C++ references

CoreHard Winter 2017

solarwinds msp

LOGICnow

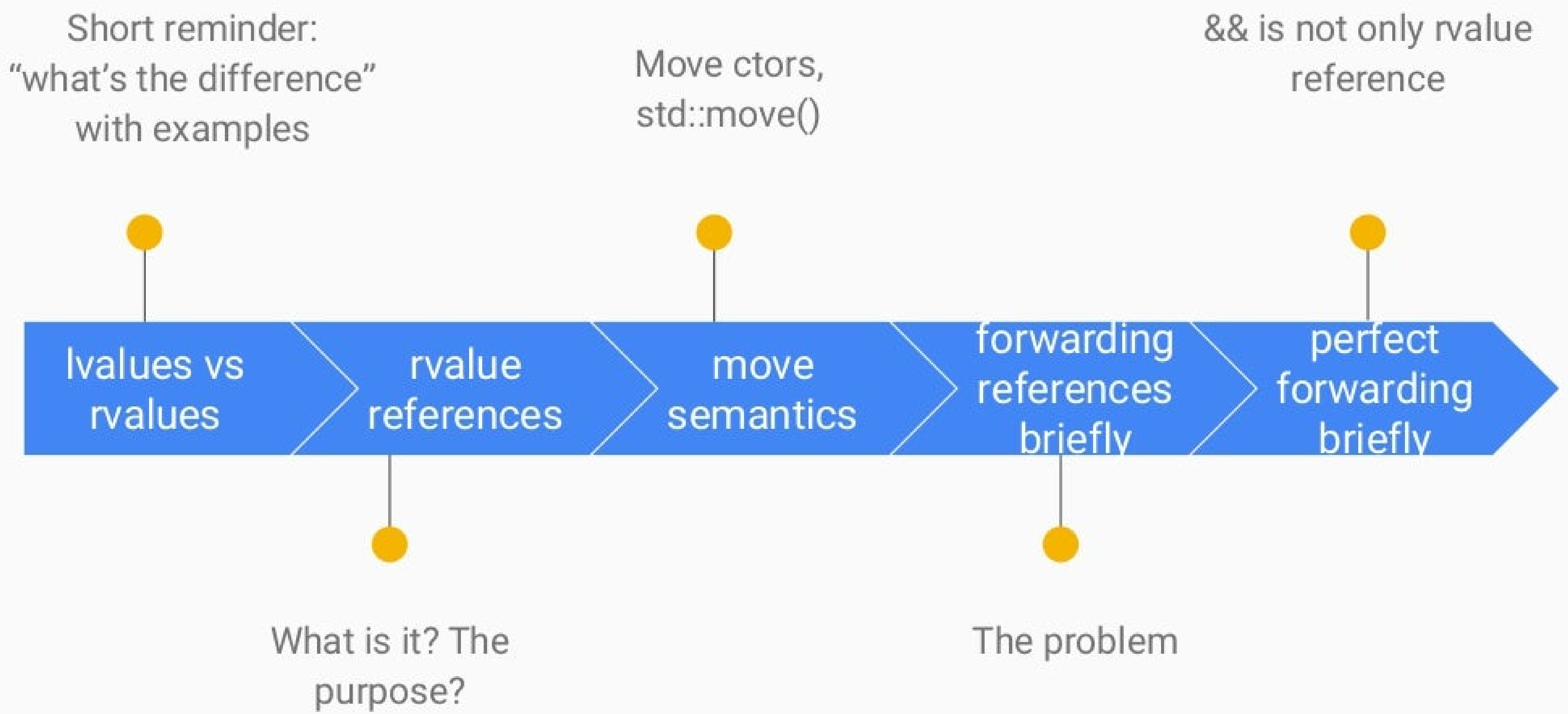
My name is Gavrilovich Yury =)

Objective

- Overview of “ancient” C++11 feature: rvalue references
- Leave in your memory not numerous details but high-level ideas!
- Inspire to get profound understanding by yourselves! (Homework presents)

For those who is not aware of:

- Rvalue references (&&)
- Forwarding (universal) references
- Move semantics
- Perfect forwarding


Short reminder:
“what’s the difference”
with examples
lvalues vs rvalues
rvalue references
What is it? The purpose?
Move ctors,
std::move()
move semantics
forwarding references briefly
The problem
&& is not only rvalue reference
perfect forwarding briefly

lvalues vs rvalues

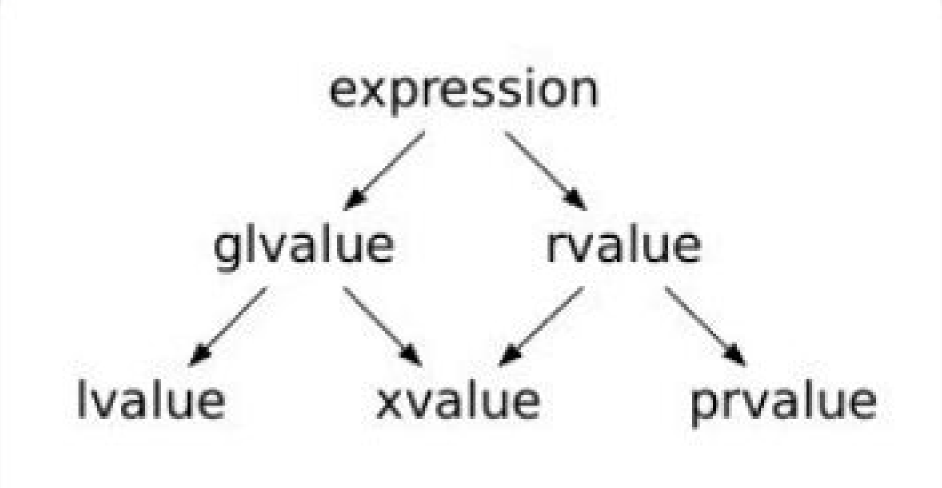
rvalue references

move semantics

forwarding references briefly

perfect forwarding briefly

Formal definition from C++ standard



(1.1) — An *lvalue* (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [*Example:* If `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object or function to which `E` points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. — *end example*]

(1.2) — An *xvalue* (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). Certain kinds of expressions involving rvalue references (8.3.2) yield xvalues. [*Example:* The result of calling a function whose return type is an rvalue reference to an object type is an xvalue (5.2.2). — *end example*]

(1.3) — A *glvalue* ("generalized" lvalue) is an lvalue or an xvalue.

(1.4) — An *rvalue* (so called, historically, because rvalues could appear on the right-hand side of an assignment expression) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.

(1.5) — A *prvalue* ("pure" rvalue) is an rvalue that is not an xvalue. [*Example:* The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as `12`, `7.3e5`, or `true` is also a prvalue. — *end example*]

Informal definition (good enough)

- If you can take the address of an expression, the expression is an **lvalue**
- If the type of an expression is an lvalue reference (e.g., `T&` or `const T&`, etc.), that expression is an **lvalue**
- Otherwise, the expression is an **rvalue**. (Examples: temporary objects, such as those returned from functions or created through implicit type conversions. Most literal values (e.g., 10 and 5.3))

Homework #1

- Function *parameters* (not *arguments*) are lvalues
- The type of an expression is independent of whether the expression is an lvalue or an rvalue. That is, given a type T, you can have lvalues of type T as well as rvalues of type T. [Effective Modern C++. Introduction]

lvalue examples

(can take address)

```
int i = 13;           // i is lvalue
int* pi = &i;         // can take address
```

```
int& foo();
foo() = 13;           // foo() is lvalue
int* pf = &foo();     // can take address
```

```
int i = 14;
int& ri = i;          // ri is lvalue
int* pri = &ri;       // can take address
```

lvalue examples

(can take address)

```
template <typename T>
T& MyVector::operator[] (std::size_t const n);

MyVector<int> v{1,3,5,7};
v[0] = 2;               // v[0] is lvalue
int* pv = &v[0];        // can take address
```

rvalue examples

(can NOT take address)

```
int i = 13;           // 13 is rvalue
i += 1;               // 1 is rvalue
```

yury.gavrilovich@yurys-Mac ~/$ objdump -d -x86-asm-syntax=intel a.out

```
mov     dword ptr [rbp - 8], 13
mov     ecx, dword ptr [rbp - 8]
add     ecx, 1
```

rvalue examples

(can NOT take address)

```
a + b = 3;    // error: lvalue required as left operand of assignment
```

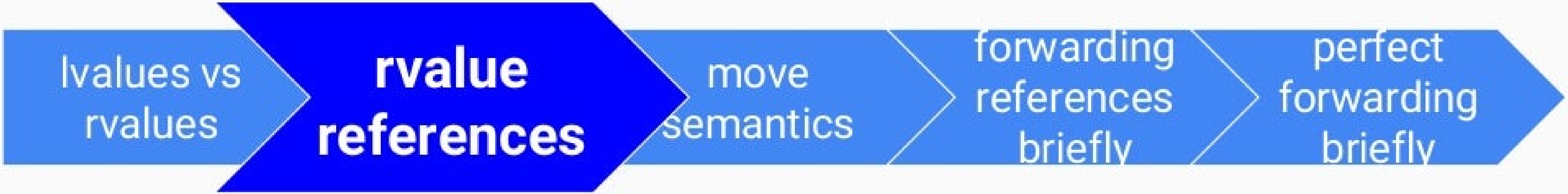
```
int bar();    // bar() is rvalue
bar() = 13;   // error: expression is not assignable
int* pf = &bar(); // error: cannot take the address of an rvalue of type 'int'
```

```
struct X{};
X x = X();      // X() is rvalue
X* px = &X();   // error: taking the address of a temporary object of type 'X'.
```

Question for your understanding:

```
const int i = 1;
```

rvalue or **lvalue**?


lvalues vs rvalues
rvalue references
move semantics
forwarding references briefly
perfect forwarding briefly

lvalue references (&)

```
X x;
X& lv = x;
X& lv2 = X();  // error:  non-const lvalue reference to
type 'X' cannot bind to a temporary of type 'X'
X const& lv3 = X(); // OK, since const&
```

rvalue references (&&)

```
X x;

X&& rv2 = x;   // error: rvalue reference to type 'X' cannot bind
to lvalue of type 'X'
X&& rv = X();  // OK
```

rvalue references (&&)

rvalue references is a small technical extension to the C++ language. Rvalue references allow programmers to **avoid logically unnecessary copying** and to provide **perfect forwarding** functions. They are primarily meant to aid in the design of higher performance and more robust libraries.

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
"A Brief Introduction to Rvalue References"
Document number: N2027=06-0097
Howard E. Hinnant, Bjarne Stroustrup, Bronek Kozicki
2006-06-12

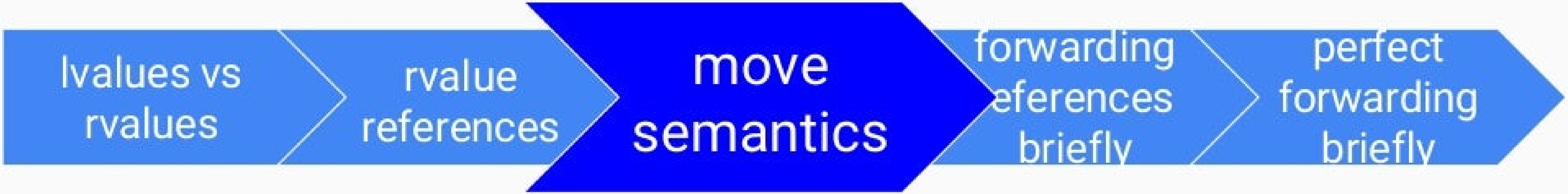
Besides resolving aforementioned problems

Rvalue references allow branching at compile time:

```
void foo(int& a); // lvalue reference overload
void foo(int&& a); // rvalue reference overload

int a = 1;
int bar();

foo(a); // argument is lvalue: calls foo(int&)
foo(bar()); // argument is rvalue: calls foo(int&&)
```


lvalues vs rvalues
rvalue references
move semantics
forwarding eferences briefly
perfect forwarding briefly

Our Vector class

```
template <typename T>
class Vector
{
public:
    explicit Vector(std::size_t const size);
    ~Vector();

    Vector(Vector const& other);

    Vector& operator= (Vector const& rhs);

    // ...
    // ...

private:
    std::size_t m_size;
    T* m_data;
};
```

Vector class. Constructor, Destructor

```cpp
template <typename T>
Vector<T>::Vector(std::size_t const size) :
    m_size(size),
    m_data(new T[size])
{
    std::cout << this << " ctor" <<
std::endl;
};
```

```cpp
template <typename T>
Vector<T>::~Vector()
{
    std::cout << this << " ~dtor" << std::endl;
    delete[] m_data;
}
```

Vector class. Naive implementation (1-3 problems at least)

```
template <typename T>
Vector<T>::Vector(Vector const& other) :
    m_size(other.m_size),
    m_data(new T[m_size])
{
    std::cout << this << " copy ctor" <<
std::endl;
    std::cout << this << "\t copying data into
allocated memory" << std::endl;

    std::copy(other.m_data, other.m_data + m_size,
m_data);
}
```

```
template <typename T>
Vector<T>& Vector<T>::operator= (Vector const& rhs)
{
    std::cout << this << " copy assignment operator" <<
std::endl;
    if (this == &rhs)
    {
        return *this;
    }

    std::cout << this << "\t removing old data" <<
std::endl;
    delete[] m_data;

    m_size = rhs.m_size;

    std::cout << this << "\t allocating and copying
data" << std::endl;
    m_data = new T[rhs.m_size];
    std::copy(rhs.m_data, rhs.m_data + m_size, m_data);

    return *this;
}
```

C++ exception safety levels (wikipedia)

1) **No-throw guarantee**, also known as failure transparency
2) **Strong exception safety**, also known as **commit or rollback semantics**
3) **Basic exception safety**, also known as a **no-leak guarantee**
4) **No exception safety**: No guarantees are made.

A better implementation (copy_and_swap idiom - strong exception guarantee)

```
template <typename T>
Vector<T>::Vector(Vector const& other) :
    m_size(other.m_size),
    m_data(new T[m_size])
{
    std::cout << this << " copy ctor" <<
std::endl;
    std::cout << this << "\t copying data
into allocated memory" << std::endl;

    std::copy(other.m_data, other.m_data +
m_size, m_data);
}
```

```
template <typename T>
Vector<T>& Vector<T>::operator= (Vector const& rhs)
{
    std::cout << this << " copy assignment operator"
<< std::endl;

    Vector<T> tmp(rhs);
    std::swap(m_size, tmp.m_size);
    std::swap(m_data, tmp.m_data);

    return *this;
}
```

The problem

```
Vector<int> v1(1000);
Vector<int> v2(1);
v2 = v1;  // Reasonable copy, since v1 is
lvalue
```

```
v2 = Vector<int>(2);  // Unnecessary copy
```

```
template<typename T>
Vector<T> CreateVector(std::size_t const n);

v2 = CreateVector<int>(3); // Either
unnecessary copy
```

Problem example

```
Vector<int> v3(1000);

std::cout << "== create temporary in place" << std::endl;
v3 = Vector<int>(2);

std::cout << "== creating temporary through return value"
<< std::endl;
v3 = CreateVector<int>(1000);
```

```
yury.gavrilovich@yurys-Mac ~/ $ clang++ -std=c++11 04.cc && ./a.out
0x7fff583cb3c8 ctor

== create temporary in place
0x7fff583cb3a8 ctor
0x7fff583cb3c8 copy assignment operator
0x7fff583cb310 copy ctor
0x7fff583cb310     allocating and copying data
0x7fff583cb310 ~dtor
0x7fff583cb3a8 ~dtor

== creating temporary through return value
0x7fff583cb398 ctor
0x7fff583cb3c8 copy assignment operator
0x7fff583cb310 copy ctor
0x7fff583cb310     allocating and copying data
0x7fff583cb310 ~dtor
0x7fff583cb398 ~dtor
0x7fff583cb3c8 ~dtor
```

Just MOVE it

```
template <typename T>
Vector<T>::Vector(Vector&& other) :
    m_data(nullptr)
{
    std::cout << this << " move ctor" <<
std::endl;
    std::swap(m_size, other.m_size);
    std::swap(m_data, other.m_data);
}
```

```
template <typename T>
Vector<T>& Vector<T>::operator= (Vector&& rhs)
{
    std::cout << this << " move assignment
operator" << std::endl;
    std::swap(m_size, rhs.m_size);
    std::swap(m_data, rhs.m_data);
    return *this;
}
```

Just MOVE it

```
Vector<int> v3(1000);

Std::cout << "== create temporary in place" <<
std::endl;
v3 = Vector<int>(2);

std::cout << "== creating temporary through return
value" << std::endl;
v3 = CreateVector<int>(1000);
```

```
yury.gavrilovich@yurys-Mac ~/ $ clang++ -std=c++11  -O0 041.cc &&
./a.out
0x7fff5e9193c8 ctor

== create temporary in place
0x7fff5e9193a8 ctor
0x7fff5e9193c8 move assignment operator
0x7fff5e9193a8 ~dtor

== creating temporary through return value
0x7fff5e919398 ctor
0x7fff5e9193c8 move assignment operator
0x7fff5e919398 ~dtor
0x7fff5e9193c8 ~dtor
```

Anything besides move constructors? You can std::move() things!

But… what is std::move()?

std::move() - makes T&& from anything since 2011

Implementation:

```
template<class T>
typename remove_reference<T>::type&&
std::move(T&& a) noexcept
{
 typedef typename remove_reference<T>::type&& RvalRef;
 return static_cast<RvalRef>(a);
}
```

Equivalent to:

```
static_cast<typename std::remove_reference<T>::type&&>(t)
```

Example:

```
Vector<int> v3(1);
auto x1 = std::move(v3);       // explicit intention
auto x2 = static_cast<Vector<int>&&>(x1);   // ugly
```

std::move example 1

```
Vector<int> v1(1000);
Vector<int> v2 = std::move(v1);  // don't care about v1 anymore
Vector<int> v3(v1); // easy way to get segfault
```

```
lite @ yurketPC ~/ $ g++ -std=c++11 -O0 041.cc && ./a.out
0x7ffe881fe0b0 ctor
0x7ffe881fe0c0 move ctor
0x7ffe881fe0d0 copy ctor
0x7ffe881fe0d0   copying data into allocated memory
Segmentation fault
```

std::move example 2a

```
Vector<double> v1(1000);
std::vector<Vector<double>> v;

v.push_back(v1);
```

```
lite @ yurketPC ~/ $ g++ -std=c++11 -O0 041.cc && ./a.out
0x7ffc61c22930 ctor
0xd52f80 copy ctor
0xd52f80        copying data into allocated memory
0xd52f80 ~dtor
0x7ffc61c22930 ~dtor
```

```
lite @ yurketPC ~/ $ valgrind ./a.out
==23980== HEAP SUMMARY:
==23980==     in use at exit: 72,704 bytes in 1 blocks
==23980==   total heap usage: 5 allocs, 4 frees, 89,744 bytes
allocated
```

std::move example 2b

```
Vector<double> v1(1000);
std::vector<Vector<double>> v;

v.push_back(std::move(v1));
```

```
lite @ yurketPC ~/ $ g++ -std=c++11 -O0 041.cc && ./a.out
0x7ffcc5fecc00 ctor
0x1ebaf80 move ctor
0x1ebaf80 ~dtor
0x7ffcc5fecc00 ~dtor
```

```
lite @ yurketPC ~/ $ valgrind ./a.out
==24060== HEAP SUMMARY:
==24060==     in use at exit: 72,704 bytes in 1 blocks
==24060==   total heap usage: 4 allocs, 3 frees, 81,744 bytes allocated
```

Movable only types

(nice “side effect”)

- Non-value types
- Only 1 instance should exist

- `unique_ptr` is a good example
- Poco MongoDB connections

Movable but not copyable example

```
typedef std::unique_ptr<Vector<int>> VectorPtr;
std::vector<VectorPtr> v1, v2;
v1.push_back(VectorPtr(new Vector<int>(10)));


v2 = v1; // error: use of deleted function unique_ptr<T>&
unique_ptr<T>::operator=(const unique_ptr<T>&)
v2 = std::move(v1); // OK
```

So. Move semantics is good because...

- Improves performance for new code (inplace sorting in STL containers)
- Free performance gain by upgrading from C++03 to C++11
- Allows movable only types


lvalues vs rvalues
rvalue references
move semantics
forwarding references briefly
perfect forwarding briefly

Forwarding references (T&&)

Forwarding reference is special reference in type deduction context (in type declaration, or template parameters) which can be resolved to *either* rvalue reference *or* lvalue reference

Forwarding references. Example 1

```
int&& rr = 13;                  // explicitly declared rvalue

// type deduction taking place
int i = 14;
auto&& ar = i;                  // ar is lvalue reference of type int&
auto&& ar2 = 15;               // ar2 is rvalue reference of type int&&
```

Forwarding references. Example 2

```
template <typename T>
void foo(T&& arg)
{
    ...
}

int i = 13;
foo(i);     // arg is of type int&
foo(5);     // arg is of type int&&
```

Reference collapsing rule on type deduction

The rule is very simple. & always wins.

- A& & becomes A&
- A& && becomes A&
- A&& & becomes A&
- A&& && becomes A&&

Special type deduction rules

```
template<typename T>
void foo(T&&);
```

1. When foo is called on an **lvalue** of type A, then T resolves to A& and hence, by the reference collapsing rules above, the argument type effectively becomes **A&**.
2. When foo is called on an **rvalue** of type A, then T resolves to A, and hence the argument type becomes **A&&**.

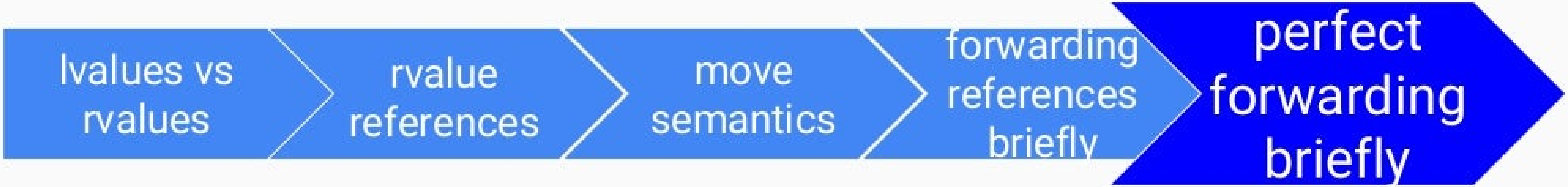
Forwarding references. Example 2

```
template <typename T>
void foo(T&& arg)
{
    ...
}

int i = 13;
foo(i);     // foo(int& && arg) -> arg is of type int&
foo(5);     // foo(int && arg) -> arg is of type int&&
```

The main thing to remember about T&& -

Forwarding reference preserve the value category (lvaluesness or rvaluesness) of its “argument”. Or we can say they FORWARD value category.


lvalues vs rvalues
rvalue references
move semantics
forwarding references briefly
perfect forwarding briefly

Problem is pretty old

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1385.htm

Document number: N1385=02-0043

Programming Language C++

Peter Dimov, pdimov@mmltd.net

Howard E. Hinnant, hinnant@twcny.rr.com

Dave Abrahams, dave@boost-consulting.com

September 09, 2002

The problem

We'd like to define a function `f(t1, …, tn)` with generic parameters that forwards its parameters *perfectly* to some other function `E(t1, …, tn)`.

The problem pseudocode

```
template <typename T1, typename T2>
void f(T1? t1, T2? t2)
{
    E(?t1, ?t2);
}
```

Homework #2: Try to derive such a function (or set of functions) for a number of parameter types: `T`, `T&`, `const T&` in terms of C++03

The solution

```
template<typename T1, typename T2>
void f(T1&& t1, T2&& t2)
{
    E(std::forward<T1>(t1),
std::forward<T2>(t2));
}
```

1) forwarding references
2) std::forward()

std::forward( )

```
template<class T>
T&& forward(typename std::remove_reference<T>::type& t)
noexcept {
  return static_cast<T&&>(t);
}
```

Forwards lvalues as either lvalues or as rvalues, depending on `T` (much easier to grasp this concept after `Homework #1`)

What to remember?

lvalues vs rvalues

rvalue references

move semantics

forwarding references briefly

perfect forwarding briefly

The end

Homework ##: RVO + copy elision

```
Vector& operator= (Vector const& rhs)
{
    std::cout << this << " copy assignment operator"
<< std::endl;
    Vector<T> tmp(rhs);
    swap(*this, tmp);
    return *this;
}
```

```
Vector& operator= (Vector rhs)
{
    std::cout << this << " copy assignment operator" <<
std::endl;
    swap(*this, rhs);
    return *this;
}
```

Why do we need move ctor then?

How more than 1 reference could be?

```
template <typename T>
void bar(T t) {
  T& v = t;        // int& & v = t;
}

int i = 13;
bar<int&>(i);
```