Rvalue References, Move Semantics, and the Magic Thereof

Petr Zemek

Principal Developer
Security/Engineering/VirusLab
https://petrzemek.net

All About the ~~Bass~~ && !

C++ Developers' Guild, 2016-07-01

Whetting Our Appetite

- `&&` is no longer just logical `and`

Whetting Our Appetite

- `&&` is no longer just logical `and`
- `x` in `void func(std::string&& x);` is an lvalue

Whetting Our Appetite

- `&&` is no longer just logical `and`
- `x` in `void func(std::string&& x);` is an lvalue
- `type&&` $\not\Rightarrow$ rvalue reference

Whetting Our Appetite

- `&&` is no longer just logical `and`
- `x` in `void func(std::string&& x);` is an lvalue
- `type&&` $\not\Rightarrow$ rvalue reference
- `std::move(x)` does not do any moving

Whetting Our Appetite

- `&&` is no longer just logical `and`
- `x` in `void func(std::string&& x);` is an lvalue
- `type&&` $\not\Rightarrow$ rvalue reference
- `std::move(x)` does not do any moving
- `return std::move(x);` is usually a Bad Idea™

Lvalues and Rvalues

What is an lvalue and rvalue?

Lvalues and Rvalues

What is an lvalue and rvalue?

Historical origin:

- An **l***value* is an expression that may appear on the **l**eft-hand side of an assignment.
- An **r***value* is an expression that can only appear on the **r**ight-hand side of an assignment.

Lvalues and Rvalues

What is an lvalue and rvalue?

Historical origin:

- An **l***value* is an expression that may appear on the **l**eft-hand side of an assignment.
- An **r***value* is an expression that can only appear on the **r**ight-hand side of an assignment.

Not useful for C++:

```
std::string("a") = "b"; // OK
```

Lvalues and Rvalues

What is an lvalue and rvalue?

Historical origin:

- An **l***value* is an expression that may appear on the **l**eft-hand side of an assignment.
- An **r***value* is an expression that can only appear on the **r**ight-hand side of an assignment.

Not useful for C++:

```
std::string("a") = "b"; // OK

std::unique_ptr<int> p, q;
p = q; // error: use of deleted operator=()
```

There Has To Be a Better Way...

A better approach: Can I take its address?

There Has To Be a Better Way...

A better approach: Can I take its address?

```
int i;
int purr();
int& meow();
int* q;
```

There Has To Be a Better Way...

A better approach: Can I take its address?

```
int i;
int purr();
int& meow();
int* q;

i = 42; // i is lvalue, 42 is rvalue
&i;     // OK
&42;    // error: lvalue required as unary '&' op.
```

There Has To Be a Better Way...

A better approach: Can I take its address?

```
int i;
int purr();
int& meow();
int* q;

i = 42; // i is lvalue, 42 is rvalue
&i;     // OK
&42;    // error: lvalue required as unary '&' op.

purr() = i + 1; // error: lvalue required as left op.
&purr(); // error: lvalue required as unary '&' op.
```

There Has To Be a Better Way...

A better approach: Can I take its address?

```
int i;
int purr();
int& meow();
int* q;

i = 42; // i is lvalue, 42 is rvalue
&i;     // OK
&42;    // error: lvalue required as unary '&' op.

purr() = i + 1; // error: lvalue required as left op.
&purr(); // error: lvalue required as unary '&' op.

&purr; // OK
```

There Has To Be a Better Way...

A better approach: Can I take its address?

```
int i;
int purr();
int& meow();
int* q;

i = 42; // i is lvalue, 42 is rvalue
&i;      // OK
&42;     // error: lvalue required as unary '&' op.

purr() = i + 1; // error: lvalue required as left op.
&purr(); // error: lvalue required as unary '&' op.

&purr; // OK

meow() = i + 1; // meow() is lvalue, i + 1 is rvalue
&meow();   // OK
&(i + 1); // error: lvalue required as unary '&' op.
```

There Has To Be a Better Way...

A better approach: Can I take its address?

```
int i;
int purr();
int& meow();
int* q;

i = 42; // i is lvalue, 42 is rvalue
&i;     // OK
&42;    // error: lvalue required as unary ’&’ op.

purr() = i + 1; // error: lvalue required as left op.
&purr(); // error: lvalue required as unary ’&’ op.

&purr; // OK

meow() = i + 1; // meow() is lvalue, i + 1 is rvalue
&meow();  // OK
&(i + 1); // error: lvalue required as unary ’&’ op.

q = new int[8]; // new int[8] is rvalue
*(q + 1) = 4;   // *(q + 1) is lvalue
```

A Tale of Two References

- lvalue references

```
int i = 0;
int& m = i;
const int& n = 1;
```

- rvalue references (since C++11)

```
int&& p = 42;
```

A Tale of Two References

- lvalue references

```cpp
int i = 0;
int& m = i;
const int& n = 1;
```

- rvalue references (since C++11)

```cpp
int&& p = 42;
```

```cpp
int& a = 3;          // error: cannot bind
```

A Tale of Two References

- lvalue references

```
int i = 0;
int& m = i;
const int& n = 1;
```

- rvalue references (since C++11)

```
int&& p = 42;
```

```
int& a = 3;          // error: cannot bind
const int& b = 3;   // OK
```

A Tale of Two References

- lvalue references

```
int i = 0;
int& m = i;
const int& n = 1;
```

- rvalue references (since C++11)

```
int&& p = 42;
```

```
int& a = 3;          // error: cannot bind
const int& b = 3;    // OK
int&& c = 3;         // OK
```

A Tale of Two References

- lvalue references

```cpp
int i = 0;
int& m = i;
const int& n = 1;
```

- rvalue references (since C++11)

```cpp
int&& p = 42;
```

```cpp
int& a = 3;          // error: cannot bind
const int& b = 3;    // OK
int&& c = 3;         // OK
int&& d = i;         // error: cannot bind
```

A Tale of Two References

- lvalue references

```cpp
int i = 0;
int& m = i;
const int& n = 1;
```

- rvalue references (since C++11)

```cpp
int&& p = 42;
```

```cpp
int& a = 3;          // error: cannot bind
const int& b = 3;    // OK
int&& c = 3;         // OK
int&& d = i;         // error: cannot bind
const int&& e = 3;   // OK
```

L/Rvalueness Is Independent of Type

Hint: If it has a name, it is an lvalue.

L/Rvalueness Is Independent of Type

Hint: If it has a name, it is an lvalue.

```
class Cat;

void woof(Cat&& c) { // c is an lvalue
    Cat* p = &c;     // OK
}
```

L/Rvalueness Is Independent of Type

Hint: If it has a name, it is an lvalue.

```
class Cat;

void woof(Cat&& c) { // c is an lvalue
    Cat* p = &c;     // OK
}
```

```
int&& i = 1; // i is an lvalue, 1 is an rvalue,
             // the type of i is rvalue reference
```

Rvalue References: Raison D’être

1. move semantics
2. perfect forwarding

Move Semantics

Motivation Behind Move Semantics

```cpp
template <typename T> // Ignoring allocator...
class vector {
public:
    vector<T>& operator=(const vector<T>& other) {
        // ...
        // Make a copy of other.buffer.
        // Release buffer.
        // Assign the copy to buffer.
        // ...
    }
    // ...
private:
    T* buffer;
    // ...
};
```

Motivation Behind Move Semantics

```
template <typename T> // Ignoring allocator...
class vector {
public:
    vector<T>& operator=(const vector<T>& other) {
        // ...
        // Make a copy of other.buffer.
        // Release buffer.
        // Assign the copy to buffer.
        // ...
    }
    // ...
private:
    T* buffer;
    // ...
};

std::vector<int> readInput();
// ...
v = readInput();
```

Motivation Behind Move Semantics (Continued)

```
template <typename T> // Ignoring allocator...
class vector {
public:
    vector<T>& operator=(const vector<T>& other) {
        // ...
        // Make a copy of other.buffer.
        // Release buffer.
        // Assign the copy to buffer.
        // ...
    }

    vector<T>& operator=(vector<T>&& other) {
        // ...
        // Release buffer.
        // Assign other.buffer to buffer.
        // ...
    }
// ...
};
```

Motivation Behind Move Semantics (Continued)

```
template <typename T> // Ignoring allocator...
class vector {
public:
    vector(const vector<T>& other) {
        // ...
        // Make a copy of other.buffer.
        // Release buffer.
        // Assign the copy to buffer.
        // ...
    }

    vector(vector<T>&& other) {
        // ...
        // Release buffer.
        // Assign other.buffer to buffer.
        // ...
    }
// ...
};
```

Actual Implementation (Still Simplified)

```
T* buffer;
std::size_t size;
std::size_t capacity;
```

Actual Implementation (Still Simplified)

```
T* buffer;
std::size_t size;
std::size_t capacity;

vector<T>& operator=(vector<T>&& other) {
    delete[] buffer;
    buffer = other.buffer;
    other.buffer = nullptr;

    size = other.size;
    other.size = 0;

    capacity = other.capacity;
    other.capacity = 0;

    return *this;
}
```

Actual Implementation (Still Simplified, Cont'd)

```
vector<T>& operator=(vector<T>&& other) noexcept {
    assert (this != &other);
    if (this == &other) return *this;

    delete[] buffer;
    buffer = other.buffer;
    other.buffer = nullptr;

    size = other.size;
    other.size = 0;

    capacity = other.capacity;
    other.capacity = 0;

    return *this;
}
```

Forcing Move Semantics

The First Amendment to the C++ Standard states (j/k):

> *The committee shall make no rule that prevents C++ programmers from shooting themselves in the foot.*

Forcing Move Semantics (Continued)

So, how do I force a move?

Forcing Move Semantics (Continued)

So, how do I force a move?

```
void devour(std::vector<int> x);

void foo() {
    std::vector<int> v;
    // ...
    devour(v); // OK, but copies v
}
```

Forcing Move Semantics (Continued)

So, how do I force a move?

```
void devour(std::vector<int> x);

void foo() {
    std::vector<int> v;
    // ...
    devour(rvalue_cast(v)); // ?!
}
```

Forcing Move Semantics (Continued)

So, how do I force a move?

```
void devour(std::vector<int> x);

void foo() {
    std::vector<int> v;
    // ...
    devour(std::move(v)); // OK, v is moved
}
```

What Does std::move() Do, Anyway?

An almost conforming implementation:

```
// C++11, in namespace std
template <typename T>
typename remove_reference<T>::type&&
move(T&& param) {
    using ReturnType =
        typename remove_reference<T>::type&&;
    return static_cast<ReturnType>(param);
}
```

What Does std::move() Do, Anyway?

An almost conforming implementation:

```
// C++11, in namespace std
template <typename T>
typename remove_reference<T>::type&&
move(T&& param) {
    using ReturnType =
        typename remove_reference<T>::type&&;
    return static_cast<ReturnType>(param);
}

// C++14, in namespace std
template <typename T>
decltype(auto)
move(T&& param) {
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

Another Application of `std::move()`

```cpp
class Person {
public:
// ...

    void setName(std::string&& n) {
        name = std::move(n); // Why move()?
    }

// ...

private:
    std::string name;
};
```

Yet Another: Move-Only Types

```
void transmogrify(std::unique_ptr<Person> p);

auto p = std::make_unique<Person>("Steve Kady");
// ...
transmogrify(p); // error: use of deleted function
```

Yet Another: Move-Only Types

```cpp
void transmogrify(std::unique_ptr<Person> p);

auto p = std::make_unique<Person>("Steve Kady");
// ...
transmogrify(std::move(p)); // OK
```

std::move() Overdose

```cpp
std::vector<int> readInput() {
    std::vector<int> v;

    // ...

    return v;
}
```

std::move() Overdose

```
std::vector<int> readInput() {
    std::vector<int> v;

    // ...

    return std::move(v); // ?!  (don't do that)
}
```

std::move() Overdose

```cpp
std::vector<int> readInput() {
    std::vector<int> v;

    // ...

    return std::move(v); // ?!  (don't do that)
}
```

RVO	Return Value Optimization
NRVO	Named Return Value Optimization

std::move() Overdose

```
std::vector<int> readInput() {
    std::vector<int> v;

    // ...

    return std::move(v); // ?!  (don't do that)
}
```

RVO	Return Value Optimization
NRVO	Named Return Value Optimization

Not optimized? The compiler has to treat it as if `std::move()` was applied (C++14, 12.8/32).

OK... But What About This?

```
std::tuple<std::string, std::string> readInput() {
    std::pair<std::string, std::string> p;

    // ...

    return std::move(p); // OK (types are different)
}
```

Returning References

What is wrong about this code?

```
std::string&& readInput() { // ?! (don't do that)
    std::string input;
    // ...
    return std::move(input);
}
```

Returning References

What is wrong about this code?

```
std::string&& readInput() { // ?! (don't do that)
    std::string input;
    // ...
    return std::move(input);
}
```

You wouldn't do this in C++98, would you?

```
std::string& readInput() {
    std::string input;

    // warning: reference to local var returned
    return input;
}
```

std::move() Does Not Imply Movement

```
class Person {
public:
// ...

    void setName(const std::string n) {
        name = std::move(n); // Copies n!
    }

// ...

private:
    std::string name;
};
```

`std::move()` Does Not Imply Movement

```cpp
class Person {
public:
// ...

    void setName(const std::string n) {
        name = std::move(n); // Copies n!
    }

// ...

private:
    std::string name;
};
```

Note: “Movement” of legacy types (backward compatibility).

Towards the Need To Define Move Operations

What special members are there in C++?

Towards the Need To Define Move Operations

What special members are there in C++?

1. default constructor	`X();`	
2. destructor	`~X();`	
3. copy constructor	`X(const X&);`	
4. copy assignment	`X& operator=(const X&);`	
5. move constructor	`X(X&&);`	`// C++11`
6. move assignment	`X& operator=(X&&);`	`// C++11`

Do I Need To Define Move Operations?

When are move operations implicitly provided?

Do I Need To Define Move Operations?

When are move operations implicitly provided?

- No copy operations are declared in the class.
- No move operations are declared in the class.
- No destructor is declared in the class.

Do I Need To Define Move Operations?

When are move operations implicitly provided?

- No copy operations are declared in the class.
- No move operations are declared in the class.
- No destructor is declared in the class.

What do the implicitly provided move operations do?

Do I Need To Define Move Operations?

When are move operations implicitly provided?

- No copy operations are declared in the class.
- No move operations are declared in the class.
- No destructor is declared in the class.

What do the implicitly provided move operations do?

- Perform member-wise move of object's bases and members.

Do I Need To Define Move Operations?

When are move operations implicitly provided?

- No copy operations are declared in the class.
- No move operations are declared in the class.
- No destructor is declared in the class.

What do the implicitly provided move operations do?

- Perform member-wise move of object's bases and members.
- The move assignment does not include the `if (this != &other)` check.

Can I Use = `default`?

Can I write this?

```cpp
class A {
public:
    ~A(); // Disables implicit gen of move ops.

    A(A&&) = default;
    A& operator=(A&&) = default;

// ...
};
```

Can I Use `= default`?

Can I write this?

```
class A {
public:
    ~A(); // Disables implicit gen of move ops.

    A(A&&) = default;
    A& operator=(A&&) = default;

// ...
};
```

Yes*.

* If the default implementation is good enough for you.

Using Objects After Move

```
std::vector<int> v;

// ...

devour(std::move(v));
// What can I now do with v?
```

Using Objects After Move

```
std::vector<int> v;

// ...

devour(std::move(v));
// What can I now do with v?
```

From C++14, 17.6.5.15:

> *Unless otherwise specified, (...) moved-from objects shall be placed in a valid but unspecified state.*

Using Objects After Move

```cpp
std::vector<int> v;

// ...

devour(std::move(v));
// What can I now do with v?
```

From C++14, 17.6.5.15:

> *Unless otherwise specified, (...) moved-from objects shall be placed in a valid but unspecified state.*

✓	✗
`v.empty()`	`v[0]`
`v = other`	`v.pop_back()`

Rvalue References In the Standard Library

`std::vector::vector()`

```
vector(vector&& other);                    // C++11
```

Rvalue References In the Standard Library

`std::vector::vector()`

```
vector(vector&& other);                 // C++11
```

`std::vector::push_back()`

```
void push_back(const T& value);
void push_back(T&& value);              // C++11
```

Rvalue References In the Standard Library

std::vector::vector()

```
vector(vector&& other);                  // C++11
```

std::vector::push_back()

```
void push_back(const T& value);
void push_back(T&& value);               // C++11
```

Example:

```
std::vector<std::string> v;

std::string x("Live long and prosper.");
v.push_back(x);                    // via const T&
v.push_back(getSomeString()); // via T&&
```

Perfect Forwarding

Perfect Forwarding In a Nutshell

```
void f(X& p);   // A
void f(X&& p); // B

template <typename T>          // \
void wrapper(T&& p) {          //  \
    // Do some stuff.          //   Magic (for now).
    f(std::forward<T>(p));     //  /
}                              // /

X y;
wrapper(y);   // calls f(X& p)
wrapper(X()); // calls f(X&& p)
```

Perfect Forwarding In a Nutshell

```cpp
void f(X& p);  // A
void f(X&& p); // B

template <typename T>         // \
void wrapper(T&& p) {         //  \
    // Do some stuff.         //   Magic (for now).
    f(std::forward<T>(p));    //  /
}                             // /

X y;
wrapper(y);   // calls f(X& p)
wrapper(X()); // calls f(X&& p)
```

Notes:

- `std::forward()` does not forward anything.
- Perfect forwarding is imperfect.

The Double Life of `type&&`

```cpp
void f(int&& a) { /* ... */ }

template <typename T>
void g(T&& a) { /* ... */ }
```

The Double Life of `type&&`

```cpp
void f(int&& a) { /* ... */ }

template <typename T>
void g(T&& a) { /* ... */ }
```

```cpp
f(1); // OK
g(1); // OK
```

The Double Life of `type&&`

```
void f(int&& a) { /* ... */ }

template <typename T>
void g(T&& a) { /* ... */ }

f(1); // OK
g(1); // OK

int i = 1;
f(i); // error: cannot bind int lvalue to int&&
g(i); // OK (huh?)
```

The Double Life of `type&&`

```
void f(int&& a) { /* ... */ }

template <typename T>
void g(T&& a) { /* ... */ }

f(1); // OK
g(1); // OK

int i = 1;
f(i); // error: cannot bind int lvalue to int&&
g(i); // OK (huh?)

int&& i = 1;  // OK
auto&& j = 1; // OK
```

The Double Life of `type&&`

```
void f(int&& a) { /* ... */ }

template <typename T>
void g(T&& a) { /* ... */ }

f(1); // OK
g(1); // OK

int i = 1;
f(i); // error: cannot bind int lvalue to int&&
g(i); // OK (huh?)

int&& i = 1;  // OK
auto&& j = 1; // OK

int a = 1;
int&& k = a;  // error: cannot bind ...
auto&& l = a; // OK (huh?)
```

The ~~Lie~~ Abstraction

If a variable or parameter has declared type

`T&&`

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

The ~~Lie~~ Abstraction

If a variable or parameter has declared type

```
T&&
```

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

- Rvalue reference when initialized with rvalue.

The ~~Lie~~ Abstraction

If a variable or parameter has declared type

`T&&`

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

- Rvalue reference when initialized with rvalue.
- Lvalue reference when initialized with lvalue.

The ~~Lie~~ Abstraction

If a variable or parameter has declared type

`T&&`

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

- Rvalue reference when initialized with rvalue.
- Lvalue reference when initialized with lvalue.

It binds to everything.

The ~~Lie~~ Abstraction (Continued)

If a variable or parameter has declared type

```
T&&
```

for some deduced type `T`, it is a *universal* (or *forwarding*) reference.

The ~~Lie~~ Abstraction (Continued)

If a variable or parameter has declared type

T&&

for some deduced type `T`, it is a *universal* (or *forwarding*) reference.

```cpp
template <typename T>
void f(T&& p);         // Universal reference.

template <typename T>
void g(const T&& p); // Not universal reference.
```

The ~~Lie~~ Abstraction (Continued)

If a variable or parameter has declared type

`T&&`

for some deduced type `T`, it is a *universal* (or *forwarding*) reference.

```
template <typename T>
void f(T&& p);         // Universal reference.

template <typename T>
void g(const T&& p); // Not universal reference.
```

Name and whitespace do not matter:

```
template <typename K>
void f(  K    &&   p  ); // Universal reference.
```

The ~~Lie~~ Abstraction (Continued)

If a variable or parameter has declared type

```
T&&
```

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

The ~~Lie~~ Abstraction (Continued)

If a variable or parameter has declared type

T&&

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

```
template <typename T>
void f(T&& p); // Universal reference.

using T = int;
void h(T&& p); // Not universal reference.
```

The ~~Lie~~ Abstraction (Continued)

If a variable or parameter has declared type

`T&&`

for some **deduced type `T`**, it is a *universal* (or *forwarding*) reference.

```cpp
template <typename T>
void f(T&& p); // Universal reference.

using T = int;
void h(T&& p); // Not universal reference.
```

```cpp
template <typename T, /* Allocator */>
class vector {
public:
    void push_back(T&& x); // Not universal ref.

// ...
};
```

Towards The Truth: Reference Collapsing

When a reference-to-reference appears during type deduction, the following rules apply:

$$\begin{array}{rrcr} \& & \& & \Rightarrow & \& \\ \& & \&\& & \Rightarrow & \& \\ \&\& & \& & \Rightarrow & \& \\ \&\& & \&\& & \Rightarrow & \&\& \end{array}$$

Towards The Truth: Reference Collapsing

When a reference-to-reference appears during type deduction, the following rules apply:

$$\begin{array}{rrcr} \& & \& & \Rightarrow & \& \\ \& & \&\& & \Rightarrow & \& \\ \&\& & \& & \Rightarrow & \& \\ \&\& & \&\& & \Rightarrow & \&\& \end{array}$$

Stephan T. Lavavej: “Lvalue references are infectious”.

Towards the Truth: Type Deduction

`T&&` references employ the following type-deduction rules:

```
template <typename T>
void f(T&& param);

int i;
f(i);                 // T is int&
f(std::move(i));      // T is int (not int&&!)
```

Towards the Truth: Type Deduction

`T&&` references employ the following type-deduction rules:

```
template <typename T>
void f(T&& param);

int i;
f(i);              // T is int&
f(std::move(i));   // T is int (not int&&!)

f(i);              // f(int& &&); => f(int&);
f(std::move(i));   // f(int&&);
```

The Truth (Yay!)

A universal (or forwarding) reference *is* actually an rvalue reference in a context where

1. type deduction distinguishes lvalues from rvalues, and
2. reference collapsing occurs.

What Does `std::forward()` Do, Anyway?

Our old magical friend:

```cpp
void f(X& p);
void f(X&& p);

template <typename T>
void wrapper(T&& p) {
    // Do some stuff.
    f(std::forward<T>(p));
}
```

What Does `std::forward()` Do, Anyway?

Our old magical friend:

```
void f(X& p);
void f(X&& p);

template <typename T>
void wrapper(T&& p) {
    // Do some stuff.
    f(std::forward<T>(p));
}
```

`std::forward<T>(p)` is simply a conditional cast:

- When `T` is an lvalue reference, `return p;`
- Else, `return std::move(p);`

What Does `std::forward()` Do, Anyway?

Our old magical friend:

```
void f(X& p);
void f(X&& p);

template <typename T>
void wrapper(T&& p) {
    // Do some stuff.
    f(std::forward<T>(p));
}
```

`std::forward<T>(p)` is simply a conditional cast:

- When `T` is an lvalue reference, `return p;`
- Else, `return std::move(p);`

Notes:

- Passing `<T>` is mandatory.

What Does `std::forward()` Do, Anyway?

Our old magical friend:

```
void f(X& p);
void f(X&& p);

template <typename T>
void wrapper(T&& p) {
    // Do some stuff.
    f(std::forward<T>(p));
}
```

`std::forward<T>(p)` is simply a conditional cast:

- When `T` is an lvalue reference, `return p;`
- Else, `return std::move(p);`

Notes:

- Passing `<T>` is mandatory.
- `std::forward<T>(p)` $\Leftrightarrow$ `static_cast<T&&>(p)`

Perfect Forwarding In the Standard Library

`std::vector::emplace_back()`

```cpp
template <typename... Args>
void emplace_back(Args&&... args); // C++11
```

Perfect Forwarding In the Standard Library

`std::vector::emplace_back()`

```cpp
template <typename... Args>
void emplace_back(Args&&... args); // C++11
```

Example:

```cpp
std::vector<std::string> v;

v.push_back("Hello kitty.");     // via temp
v.emplace_back("Hello kitty."); // no temp
```

References and Further Information

Scott Meyers
Effective Modern C++
O'Reilly Media, 2014, 336 pages

Thomas Becker
C++ Rvalue References Explained
http://thbecker.net/articles/rvalue_references/section_01.html

- Scott Meyers: Universal References in C++11 (C++ and Beyond'12)
  - https://www.youtube.com/watch?v=dkeErTEO28Y
- Stephan T. Lavavej: Don't Help the Compiler (GoingNative'13)
  - https://www.youtube.com/watch?v=AKtHxKJRwp4
- Scott Meyers: An Effective C++11/14 Sampler (GoingNative'13)
  - https://www.youtube.com/watch?v=BezbcQIuCsY