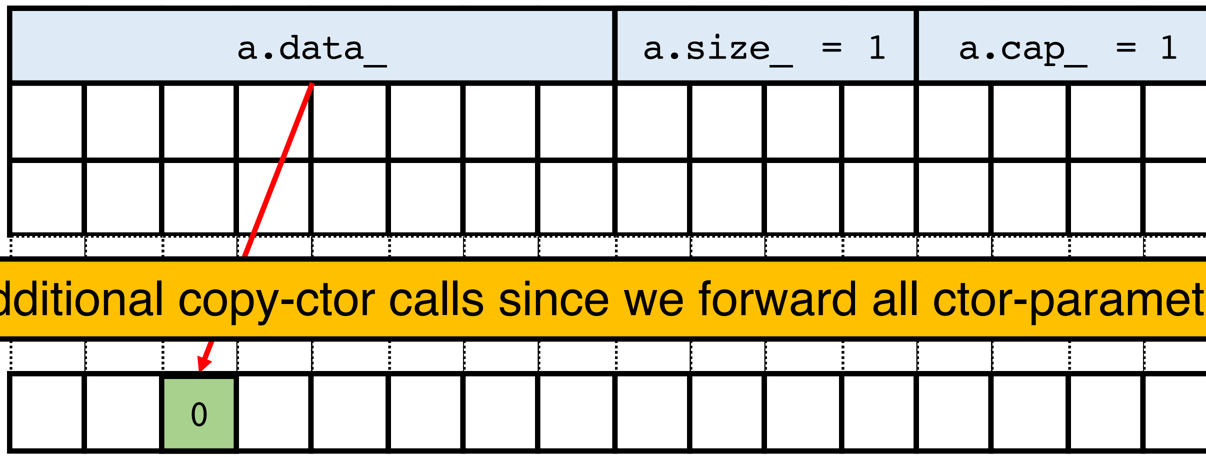


we can remove these additional copy-ctor calls

emplace_back

```
int main() {
    vector<myint> a;

    a.emplace_back(0);

    return 0;
}
```

```
// if an int is passed, val is a temporary object
void push_back(value_type const & val) {
    check_capacity();
    Alloc::construct(data_ + size_++, val);
}
```



no additional copy-ctor calls since we forward all ctor-parameter

```
template<typename... Args> // variadic template (later lecture)
void emplace_back(Args&&... args) {
    check_capacity();
    Alloc::construct(data_ + size_++, std::forward<Args>(args)...);
}
```

upgrade the `allocator` further

```
template<typename T>
struct allocator {

    static T * allocate(std::size_t const & n);
    static void deallocate(T * p, std::size_t const & n);

    template<typename U>
    static void construct(T * p, U && t) {
        new (p) T(std::forward<U>(t));
    }

    // forwards the args to the ctor of T and constructs in place
    template<typename... Args>
    static void construct(T * p, Args&&... args) {
        new (p) T(std::forward<Args>(args)...);
    }

    static void destruct(T * p) { p->~T(); }
};
```

what happens in memory

```
int main() {
    vector<myint>  a;

    a.emplace_back(0);
    a.emplace_back(1);
    a.emplace_back(2);
    a.emplace_back(3);
    a.emplace_back(4);


    return 0;
}
```

= 5

a.cap_ = 8

before with copy and `push_back`:

- 15 byte allocation
- 7 bytes deallocation
- 5 ctor calls
- ***7+5 copy-ctor calls***
- 7+***5*** dtor calls

with move and `push_back`:

- 15 byte allocation
- 7 bytes deallocation
- 5 ctor calls
- ***5 copy-ctor calls***
- ***7 move-ctor calls***
- 7+***5*** dtor calls

with move and `emplace_back`:

- 15 byte allocation
- 7 bytes deallocation
- 5 ctor calls
- ***7 move-ctor calls***
- 7 dtor calls

std::move example

```
// first try
template<typename  T>
constexpr  T && move(T && t) noexcept {
    return static_cast<T &&>(t);
}


// int r-value --> T = int
constexpr int && move(int && t) noexcept {
    return static_cast<int &&>(t);
}
// works

// int l-value --> T = int &
constexpr int & && move(int & && t) noexcept {
    return static_cast<int & &&>(t);
}
```

Reference Collapsing Rules:

- `& & -> &`
- `&& & -> &`
- `& && -> &`
- `&& && -> &&`

if any ref is an l-ref, the collapsed one is an l-ref

std::move example

```
// first try
template<typename  T>
constexpr T && move(T && t) noexcept {
    return static_cast<T &&>(t);
}


// int r-value --> T = int
constexpr int && move(int && t) noexcept {
    return static_cast<int &&>(t);
}
// works

// int l-value --> T = int &
constexpr int & move(int & t) noexcept {
    return static_cast<int &>(t);
}
// this doesn't do what we want...
```

Reference Collapsing Rules:

- `& & -> &`
- `&& & -> &`
- `& && -> &`
- `&& && -> &&`

if any ref is an l-ref, the collapsed one is an l-ref

std::move example

```
// second try
template<typename T>
constexpr std::remove_reference_t<T> && move(T && t) noexcept {
    return static_cast<std::remove_reference_t<T> &&>(t);
}


// int r-value --> T = int
constexpr std::remove_reference_t<int> && move(int && t) noexcept {
    return static_cast<std::remove_reference_t<int> &&>(t);
}
```

std::move example

```
// second try
template<typename T>
constexpr std::remove_reference_t<T> && move(T && t) noexcept {
    return static_cast<std::remove_reference_t<T> &&>(t);
}


// int r-value --> T = int
constexpr int && move(int && t) noexcept {
    return static_cast<int &&>(t);
}
// still works

// int l-value --> T = int &
constexpr std::remove_reference_t<int &> && move(int & && t) noexcept {
    return static_cast<std::remove_reference_t<int &> &&>(t);
}
```

std::move example

```
// second try
template<typename T>
constexpr std::remove_reference_t<T> && move(T && t) noexcept {
    return static_cast<std::remove_reference_t<T> &&>(t);
}


// int r-value --> T = int
constexpr int && move(int && t) noexcept {
    return static_cast<int &&>(t);
}
// still works

// int l-value --> T = int &
constexpr int && move(int & && t) noexcept {
    return static_cast<int &&>(t);
}
```

Reference Collapsing Rules:

- `& & -> &`
- `&& & -> &`
- `& && -> &`
- `&& && -> &&`

if any ref is an l-ref, the collapsed one is an l-ref

std::move example

```
// second try
template<typename T>
constexpr std::remove_reference_t<T> && move(T && t) noexcept {
    return static_cast<std::remove_reference_t<T> &&>(t);
}


// int r-value --> T = int
constexpr int && move(int && t) noexcept {
    return static_cast<int &&>(t);
}
// still works

// int l-value --> T = int &
constexpr int && move(int & t) noexcept {
    return static_cast<int &&>(t);
}
// works now as well
```

how does `std::forward` work?

std::forward challenge

```
template<???>
??? forward(???);
```

week2: auto universal reference

```
int num = 10;
int & num_r = num;
int const num_c = 10;
int const & num_cr = num_c;

// by value
auto a0 = num;         // int
auto a1 = num_r;       // int
auto a2 = num_c;       // int
auto a3 = num_cr;      // int

// reference (or pointer)
auto & a0ref = num;    // int &
auto & a1ref = num_r;  // int &
auto & a2ref = num_c;  // int const &
auto & a3ref = num_cr; // int const &

// universal reference (since auto uses template type deduction)
auto && a0uref = num;  // int & (collapsed from int & &&)
auto && a1uref = 10;   // int &&
```

universal refs can occure in:

- templates
- auto
- templated using

ignores cv-qualifier
ignores ref-ness

ignores ref-ness

no c++11 magic anymore

Templates Types

```
// class template (or struct)
template<class T1, class T2>
class pair;
```

```
// function template
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 &&, T2 &&);

// "typename" and "class" inside <...> behave the same way
```

Templates Specialization

```
// class template (or struct)
template<class T1, class T2>
class pair;

// class template can be partially specialized
template<typename T1>
class pair<T1, int>; // implement differently if T2 == int

// class template can be fully specialized
template<>
class pair<int, int>;

// function template
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 &&, T2 &&);

// function templates cannot be partially specialized
// since we can do the same with overloads
template<typename T1>
pair<T1, int> make_pair(T1 &&, int &&);

// function template can also be fully specialized
template<>
pair<int, int> make_pair<int, int>(int &&, int &&);
```

Type Deducing

```
// class template (or struct)
template<class T1, class T2>
class pair;

// function template
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 &&, T2 &&);


int main() {
    // class template never deduce types via ctor
    // even if it could be done (it's a language design decision)
    pair<double, int> p(1.5, 1); // template parameter mandatory
    auto p = pair(1.5, 1);       // fails: never deduces types

    // function templates deduce the type if possible
    make_pair(1.5, 1);            // ok, <double, int>
    make_pair<int, int>(1.5, 1); // also ok, 1.5 will be an int = 1
}
```

Instantiation

```
// pair.hpp
template<class T1, class T2>
class pair {
    pair(T1 const &, T2 const &);

    T1 first;
    T2 second;
};
```

```
// pair.cpp
#include "pair.hpp"

template<class T1, class T2>
pair<T1, T2>::pair(T1 const & f
                 , T2 const & s)
: first(f), second(s) {}
```

```
// main.cpp
#include "pair.hpp"

int main() {
    pair<double, double> p(1.5, 1.6);
    return 0;
}
```

```
// pair.o (has no clue what is
needed)
```

```
// main.o (has no clue how to
instantiate what's needed)
```

```
// main (linker error: invalid reference
to pair<double, double>::pair)
```

Explicit Instantiation

```cpp
// pair.hpp
template<class T1, class T2>
class pair {
    pair(T1 const &, T2 const &);

    T1 first;
    T2 second;
};
```

```cpp
// pair.cpp
#include "pair.hpp"

template<class T1, class T2>
pair<T1, T2>::pair(T1 const & f
                  , T2 const & s)
: first(f), second(s) {}
// explicit instantiation
template class pair<double, double>;
```

```cpp
// main.cpp
#include "pair.hpp"

int main() {
    pair<double, double> p(1.5, 1.6);
    return 0;
}
```

```
// pair.o
```

```
// main.o (still has no clue how to
instantiate what's needed)
```

```
// main (only works for
pair<double, double>)
```

Header-only

```cpp
// pair.hpp
template<class T1, class T2>
class pair {
    pair(T1 const & f, T2 const & s)
    : first(f), second(s) {}

    T1 first;
    T2 second;
};
```

```cpp
// main.cpp
#include "pair.hpp"

int main() {
    pair<double, double> p(1.5, 1.6);
    return 0;
}
```

```
// main.o (knows now how to
instantiate what's needed)
```

```
// main
```

Instantiation

```
// class template (or struct)
template<class T1, class T2>
class pair;

// function template
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 &&, T2 &&);


// class templates can be instantiated explicitly to reduce compiletime
template class pair<double, double>;

// function templates can be instantiated explicitly to reduce compiletime
template pair<double, double> make_pair(double&&, double&&);
template pair<double, double> make_pair<double, double>(double&&, double&&);
```

SFINAE

```
//doesn't work for int since int doesn't have size_type
template<typename T>
void fct(typename T::size_type t);

//doesn't work for foo since 10 (in the call) is not convertible to foo
template<typename T>
void fct(T t);

struct foo {
    using size_type = int;
};

int main() {

    fct(10);       // deduction works here
    fct<foo>(10);  // no deduction possible, that‘s why the <foo> is needed

    // SFINAE
    // Substitution Failure Is Not An Error

    return 0;
}
```

Uniform Distribution

```
template<typename T>
enable_if_t<std::is_integral<T>::value, T>
  uniform_dist(double p, T lower, T upper) {
    // very simplified version
    return lower + p * (upper + 1 - lower);
}

template<typename T>
enable_if_t<!std::is_integral<T>::value, T>
  uniform_dist(double p, T lower, T upper) {
    // very simplified version
    return lower + p * (upper - lower);
}

int main() {
    // p is a random number in [0, 1) from a uniform distribution
    uniform_dist<int8_t>(p, 1, 6); // should return between [1, 6]
    uniform_dist<double>(p, 1, 6); // should return between [1, 6)
    return 0;
}
```

enable_if_t

```
template<bool cond, typename T>
struct enable_if {
    using type = T;
};

// partial specialization
template<typename T>
struct enable_if<false, T> {
    // don't define type -> SFINAE prevents instantiation
};

// only makes the usage nicer
template<bool cond, typename T>
using enable_if_t = typename enable_if<cond, T>::type;
// ::type only exist if cond == true
```