Geheimnisse der Move-Semantik

Nicolai M. Josuttis
IT-communication.com

3/16

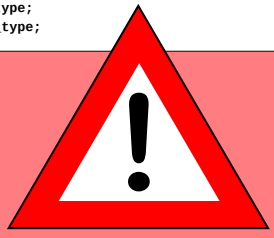
Nicolai M. Josuttis

- **Independent consultant**
  - continuously learning since 1962
- **Systems Architect, Technical Manager**
  - finance, manufacturing, automobile, telecommunication
- **Topics:**
  - C++
  - SOA (Service Oriented Architecture)
  - Technical Project Management
  - Privacy (contributor of Enigmail)

pair<> with C++98 and C++11

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;

    pair();
    pair(const

    template<c
      pair(con
};
```

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
```



This talk will:

- Raise your fears
- Sell stealing as a good thing
- Let you hate C++ even more

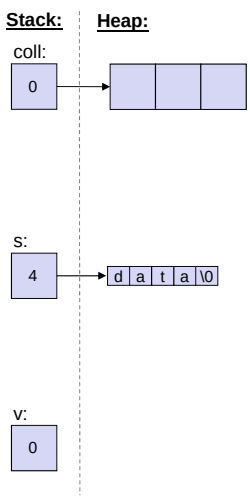
```
>::value &&
>::value);
```

```
  void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
};
```


Let's look at a Naive Function using Vectors and Strings in C++98/C++03

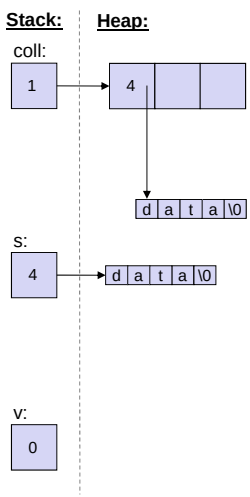
Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



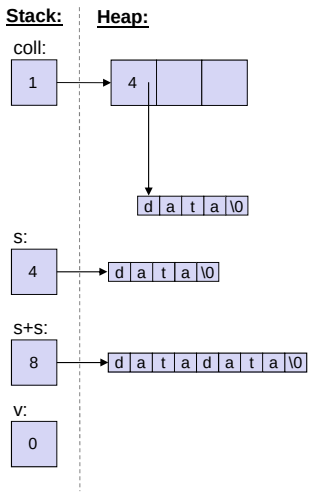
Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



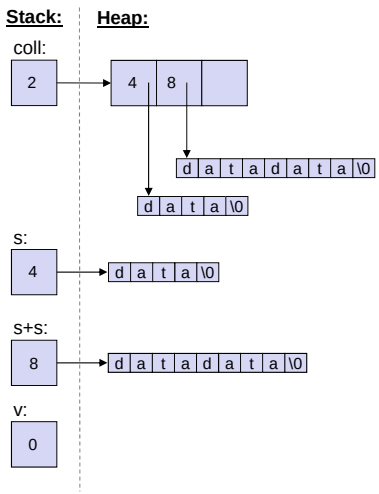
Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Copy Semantics of C++03

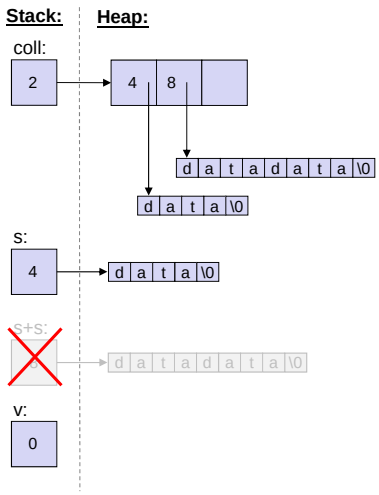
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Copy Semantics of C++03

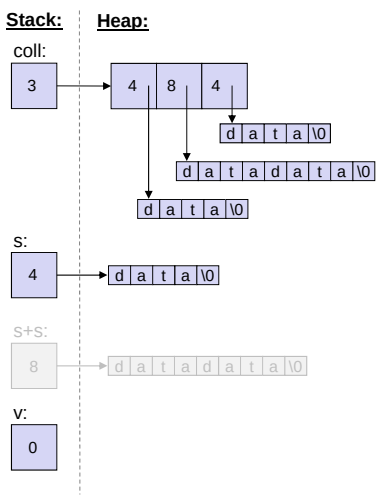
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

destruct temporary



Copy Semantics of C++03

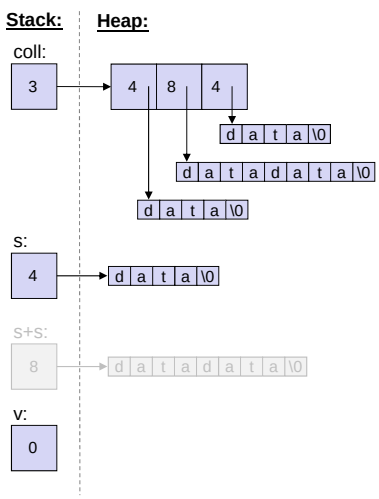
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

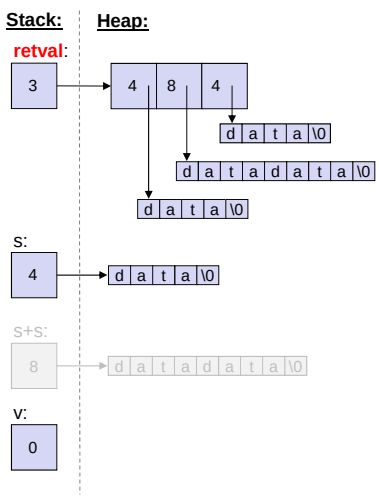
MAY deep copy coll



Copy Semantics of C++03

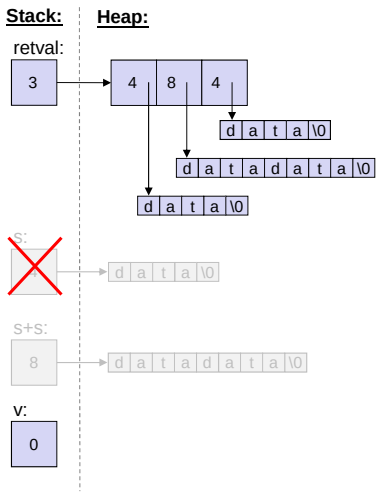
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

MAY deep copy coll



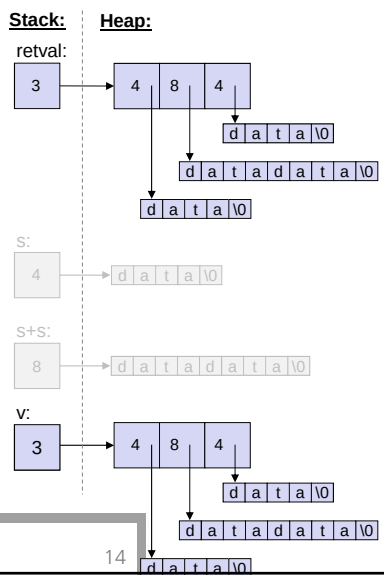
Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



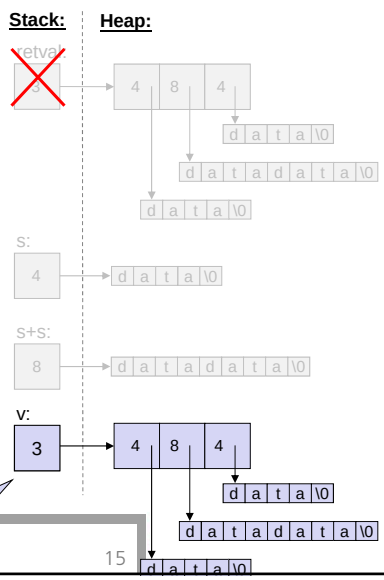
Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Copy Semantics of C++03

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(s);

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



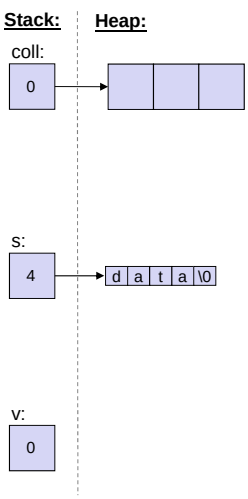
10 malloc/new
6 free/delete

Now let's make it better:

Use the Naive Function using Vectors and Strings in C++11

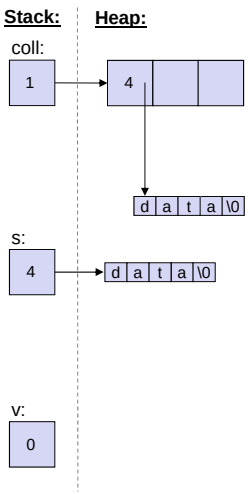
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



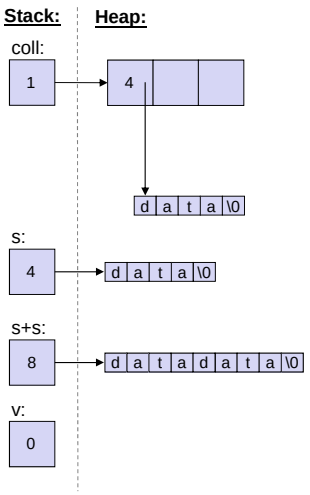
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



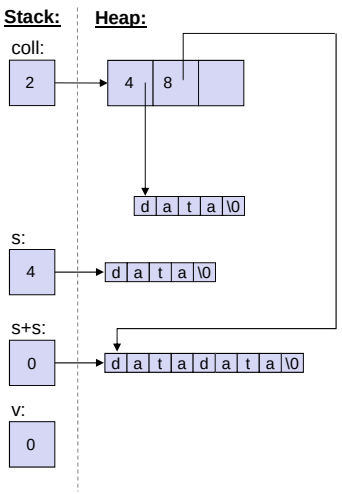
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



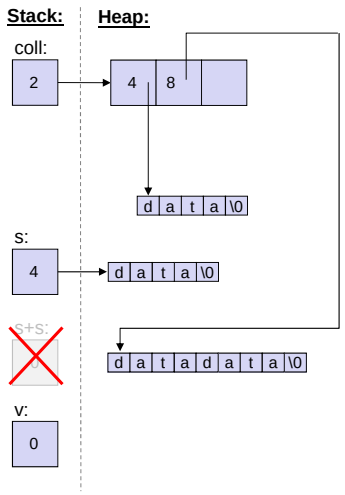
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



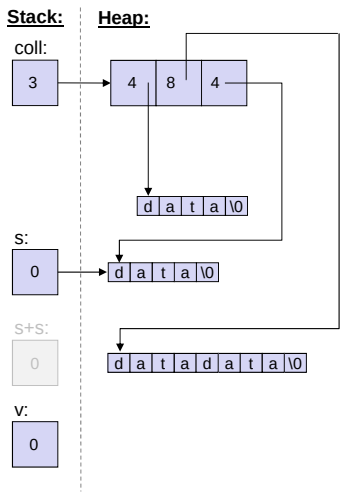
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::
                   destruct temporary

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Move Semantics of C++11

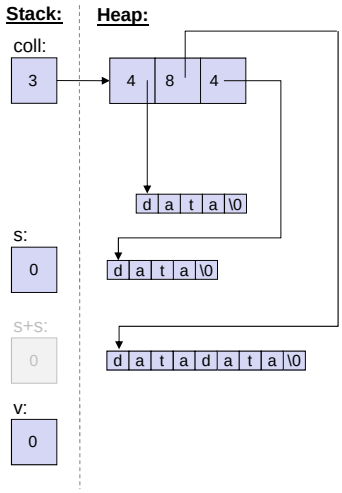
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

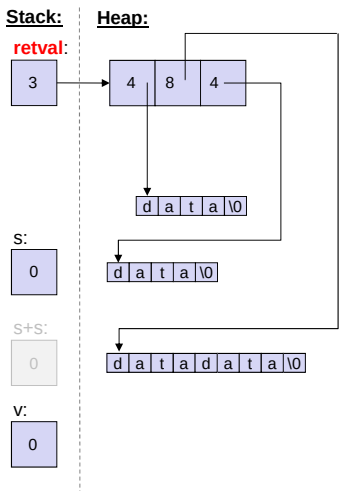
MAY move coll



Move Semantics of C++11

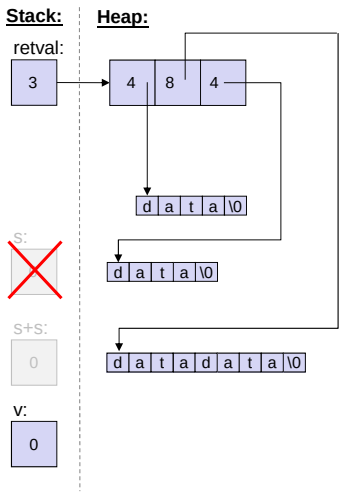
```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```

MAY move coll



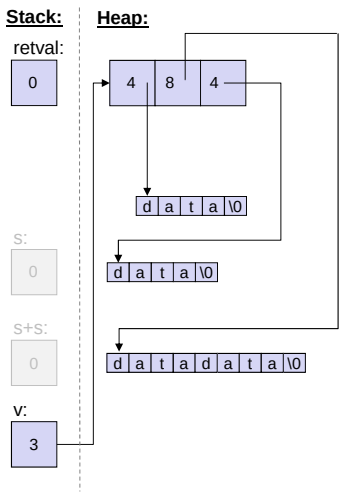
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



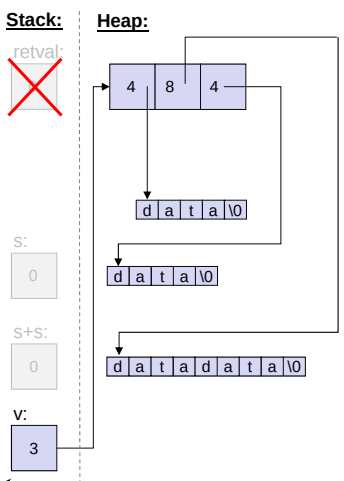
Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



Move Semantics of C++11

```
std::vector<std::string> createAndInsert()
{
    std::vector<std::string> coll;
    coll.reserve(3);
    std::string s("data");

    coll.push_back(s);

    coll.push_back(s+s);

    coll.push_back(std::move(s));

    return coll;
}

std::vector<std::string> v;
v = createAndInsert();
```



4 malloc/new
0 free/delete

So:

What changed with C++11?

Without Move Semantics

- **Containers have value semantics**
  - copy passed new elements into their containers
  - allows to pass rvalues (such as temporaries)
- **This leads to unnecessary copies with C++98/C++03**

```
template <typename T>
class vector {
  public:
    …
    // insert a copy of elem:
    void push_back (const T& elem);
    …
};
```

unnecessary copies in C++98 / C++03

```
std::vector<std::string> createAndInsert ()
{
    std::vector<std::string> coll;
    std::string s("data");    // create a string s
    …
    coll.push_back(s);        // insert a copy of s into coll
    …                         // s is used and modified afterwards

    coll.push_back(s+s);      // insert copy of temporary rvalue

    coll.push_back(s);        // insert copy of s again
                              // but s is no longer used here
    return coll;              // may copy coll
}
```


With Move Semantics

- **With rvalue references you can provide move semantics**
- **RValue references represent modifiable object where the value is no longer needed so that you can *steal* their content**

```
template <typename T>
class vector {
  public:
    …
    // insert a copy of elem:
    void push_back (const T& elem);
    …
    // insert elem with its content moved:
    void push_back (T&& elem);
    …
};
```

declares rvalue reference

```
#include <utility>        // declares std::move()

std::vector<std::string> createAndInsert ()
{
    std::vector<std::string> coll;
    std::string s("data");    // create a string s
    …
    coll.push_back(s);        // insert a copy of s into coll
    …                         // s is used and modified afterwards

    coll.push_back(s+s);      // move temporary into coll

    coll.push_back(std::move(s));
                              // move s into coll
                              // OK, because s is no longer used
    return coll;              // may move coll
}
```

With Move Semantics

- **To support move semantics for non-trivial types you should:**
  - provide a **move constructor**
  - provide a **move assignment operator**

  where the move version is optimized to
  - steal contents from the passed object
  - and set the assigned object in a valid but undefined (or initial) state

```
class string {
  private:
    int len;       // current number of characters
    char* elems;   // array of characters

  public:
    // create a full copy of s:
    string (const string& s)
     : len(s.len) {
        elems = new char[len+1]; // new memory
        memcpy(elems,s.elems,len+1);
    }

  public:
    // create a copy of s with its content moved:
    string (string&& s)
     : len(s.len),
       elems(s.elems) {     // copy pointer to memory
        s.elems = nullptr;  // otherwise destructor of s
                            // frees stolen memory
        s.len = 0;
    }
    …
};
```


With Move Semantics

- **With rvalue references you can provide move semantics**
- **RValue references represent modifiable object where the value is no longer needed so that you can *steal* their content**

```
template <typename T>
class vector {
  public:
    …
    // insert a copy of elem:
    void push_back (const T& elem);
    …
    // insert elem with its content moved:
    void push_back (T&& elem);
    …
};
```

don't use const here:

declares rvalue reference

```
#include <utility>        // declares std::move()

std::vector<std::string> createAndInsert ()
{
    std::vector<std::string> coll;
    std::string s("data");    // create a string s
    …
    coll.push_back(s);    // insert a copy of s into coll
    …                     // s is used and modified afterwards

    coll.push_back(s+s); // move temporary into coll

    coll.push_back(std::move(s));
                          // move s into coll
                          // OK, because s is no longer used
    return coll;          // may move coll
}
```

So:

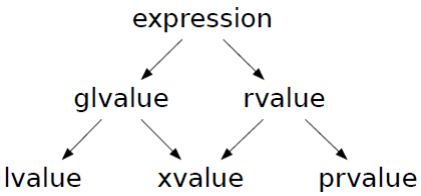
What are the consequences?

Value Categories until C++03

- **CPL, BCPL, B, K&R C (C without const):**
  - **LValue**
    - Value suitable for the **left-hand-side** of assignments
  - **RValue**
    - Value only suitable for the **right-hand-side** of assignments
- **C++98/C++03:**
  - **LValue:**
    - **Localizable** value (object with an identifiable memory location)
    - Object or returned reference
  - **RValue:**
    - Non-LValue (object with no identifiable memory location)
      - Every expression is either an lvalue or an rvalue
    - Kind of **Read-only** value
    - Temporary, that is not returned by reference
      - If of class type, it **can be modified** by non-const member functions

Value Categories since C++11

N3055:

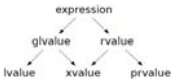


Every expression is one of:

- **LValue: Localizable value**
  - Variable, data member, function, string literal, returned lvalue reference
  - Can be on the left side of an assignment only if it's modifiable
- **PRValue: Pure RValue** (former RValue)
  - All Literals except string literals (**`42`**, **`true`**, **`nullptr`**,...), **`this`**, lambda, returned non-reference, result of constructor call (**`T(...)`**)
- **XValue: eXpiring value**
  - Returned rvalue reference (e.g. by **std::move()**), cast to rvalue reference

General categories: GLValue: Generalized LValue, RValue: generalized RValue

Overloading Rules with RValue References



modifiable lvalue

non-modifiable lvalue

Test program:

```
class X;

X v;
const X c;

foo(v);
foo(c);
foo(X());
```

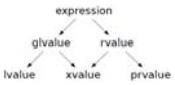
Declarations:

Calls:	`foo(X&)`	`foo(const X&)`
`foo(v)`	1	2
`foo(c)`	*error*	1
`foo(X())`	*error*	1

- **With prio 1 call foo(X&)**
- **With prio 2 call foo(const X&)**

rvalue => prvalue

Overloading Rules with RValue References



modifiable lvalue

non-modifiable lvalue

Test program:

```
class X;

X v;
const X c;

foo(v);
foo(c);
foo(X());
foo(std::move(v));
```

Declarations:

Calls:	foo(X&)	foo(const X&)	foo(X&&)
foo(v)	1	2	error
foo(c)	error	1	error
foo(X())	error	2	1
foo(move(v))	error	2	1

prvalue

xvalue

- can only pass rvalues to rvalue references
- *might* move argument

Overloading Rules with by-reference and by-value

Test program:

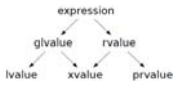
```
class X;

X v;
const X c;

foo(v);
foo(c);
foo(X());
foo(std::move(v));
```

	by reference			by value
Calls:	foo(X&)	foo(const X&)	foo(X&&)	foo(X)
foo(v)	1	2	error	*)
foo(c)	error	1	error	*)
foo(X())	error	2	1	*)
foo(move(v))	error	2	1	*)

*) Note:
- A call of foo() overloaded by-value and by-reference is an ambiguity error

- might move arg (whether it moves depends on the implementation)
- will move arg (for temporaries, move might even be optimized away)

Type versus Value Category




```
void passRValueRef (std::string&&);   // declaration
```

can only pass rvalues

```
std::string s;                  // s is lvalue
passRValueRef(s);               // ERROR: cannot bind lvalue to rvalue reference
passRValueRef("hello");         // OK: can pass prvalue (string literal (lvalue) converted to std::string)
passRValueRef(std::move(s));    // OK: can pass xvalue
```

```
void passRValueRef (std::string&& s)
{
  ...
  passRValueRef(s);             // ERROR: cannot bind lvalue to rvalue reference
  passRValueRef(std::move(s));  // OK: can pass xvalue
}
```

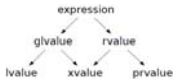
s has:
- Type `std::string&&`
- Value category lvalue

- **An object (parameter) declared with type rvalue reference is an lvalue (as any variable) when it is used**

Thus:

- **Move semantics is not automatically passed through**

`decltype` and Value Categories



- **`decltype` can be used to check both type and value category:**
  - **`decltype(e)`** yields type of an **entity** e
  - **`decltype((e))`** yields a type for value category of **expression** e:
    - for prvalues: ***type***
    - for lvalues: ***type&***
    - for xvalues: ***type&&***

s has:
- Type `std::string&&`
- Value category lvalue

```
std::string&& passRValueRef (std::string&& s)
{
    is_lvalue_reference<decltype(s)>::value          // false
    is_rvalue_reference<decltype(s)>::value          // true
    is_same<decltype(s),std::string&>::value         // false
    is_same<decltype(s),std::string&&>::value        // true

    is_lvalue_reference<decltype((s))>::value        // true
    is_rvalue_reference<decltype((s))>::value        // false
    is_same<decltype((s)),std::string&>::value       // true
    is_same<decltype((s)),std::string&&>::value      // false
}
```

Type of entity s: `std::string&&`

Type of expression s: `std::string&` because s is an lvalue

Forwarding Move Semantics

- **You have to forward move semantics explicitly:**

```
class X;

void g (X&);          // for variable values
void g (const X&);    // for constant values
void g (X&&);         // for values that are no longer used (move semantics)


void f (X& t) {
    g(t);             // t is non const lvalue => calls g(X&)
}
void f (const X& t) {
    g(t);             // t is const lvalue     => calls g(const X&)
}
void f (X&& t) {
    g(std::move(t)); // t is non const lvalue => needs std::move() to call g(X&&)
}                    // - When move semantics would always be passed,
                     //   calling g(t) twice would be a problem

X v;
const X c;
f(v);             // calls f(X&)       => calls g(X&)
f(c);             // calls f(const X&) => calls g(const X&)
f(X());           // calls f(X&&)      => calls g(X&&)
f(std::move(v)); // calls f(X&&)      => calls g(X&&)
```


Perfect Forwarding

- **Special semantics for && with template types**
  - You can pass temporaries and constants and variables and the template type knows what they are
  - You can use **`std::forward<>()`** to keep this semantics
- **"*Universal Reference*"** (term by Scott Meyers)

```
void g (X&);          // for variable values
void g (const X&);    // for constant values
void g (X&&);         // for values that are no longer used (move semantics)

template <typename T>
void f (T&& t)
{
    g(std::forward<T>(t)); // forwards move semantics
}                          // (without forward<>, only calls g(const X&) or g(X&))

X v;
const X c;

f(v);
f(c);
f(X());
f(std::move(v));
```

The Semantic Difference of &&

- **Declarations with && have different semantics between non-templates and templates**

```
typedef std::vector<std::string> Coll;


void foo (Coll&& v)        // v is rvalue reference (always non-const)
{
   auto pos = v.begin();   // always yields Coll::iterator
   …
}


const Coll c;
Coll v;

foo(c);              // ERROR: needs foo(Coll) or foo(const Coll&)
foo(v);              // ERROR: needs foo(Coll) or foo(const Coll&) or foo(Coll&)
foo(Coll());         // OK, v is non-const rvalue reference
foo(std::move(v));   // OK, v is non-const rvalue reference
```


The Semantic Difference of &&

- **Declarations with && have different semantics between non-templates and templates**

```
typedef std::vector<std::string> Coll;

template <typename Coll>
void foo (Coll&& v)        // v is universal reference (might be non-const or const)
{
   auto pos = v.begin();   // might yield Coll::iterator or Coll::const_iterator
   …
}


const Coll c;
Coll v;

foo(c);              // OK, v is const
foo(v);              // OK, v is non-const
foo(Coll());         // OK, v is non-const rvalue reference
foo(std::move(v));   // OK, v is non-const rvalue reference
```

Universal Reference with auto&&

auto&&

- **Universal reference outside templates**
- **Only generic type that can refer to**
  - temporaries and
  - constants and
  - variables

  while not being const, if it refers to a non-const

```
int i=42;
int& r=i;

auto a = r;                                    // non-const lvalue from r
  std::is_reference<decltype(a)>::value        // false
  std::is_same<decltype(a),int>::value         // true
  std::is_same<decltype(a),int&>::value        // false

auto&& aa = r;                                 // same type as r
  std::is_reference<decltype(aa)>::value       // true
  std::is_same<decltype(aa),int>::value        // false
  std::is_same<decltype(aa),int&>::value       // true
```


Range-Based for Loops

- **The expression:**

```
for ( decl : coll ) {
    statement
}
```

```
// print all elements of coll (using a range-based for loop):
for (const auto& elem : coll) {
    std::cout << elem << std::endl;
}
```

- **is equivalent to:**

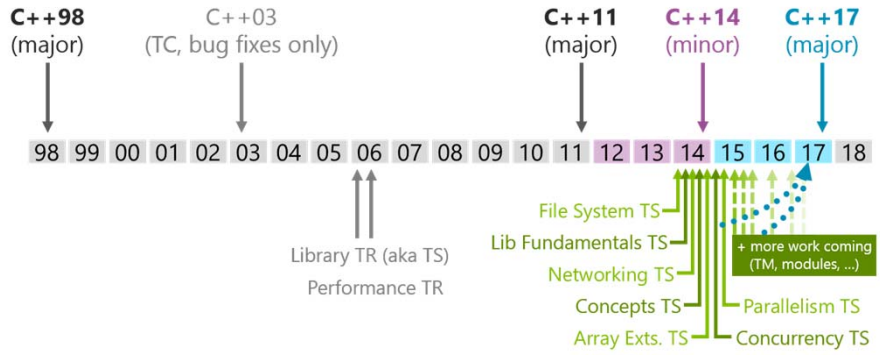
```
{
    for (auto&& _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos ) {
        decl = *_pos;
        statement
    }
}
```

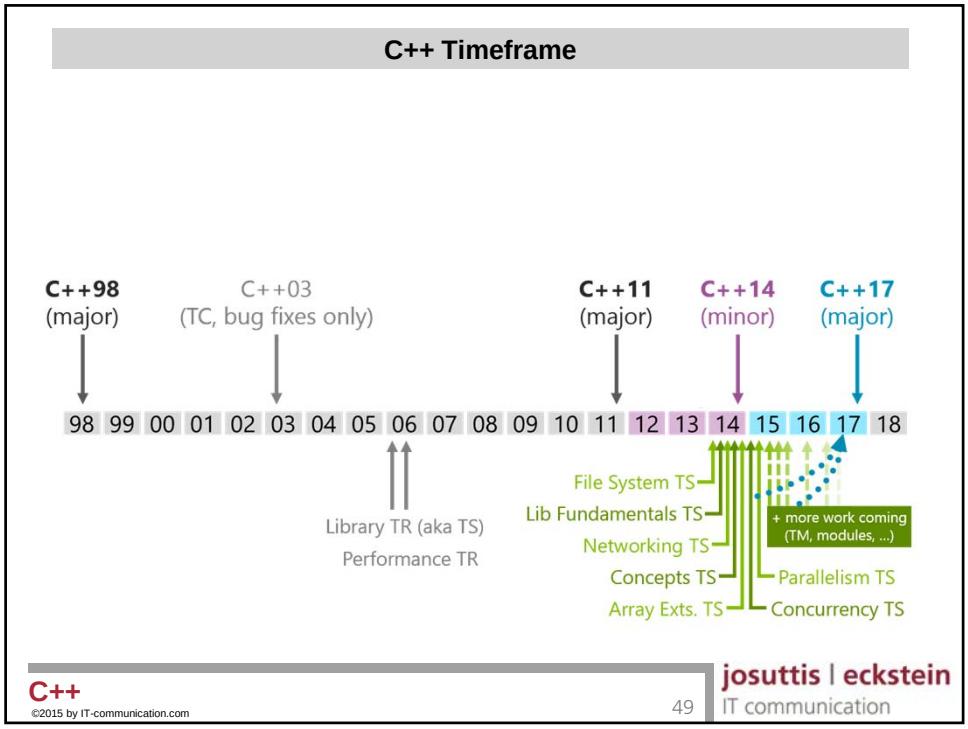
```
// print all elements of coll (using a range-based for loop):
{
 for (auto&& _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos ) {
    const auto& elem = *_pos;
    std::cout << elem << std::endl;
 }
}
```

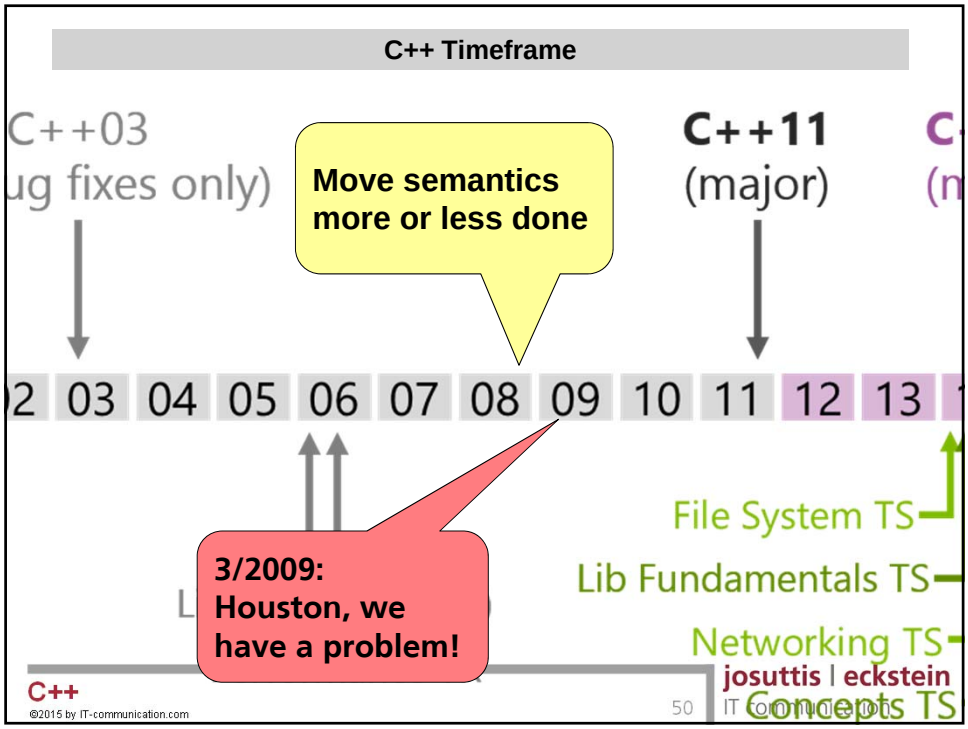
So:

What are the nasty consequences?

C++ Timeframe







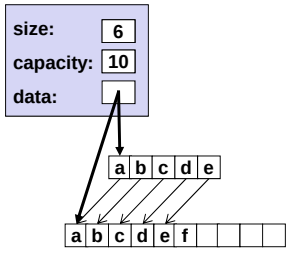
Exception Safety Guarantees in the Standard Library

- **Basic exception guarantee**
  - The invariants of the component are preserved and no resources are leaked
  - Always given throughout the Standard Library
- **Strong exception guarantee**
  - "Transaction safety" / "Commit-or-Rollback behavior"
  - An operation either completes successfully or throws an exception, leaving the program state exactly as it was before the operation started
  - Since C++98, the C++ Standard gives the strong exception guarantee for **`push_back()`** and **`push_front()`**:
    - "If an exception is thrown by a **`push_back()`** or **`push_front()`** function, that function has no effects."

Exception Safety Guarantee for Vector's push_back()

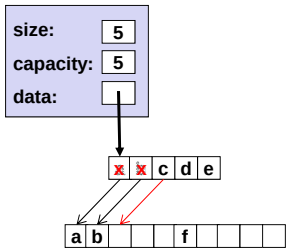
- **In C++98/C++03 the guarantee is possible because:**
  - Reallocation is done by the following steps:
    - allocate new memory
    - assign new value
    - copy old elements (element by element)

    ------ point of no rollback ------
    - assign new memory to internal pointer
    - delete old elements and free old memory
    - update size and capacity



Exception Safety Guarantee for Vector's push_back()

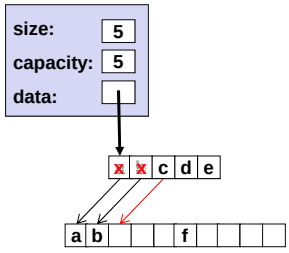
- **With move semantics the strong guarantee is no longer possible**
  - Moving elements might throw but is not a reversible step
- **But, although it might not often be used, we can't silently break this strong exception guarantee**
- **And replacing `push_back()` by something new is not an option**
- **So:**
  We can only move when it's safe



Exception Safety Guarantee for Vector's push_back()

- **Solution:**
  Vectors use move operations for elements only if these operations can't throw
- **For this:**
  - It's worth to know whether move operations of elements can't throw
  - But for template elements (e.g. **`pair<>`**) this might depend on the template args

=> We need a way to specify a conditional guarantee not to throw



Keyword noexcept

- **New keyword that replaces exception specifications**
  - Exception specifications are deprecated since C++11
- **Because throw specifications face the following problems:**
  - C++ exception specifications are checked at runtime rather than at compile time
    - runtime overhead (because it requires stack unwinding)
    - no guarantee that all exceptions have been handled (=> `unexpected()`)
  - hard to specify in generic code
- **New approach with `noexcept`:**
  - noexcept specifies that a function does not throw any exception
    - no runtime overhead
    - if an uncaught exception occurs, `terminate()` gets called

Keyword noexcept

- **Keyword `noexcept` can be used**
  - as declarator
    - to specify whether/when a function guarantees not to throw
  - as operator
    - which yields **`true`** if an expression guarantees not throw an exception
- **Using both, you can specify a condition under which functions do not throw**
  - **`noexcept`** is a shortcut for **`noexcept(true)`**

```
template <…> class vector {
  public:
    iterator begin() noexcept;
    …
};

void swap (Type& x, Type& y) noexcept( noexcept(x.swap(y)) )
{
    x.swap(y);
}
```

noexcept declaration:
`swap()` does not throw if condition yields `true`

operator noexcept(…):
yields `true` if `x.swap(y)` can't throw

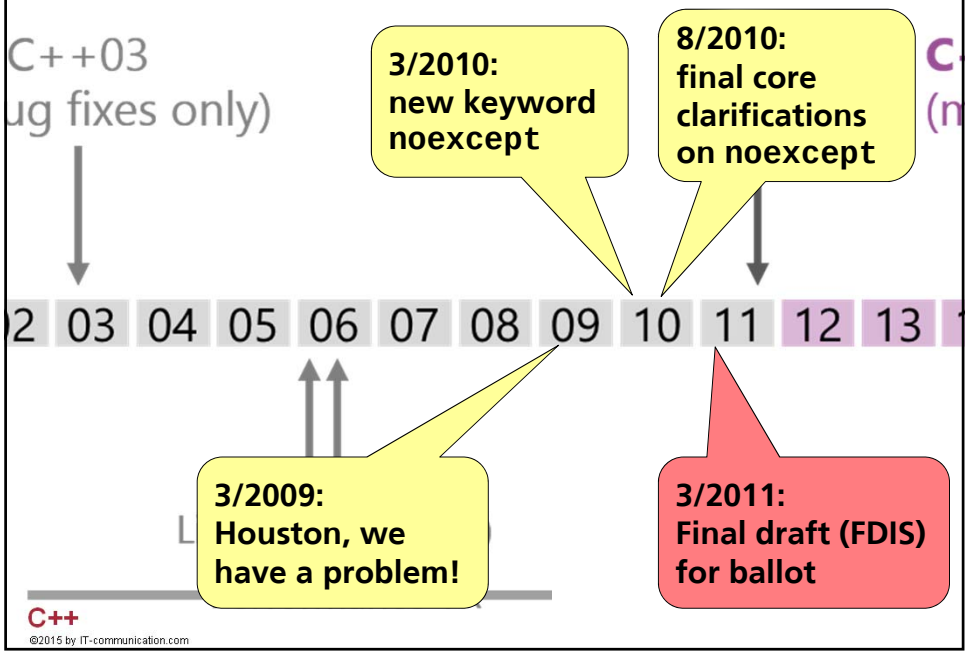
Explicit noexcept of `std::pair<>`

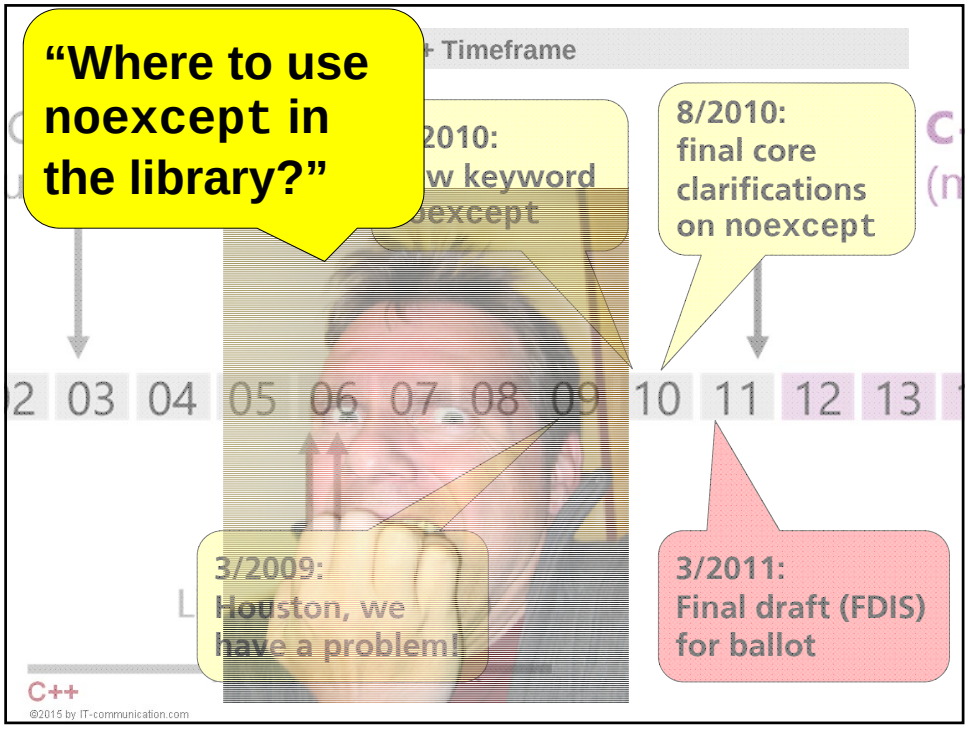
- **Class std::pair<> should be explicitly defined as follows:**

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;

  constexpr pair() noexcept( is_nothrow_default_constructible<T1>::value &&
                             is_nothrow_default_constructible<T2>::value );
  pair(const T1& x, const T2& y) noexcept( is_nothrow_copy_constructible<T1>::value &&
                                           is_nothrow_copy_constructible<T2>::value );

  pair(const pair&) noexcept( is_nothrow_copy_constructible<T1>::value &&
                              is_nothrow_copy_constructible<T2>::value );
  pair(pair&&)      noexcept( is_nothrow_move_constructible<T1>::value &&
                              is_nothrow_move_constructible<T2>::value );
  …
  pair& operator= (const pair& p) noexcept( is_nothrow_copy_assignable<T1>::value &&
                                            is_nothrow_copy_assignable<T2>::value );
  pair& operator= (pair&& p)      noexcept( is_nothrow_move_assignable<T1>::value &&
                                            is_nothrow_move_assignable<T2>::value );
  …
  void swap(pair& p) noexcept( noexcept(swap(first,p.first)) &&
                               noexcept(swap(second,p.second)) );
};
```

Uses Type Traits
- Utilities for programming with types

C++ Timeframe





Document: N3248=11-0018
Date: 2011-02-28
Authors: Alisdair Meredith (ameredith1@bloomberg.net)
John Lakos (jlakos@bloomberg.net)

Abstract

The `noexcept` language facility was added at the Pittsburg meeting immediately prior to the FCD to solve some very specific problems with move semantics. This new facility also addresses a long-standing desire for many libraries to flag which functions can and cannot throw exceptions in general, opening up optimization opportunities.

The Library Working Group is now looking for a metric to decide when it is appropriate to apply the `noexcept` facility, and when to be conservative and say nothing. After spending some time analyzing the problem, the authors have concluded that the current specification for `noexcept` greatly restricts the number of places it can be used safely in a library specification such as (but not limited to) the standard library.

In this paper we propose a strict set of criteria to test before the Library Working Group should mark a function as `noexcept`. We further propose either lifting the requirement that throwing exceptions from a `noexcept` function must terminate a program (in favor of general undefined behavior), or adopting additional criteria that severely restrict the use of `noexcept` in the standard library.

Conservative use of `noexcept` in the Library

Document: N3279=11-0049
Date: 2011-03-25
Authors: Alisdair Meredith (ameredith1@bloomberg.net)
John Lakos (jlakos@bloomberg.net)

Motivation

The paper N3248 raised a number of concerns with widespread adoption of `noexcept` exception specifications in the standard library specification, preferring their use be left as a library vendor quality-of-implementation feature until we have more experience.

Further discussion at the Madrid meeting, 2011, showed that while the committee shared some of these concerns, it also wanted to promote the use of such exception specifications where they provided a benefit, and did no harm.

After some discussion, the following set of guidelines for appropriate use of `noexcept` in a library specification were adopted. The rest of this paper applies these guidelines to the working paper N3242.

`noexcept` Policy according to N3279

- Each library function
  - that the LWG [library working group] agree **cannot throw**,
  - and having a "**wide contract**"
    [i.e. does **not** specify **undefined behavior** due to a precondition],

  should be marked as **unconditionally `noexcept`**.
- If a library **swap** function, **move** constructor, or **move** assignment operator is "conditionally wide" (i.e. can be proven not to throw by applying the [conditional] noexcept operator)
  then it should be marked as **conditionally `noexcept`**.
  No other function should use a conditional **`noexcept`** specification.
- No library destructor should throw. It shall use the implicitly supplied (non-throwing) exception specification.
- Library functions designed for compatibility with C code ... may be marked as unconditionally **`noexcept`**.

C++
©2015 by IT-communication.com
62
josuttis | eckstein
IT communication

noexcept vs. "Throws: Nothing"

21.4.4 basic_string capacity [string.capacity]

```
size_type size() const noexcept;
```

Returns: A count of the number of char-like objects currently in the string.

Complexity: Constant time.

...

```
bool empty() const noexcept;
```

Returns: `size() == 0`.

21.4.5 basic_string element access [string.access]

```
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
```

Requires: `pos <= size()`.

Returns: `*(begin() + pos)` if `pos < size()`. Otherwise, returns a reference to an object of type `charT` with value `charT()`, where modifying the object leads to undefined behavior.

Throws: Nothing.

Complexity: Constant time.

noexcept of std::pair<>

- **Class std::pair<> is defined as follows:**

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;

  constexpr pair();                 // missing noexcept declaration
  pair(const T1& x, const T2& y);   // missing noexcept declaration

  pair(const pair&) = default;      // OK, default derives noexcept declaration from members
  pair(pair&&) = default;           // OK, default derives noexcept declaration from members
  …
  pair& operator= (const pair& p); // missing noexcept declaration
  pair& operator= (pair&& p)       noexcept( is_nothrow_move_assignable<T1>::value &&
                                             is_nothrow_move_assignable<T2>::value );
  …
  void swap(pair& p) noexcept( noexcept(swap(first,p.first)) &&
                               noexcept(swap(second,p.second)) );
};
```

noexcept Policy for the Standard Library

- **Standard containers don't define their move operations as noexcept yet in C++11/C++14**
  - some implementations do

For example:

```
template <class T, class Allocator = allocator<T> >
class vector {
 public:
  vector (vector&&);                 // no noexcept
  vector& operator= (vector&& x);    // no noexcept
  …
};
```

Note: Implementations are allowed to add noexcept (g++/clang do)

noexcept Policy for the Standard Library

- **For std::string, the C++11/C++14 standard library defines move operations with noexcept**
- **But:**
  - There was a request to remove that support
  - Unconditional **noexcept** is simply wrong

```
template<…>
class basic_string {
 public:
  …
  basic_string (basic_string&&) noexcept;              // move constructor
  basic_string& operator= (basic_string&&) noexcept; // move assignment
  …
};
```

Example with and without noexcept [c++std-lib-35804]

```
#include <vector>
#include <string>
#include <chrono>
#include <iostream>
using namespace std;
using namespace std::chrono;

class X
{
  private:
    string s;
  public:
    X()
     : s(100, 'a') {
    }

    X(const X& x) = default;

    X (X&& x) NOEXCEPT
     : s(move(x.s))
    {
    }
};
```

```
int main()
{
  // create vector of 1 million X (with string member)
  vector<X> v(1000000);
  cout << "cap.: " << v.capacity() << endl;

  // measure time to add 1 additional element
  auto t0 = high_resolution_clock::now();
  v.emplace_back();
  auto t1 = high_resolution_clock::now();

  auto d = duration_cast<milliseconds>(t1-t0);
  cout << d.count() << " ms\n";
}
```

clang++ -std=c++11 test.cpp -O3 -DNOEXCEPT="noexcept"

is 10 times faster than

clang++ -std=c++11 test.cpp -O3 -DNOEXCEPT=""

Example with and without noexcept [c++std-lib-35804]

```
#include <vector>
#include <string>
#include <chrono>
#include <iostream>
using namespace std;
using namespace std::chrono;

class X
{
  private:
    string s;
  public:
    X()
     : s(100, 'a') {
    }

    X(const X& x) = defau

};
```

```
int main()
{
  // create vector of 1 million X (with string member)
  vector<X> v(1000000);
  cout << "cap.: " << v.capacity() << endl;

  // measure time to add 1 additional element
  auto t0 = high_resolution_clock::now();
  v.emplace_back();
  auto t1 = high_resolution_clock::now();

  auto d = duration_cast<milliseconds>(t1-t0);
  cout << d.count() << " ms\n";
```

different machines!

Reallocation of # Elements		Without noexcept	With noexcept
clang++	1,000,000	228 - 239 ms	19 - 22 ms
g++49	1,000,000	15 – 31 ms	0 ms
g++49	10,000,000	234 – 249 ms	15 – 31 ms
VC++18	1,000,000	~150 ms	~15 ms

Discussion in Library Evolution Working Group

- **In Rapperswil 2014/06 we discussed this topic based on paper N4002 in LEWG and came to the following concluding vote:**
  - For **vectors** we want to have
    - noexcept default constructor
    - noexcept move constructor
    - conditional noexcept move assignment
    - conditional noexcept swap
  - For **strings** we want to have
    - noexcept default constructor
    - noexcept move constructor
    - conditional noexcept move assignment
    - conditional noexcept swap
  - For **all other containers** we want to have
    - conditional noexcept move assignment
    - conditional noexcept swap
    - **no** required noexcept for move constructor

Best performance guaranteed in all modes for our "almost fundamental data types"
- **vector**
- **string**

Final Changes in Library Working Group for C++17

In Urbana 2014/11 we agreed in LWG on N4258 (extract of 18 pages):

- In §**21.4 [basic.string]** in class std::basic_string

Modify (add)**:**

```
basic_string() noexcept : basic_string(Allocator()) { }

explicit basic_string(const Allocator& a) noexcept;
```

Unlike library issue 2319 proposed,
keep:

```
basic_string(basic_string&& str) noexcept;
```

Modify (add)**:**

```
basic_string& operator=(basic_string&& str)
noexcept(allocator_traits<Allocator>
           ::propagate_on_container_move_assignment::value
         || allocator_traits<Allocator>::is_always_equal::value);
```

similar changes for swap

Move Semantics Bonus Track:

Which Library Function changed with each and every C++ Version?

(i.e. with C++98, C++03, C++11, C++14)

make_pair() in C++98

- **`std::make_pair()` is a convenience function to create a `std::pair` of values without declaring their types**
  - The types of the pair are deduced from the passed arguments
- **In C++98, the parameters were declared as const references, which disabled decay**

```
namespace std {
  // implementation according to C++98:
  template <typename T1, typename T2>
  pair<T1,T2> make_pair (const T1& x,
                         const T2& y) // passed by reference, so that x and y don't decay
  {
    return pair<T1,T2>(x,y);
  }
}

int a[3];
std::make_pair (a, "hello") // returns a std::pair<int[3],const char[6]>
```

Decay for Templates

- **By rule, a template argument passed by value "decays"**
  - decay:
    type/function of arrays is converted to corresponding pointer type

```
template <typename T>
void printTypeByVal (T val);

template <typename T>
void printTypeByLRef (const T& val);



printTypeByVal  ("hello");    // template parameter T has type const char*
printTypeByLRef ("hello");    // template parameter T has type const char[6]
```


make_pair() in C++03

- **To enable decay, there was a fix in C++03 to pass the parameters by value**
- **See Library Issue 181**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-defects.html#181

```
namespace std {
  // implementation according to C++03:
  template <typename T1, typename T2>
  pair<T1,T2> make_pair (T1 x, T2 y)  // passed by value, so that x and y decay
  {
    return pair<T1,T2>(x,y);
  }
}

int a[3];
std::make_pair (a, "hello") // returns a std::pair<int*,const char*>
```

make_pair() in C++11

- **In C++11 we want to support move semantics**

```
namespace std {
  // NAIVE implementation according to C++11:
  template <typename T1, typename T2>
  pair<T1,T2> make_pair (T1&& x, T2&& y)  // use universal reference
  {
    return pair<T1,T2>(forward<T1>(x),forward<T2>(y));
  }
}

int a[3];
std::make_pair (a, "hello") // returns a std::pair<int[3],const char[6]> again
```


Decay for Templates

- **By rule, a template argument passed by value "decays"**
  - decay:
    type/function of arrays is converted to corresponding pointer type
- **Arrays passed by reference remain to be arrays**
  - In C++11, this also applies to rvalue-references

```
template <typename T>
void printTypeByVal (T val);

template <typename T>
void printTypeByLRef (const T& val);

template <typename T>
void printTypeByRRef (T&& val);

printTypeByVal  ("hello");   // template parameter T has type const char*
printTypeByLRef ("hello");   // template parameter T has type const char[6]
printTypeByRRef ("hello");   // template parameter T has type const char[6]
```

make_pair() in C++11

- In C++11, `std::make_pair()` supports move semantics, so that rvalue references have to be used
- So, the implicit decay has to get replaced by an explicit decay, which is provided as `std::decay<>`

```
namespace std {
  // implementation according to C++11:
  template <typename T1, typename T2>
  constexpr pair<typename decay<T1>::type,
                 typename decay<T2>::type>
  make_pair (T1&& x, T2&& y)  // passed by rvalue ref, so that x and y don't decay
  {
    return pair<typename decay<T1>::type,
                typename decay<T2>::type>(forward<T1>(x),
                                          forward<T2>(y));
  }
}

int a[3];
std::make_pair (a, "hello")  // returns a std::pair<int*,const char*>
```


make_pair() in C++14

- In C++14,
  `std::decay_t<T>`
  can be used as replacement of
  `typename std::decay<T>::type`

```
namespace std {
  // implementation according to C++14:
  template <typename T1, typename T2>
  constexpr pair<decay_t<T1>, decay_t<T2>>
  make_pair (T1&& x, T2&& y)
  {
    return pair<decay_t<T1>, decay_t<T2>>(forward<T1>(x), forward<T2>(y));
  }
}

int a[3];
std::make_pair (a, "hello")  // returns a std::pair<int*,const char*>
```

How-To for Move Semantics

- **As an application programmer:**
  - Use **`std::move()`** if you pass an object that is no longer used (and where copying might become expensive)
  - Use vectors as containers unless you have a good (measured) reason
- **As a class designer:**
  - If copying is expensive (and this is not caused by the members only), implement your own move constructor and/or move assignment operator
  - Declare move operations with a (conditional) **`noexcept`** to support move operations where strong exceptions guarantees are required
  - If you have to define one of the 5 special member functions
    - copy constructor, move constructor, copy assignment, move assignment, destructor

    you have to define all of them
    - implement, **`=default`**, **`=delete`**
- **As a generic programmer (thus, if you provide templates):**
  - If you forward arguments through template code, declare the arguments with && ("universal reference") and pass them with **`std::forward<>()`**
    - To deal with string literals and other C arrays you might use **`std::decay<>`**

pair<> with C++98 and C++11

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;

    pair();
    pair(const T1& x, const T2& y);

    template<class U, class V>
      pair(const pair<U,V>& p);
};
```

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;

  constexpr pair();
  pair(const T1& x, const T2& y);
  pair(const pair&) = default;
  pair(pair&&) = default;

  template<class U, class V> pair(U&& x, V&& y);
  template<class U, class V> pair(const pair<U,V>& p);
  template<class U, class V> pair(pair<U,V>&& p);

  template <class... Args1, class... Args2>
    pair(piecewise_construct_t,
         tuple<Args1...> first_args,
         tuple<Args2...> second_args) noexcept;

  pair& operator= (const pair& p);
  pair& operator= (pair&& p) noexcept(is_nothrow_move_assignable<T1>::value &&
                                      is_nothrow_move_assignable<T2>::value);
  template<class U, class V> pair& operator=(const pair<U,V>& p);
  template<class U, class V> pair& operator=(pair<U,V>&& p);

  void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
};
```

Move Semantics Summary

- **With move semantics we can optimize copies**
  - make returning vectors/string cheap (reallocation with factor 10)
  - while preserving the naive syntax
- **To benefit**
  - is pretty easy for application programmers
  - has some issues for class programmers
  - has even more issues for template programmers
  - has significant issues for foundation library programmers
- **We still have several open issues**
  - in the core language
  - in the library
- **Please, Take Care!**

Contact

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com

If you care for privacy,
please donate to enigmail:

https://www.enigmail.net/home/donations.php