Nicolai M. Josuttis

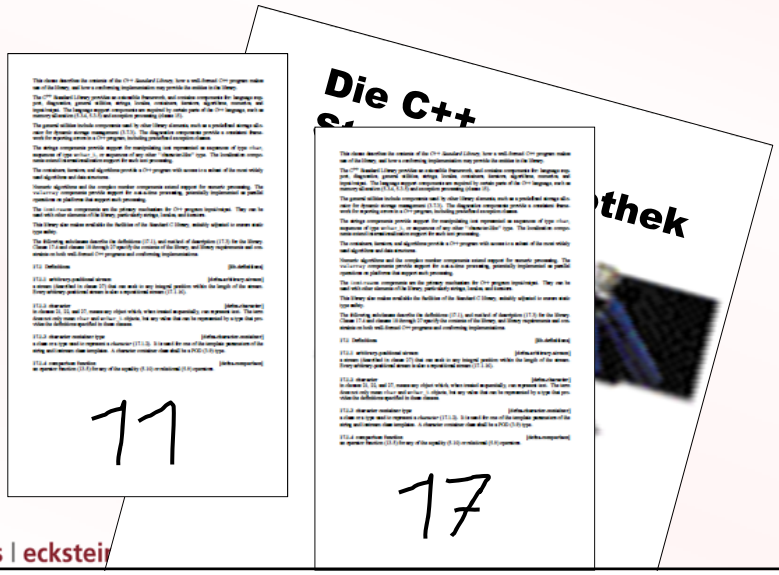
Beware of C++

C++Now 2014, Aspen

josuttis | eckstein 1

1997

- My first C++ Standardization Meeting in London



josuttis | eckstein

Disclaimer

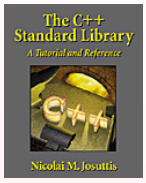
- English is not my native language
- You probably know C++ better than me
- I can be very slow
- But I am pedantic

- I will raise more questions than I answer

josuttis | eckstein 3

1998

- Final wording of "The C++ Standard Library" (1st ed.)



Nico: **What's up with this C++ library site?**
Beman: **It is a bit slow getting the site going.**
Nico: **You have to decide: Shall I mention boost?**
Beman: **OK, do it (seems I will have more time)**

See the Boost repository for C++ libraries at `http://www.boost.org/` for a collection of different smart pointer classes as an extension of the C++ standard library. (Class `CountedPtr<>` will probably be called `shared_ptr<>`.)

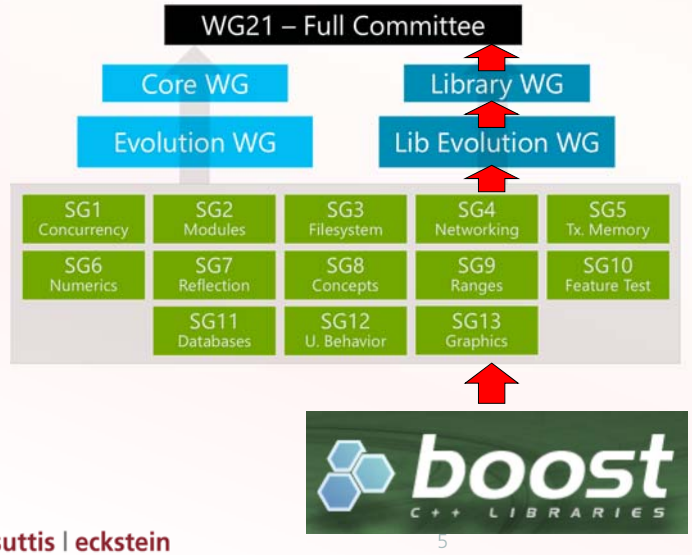
`f(g(elem1),h(elem2))`	`compose_f_gx_hy`	

Table 8.5. Possible Names of Compose Function Object Adapters

Look at the Boost repository for C++ libraries at `http://www.boost.org/` for the names that should be used in the future and for a complete implementation of all of them. In the next few subsections I discuss three of them — those that I need most often.

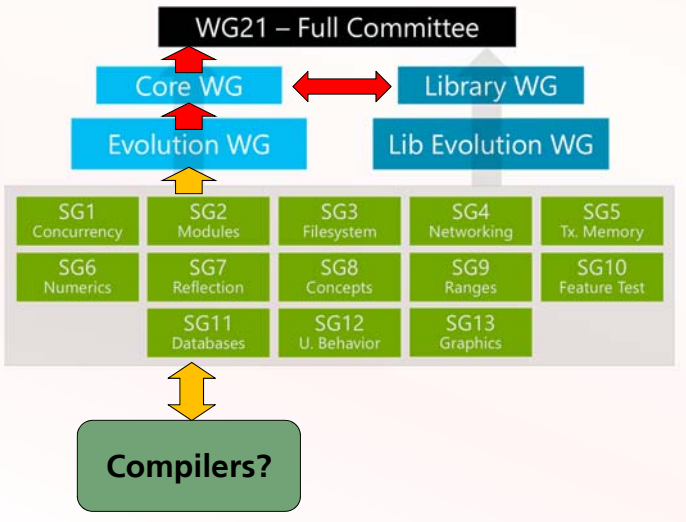
josuttis | eckstein 4

Process for New Libraries

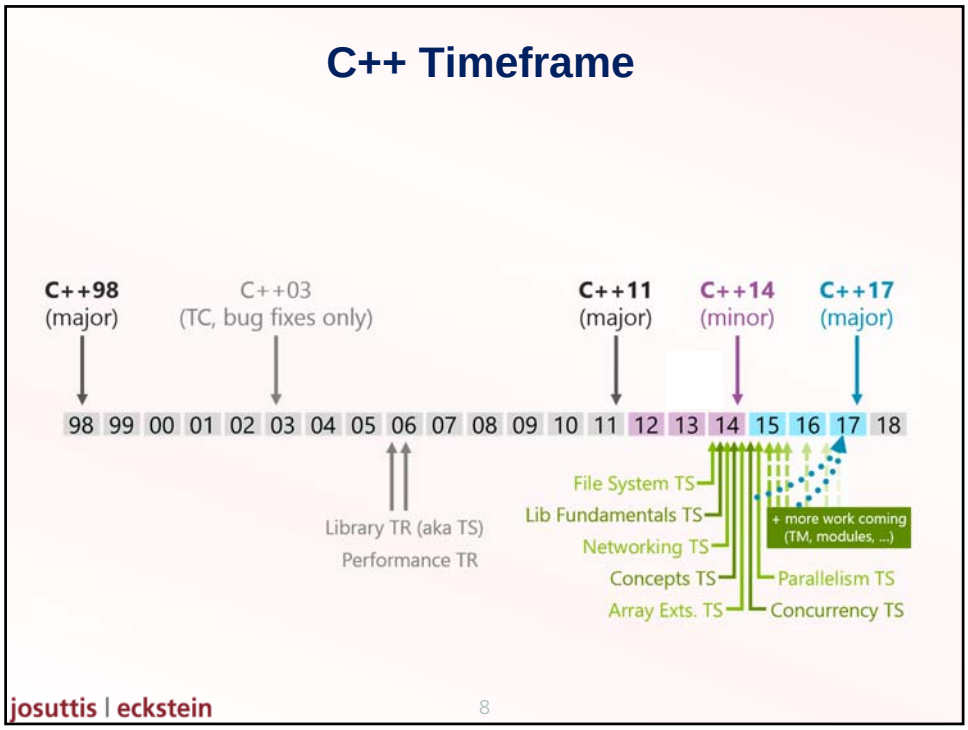


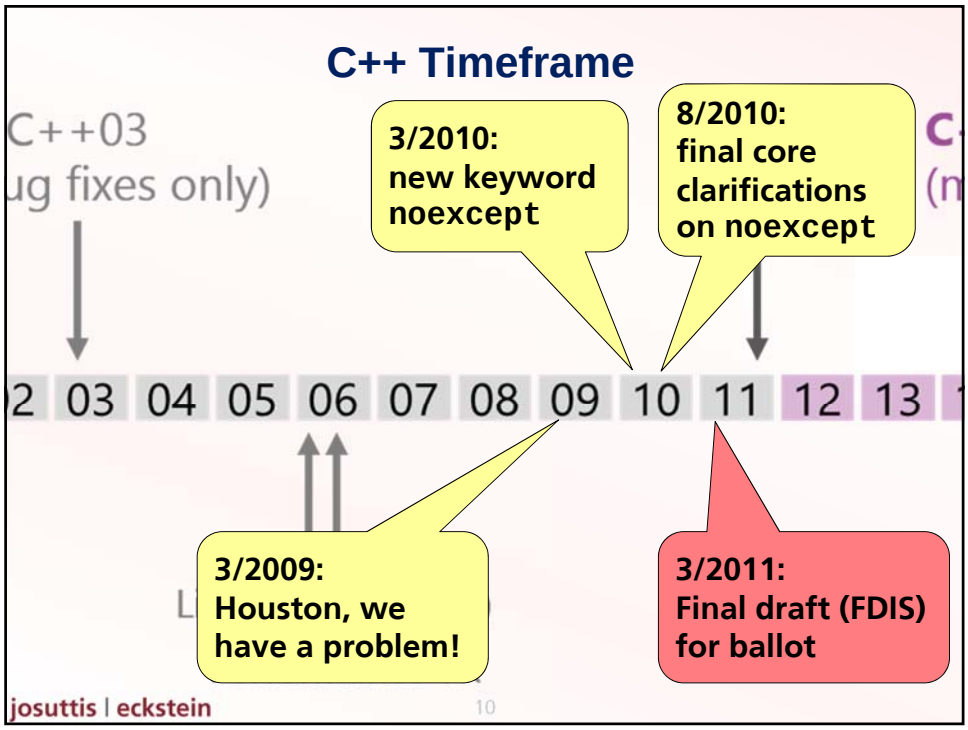
josuttis | eckstein 5

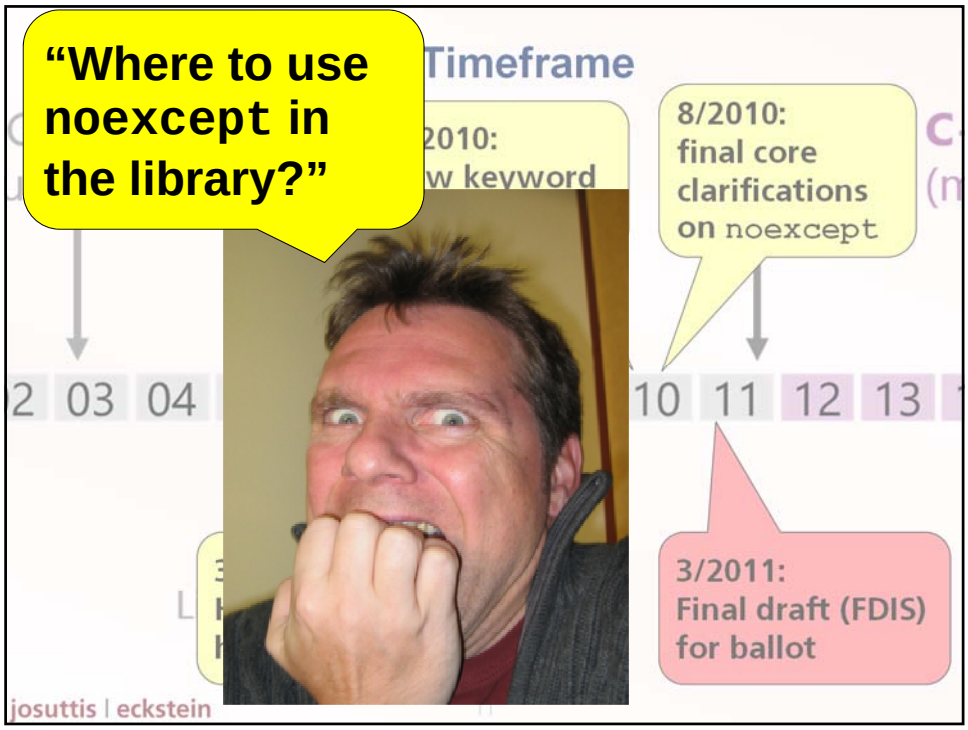
Process for New Core Features



josuttis | eckstein 6







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
  - having a **wide** contract
    [i.e. does **not** specify **undefined behavior** due to a precondition],
  - that the LWG agree **cannot throw**,

  should be marked as **unconditionally `noexcept`**.
- If a library **swap** function, **move** constructor, or **move** assignment operator ...
  - can be proven not to throw by applying the 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`.

josuttis | eckstein 14

noexcept Policy accordi

- Each library function
  - having a **wide** contract
    [i.e. does **not** specify **undefined beha**
  - that the LWG agree **cannot throw**,
  should be marked as **unconditionally noexc**.
- If a library **swap** function, **move** constructor, or **move** assignment operator ...
  - can be proven not to throw by applying the 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`.

As asked here already:
Is there any useful example (in the library) for a move operation that might throw?

noexcept Policy for the Standard Library

- **C++11/C++14 follows this policy mostly**

For a simple example:

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

According to Library Issue 2319 there is a proposal for C++17 to remove the noexcept requirement for the move constructor to give debugging implementations freedom to allocate data during a move

noexcept Policy for the Standard Library

- Standard containers don't define their move operations as explicit yet

For example:

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


Test by HH [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()
{
  vector<X> v(1000000);
  cout << "cap.: " << v.capacity() << endl;

  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=""

Test by HH [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)
     : s(x.s) {
    }
    X(X&& x) noexcept
     : s(move(x.s)) {
    }
};
```

```
int main()
{
  vector<X> v(1000000);
  cout << "cap.: " << v.capacity() << endl;

  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!

	no noexcept	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
g++49 100,000,000	core dump	core dump

josuttis | eckstein 19

Open

- **Which containers should have noexcept move operations?**
  - string and vector!
  - deque, list, associative, unordered?
- **Should we specify this in the standards as**
  - required?
  - required for release mode?
- **Should we have semantics for a definition of "strongly encouraged to be noexcept"?**
  - The proposals for the FORM were so far:
    - some form of special written noexcept (italics or so)
    - /*noexcept*/
    - [[noexcept]]
    - some form of noexcept(NDEBUG)
    - noexcept(probably)
    - throw(unlikely)
    - [[have_mercy]]

josuttis | eckstein 20

Additional noexcept Guideline?

- **If the move constructor**
  has a noexcept specification,
  the default constructor
  should have a noexcept specification
- **STL [c++std-lib-35831] :**
  - Note that default ctors and move ctors are twins when it comes to noexcept - either both should be marked, or neither.
  - This is nearly a fundamental law - if an object always needs to acquire a resource even in its default-constructed state, then the move ctor also needs to acquire such a resource (because you start with one object and end with two), in order to avoid emptier-than-empty.
  - But if an object can be default constructed noexceptly, then move construction can be implemented with default construction and nofail swap.

josuttis | eckstein 21

Additional noexcept Guideline?

- **But, HH [c++std-lib-35836] :**
  - There is a caveat here though.
  - I can not find anywhere in the allocator requirements that if the allocator is default_constructible, that it is nothrow_default_constructible.
  - We have two choices:
    1. Require that allocators be either
       !is_default_constructible<A>{} || is_nothrow_default_constructible<A>{}.
    2. vector{} is noexcept only if Allocator{} is noexcept.
       [Note: std::allocator{} is already noexcept].

    I prefer 2. It gives allocator authors more latitude for negligible cost.
  - Also we currently specify vector{} like so:
    vector() : vector(Allocator()) { }
    It would be so much better to specify it with:
    vector() noexcept(is_nothrow_default_constructible<allocator_type>{})
    I.e. not require (nor even encourage) an allocator copy construction.

josuttis | eckstein 22

Additional noexcept Guideline?

- **But, PD [c++std-lib-35832] :**
  - In my opinion, the current wide/narrow practice is wrong.
  - It's wrong on a conceptual level, because (almost) no function is actually wide. All functions have implicit requirements that their arguments, *this, and everything else reachable from them be a valid object. (Or, in the case of a constructor, that 'this' points to storage suitable to hold an object.)
  - It's also wrong because it sets up a conflict.
    When specifying, say, operator*, we now need to make a choice between adding a Requires clause and a noexcept, the two being mutually exclusive under the wide/narrow theory.
    This does not improve the quality of the specification.

josuttis | eckstein 23

- **It is key to have guidelines for how to use C++ Core Features**
- **Ideally, together with each new core feature**
- **But**
  - guidelines require experience
  - are living documents

josuttis | eckstein 24

Initializer Lists and explicit

```
class P
{
  public:
    P(int = 0);
    explicit P(std::initializer_list<int>);
};

void foo(const P&);

foo ();                 // ERROR
foo ( 47 );             // OK
foo ( {} );             // OK !
foo ( {42} );           // ERROR due to explicit
foo ( {42,43} );        // ERROR due to explicit
foo ( {42,43,44} );     // ERROR due to explicit
foo ( P{42,43,44} );    // OK, explicit conversion

P a(77);                // OK
P b{77};                // OK
P c = 77;               // OK
P d = {};               // OK !
P e = { 42 };           // ERROR due to explicit
P f = { 77,88 };        // ERROR due to explicit
```


Initializer Lists and explicit

```
class P
{
  public:
    explicit P(int = 0);
    P(std::initializer_list<int>);
};

P a;              // OK, calls  P::P(int)
P b(42);          // OK, calls  P::P(int)
P c = 42;         // ERROR

P d {};           // OK, calls  P::P(int)  (calls  P::P(initializer_list) without def. constr.)
P e { 77 };       // OK, calls  P::P(initializer_list)
P f { 77, 5 };    // OK, calls  P::P(initializer_list)

P g = {};         // ERROR (calls  P::P(initializer_list) without default constructor)
P h = { 77 };     // OK, calls  P::P(initializer_list)
P i = { 77, 5 }; // OK, calls  P::P(initializer_list)
```

Library Issue 2193

Resolution for C++14:
Split constructors:

```
~~explicit~~ vector();
explicit vector(const Allocator&);
```

```
template <class T, class Alloc
class vector {
  public:
    explicit vector(const Allocator& = Allocator());
    explicit vector(size_type n);
    vector(size_type n, const T& value, const Allocator& = Allocator());
    template <class InputIterator>
      vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
    vector(const vector<T,Allocator>& x);
    vector(initializer_list<T>, const Allocator& = Allocator());
    …
};

vector<int> v1 = { 1, 2 };  // OK
vector<int> v2 = { 1 };     // OK
vector<int> v3 = {};        // ERROR

template <typename ...T>
void g ( T... t ) {
  vector<int> v = { t... };  // OK for g(1), g(1,2), g(1,2,3), ... but ERROR for g()
}
```

Thanks to Jonathan Wakely and Marshall Clow for this example

josuttis | eckstein 34

Guidelines for `explicit` ?

- **The default constructor should never be `explicit`**
  - If all arguments of an explicit constructor have default values, declare the default constructor separately
- **An initializer list constructor should never be `explicit`**
- **Any other constructor should be `explicit`, if**
  - parameters affect behavior instead of core content
- **Shouldn't the default constructor always be its own beast?**

josuttis | eckstein 35

We still have more explicit initialization Mess

```
template <class... Types>
class tuple {
 public:
  constexpr tuple();
  explicit constexpr tuple(const Types&...);
  template <class... UTypes>
    explicit constexpr tuple(UTypes&&...);
  tuple(const tuple&) = default;
  tuple(tuple&&) = default;
  template <class... UTypes>
    constexpr tuple(const tuple<UTypes...>&);
  template <class... UTypes>
    constexpr tuple(tuple<UTypes...>&&);
  template <class U1, class U2>
    constexpr tuple(const pair<U1, U2>&); // only if sizeof...(Types) == 2
  template <class U1, class U2>
    constexpr tuple(pair<U1, U2>&&); // only if sizeof...(Types) == 2
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a);
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, const Types&...);
  template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, const tuple&);
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, tuple&&);
  template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
  template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
  template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
  template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

Class tuple<> has 18 constructors

Guidelines for `constexpr` ?

- **The current situation is a complete mess**
- **Some recent quotes from the reflector:**
  - Constexpr is not for optimization. The compilers can inline well already.
  - Use constexpr when guaranteed static initialization is important. E.g. the construction of global atomics really cannot be deferred to run time.
  - Use constexpr when you anticipate using the results to define array sizes or appear within template non-type arguments.
  - I think that "making everything possible constexpr" is borderline insane. It leads to unnecessarily increased compile times, potential code bloat, and wishes to overload on constexpr so that we can select different algorithms for compile time and run time.
  - By all means "be generous," but use constexpr only when there is a potential need for guaranteed compile-time evaluation.
  - Beneficial uses of constexpr on non-trivial computations aren't always obvious from past experience.

Guidelines for `constexpr` ?

- **Some recent quotes from the reflector (cont.):**
  - There seems to be a "potential" need for everything that can be constexpr to be constexpr. How do we gain confidence that nobody's going to need to use some function in a static initializer?
  - I saw someone asking if main() could be constexpr.
  - It will not be easy to draw a simple and clear line between constexpr and non-constexpr in the library, but I think we have to try. We are supposed to be experts, so we should be better at drawing a line than the average programmer.
  - constexpr can always be added later when there is empirical evidence showing benefit.
  - The problem is, when us experts get it wrong, everyone else waits years for us to release a fix. As per our vote in Chicago, vendors are not allowed to offer a fixed version as a conforming extension.
  - Yes, I don't think it is a question we can ignore, but for a solution to become a rule someone has to propose something and gather a consensus.

Which function has changed with each and every C++ Standard, so that we have 4 different definitions now ?

make_pair() in C++98

```
namespace std {
  // implementation according to C++98:

  template <typename T1, typename T2>
  pair<T1,T2> make_pair (const T1& x, const T2& y) {
    return pair<T1,T2>(x,y);
  }

}

std::make_pair (42, "hi") // => std::pair<int,const char[3]>
```


Trying:

```
namespace std {
  template <typename T1, typename T2>
  pair<T1,T2> make_pair (const T1& x,
                         const T2& y) {
    return pair<T1,T2>(x,y);
  }
}

int main()
{
  std::map<int,std::string> m;
  if (*m.begin() == std::make_pair(42,"hi")) {
    ;
  }
}
```

=> Error message with 238 rows

181. make_pair() unintended behavior

Section: 20.3 [pairs] **Status:** TC1 **Submitter:** Andrew Koenig **Opened:** 1999-08-03 **Last modified:** 2012-11-14

View all other issues in [pairs].

View all issues with TC1 status.

Discussion:

The claim has surfaced in Usenet that expressions such as

```
make_pair("abc", 3)
```

are illegal, notwithstanding their use in examples, because template instantiation tries to bind the first template parameter to `const char (&)[4]`, which type is uncopyable.

I doubt anyone intended that behavior...

Proposed resolution:

In 20.2 [utility], paragraph 1 change the following declaration of make_pair():

```
template <class T1, class T2> pair<T1,T2> make_pair(const T1&, const T2&);
```

to:

```
template <class T1, class T2> pair<T1,T2> make_pair(T1, T2);
```

In 20.3 [pairs] paragraph 7 and the line before, change:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

to:

```
template <class T1, class T2>
pair<T1, T2> make_pair(T1 x, T2 y);
```

and add the following footnote to the effects clause:

> According to 12.8 [class.copy], an implementation is permitted to not perform a copy of an argument, thus avoiding unnecessary copies.

Rationale:

Two potential fixes were suggested by Matt Austern and Dietmar Kühl, respectively, 1) overloading with array arguments, and 2) use of a reference_traits class with a specialization for arrays. Andy Koenig suggested changing to pass by value. In discussion, it appeared that this was a much smaller change to the standard that the other two suggestions, and any efficiency concerns were more than offset by the advantages of the solution. Two implementors reported that the proposed resolution passed their test suites.

make_pair() in C++03

```
namespace std {
  // implementation according to C++03:

  template <typename T1, typename T2>
  pair<T1,T2> make_pair (T1 x, T2 y) {   // by value !
    return pair<T1,T2>(x,y);
  }

}


std::make_pair (42, "hi") // => std::pair<int,const char*>
```

josuttis | eckstein 44

make_pair() in C++11

- `std::make_pair()` shall support move semantics
  => rvalue references have to be used
  => we have the decay problem again
  => we have to fix that problem with `std::decay<>`

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

std::make_pair (42, "hi")  // returns a std::pair<int,const char*>
```

josuttis | eckstein 45

make_pair() in C++11

- `std::make_pair()` shall support move semantics
  => rvalue references have to be used
  => we have the decay problem aga
  => we have to fix that problem wit

Note:
std::decay<> in addition strips *cv*-qualifiers from class types

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

std::make_pair (42, "hi")  // returns a std::pair<int,const char*>
```

josuttis | eckstein 46

make_pair() in C++14

- `std::decay_t<>`
- no change in semantics

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

std::make_pair (42, "hi")  // returns a std::pair<int,const char*>
```

josuttis | eckstein 47

Using the RET Trick

```
namespace std {
  template <typename T1, typename T2,
            typename RET = pair<std::decay_t<T1>,
                                std::decay_t<T2>>>
  RET make_pair (T1&& x, T2&& y) {
    return RET(std::forward<T1>(x),std::forward<T2>(y));
  }
}
```

- E.g. used in gcc in some places
- Danger: programmers are sneaky little buggers and may explicitly provide the third type
  - except for constructors

Thanks to Jonathan Wakely

Dealing with Templates

- For all these C++ templates (and xml, html, ...) it helps to see matching angle brackets:

```
template <typename T1, typename T2,
          typename RET = pair<std::decay_t<T1>,
                              std::decay_t<T2>>>
```

- VIM: In .vimrc:

```
" match pairs of < and >
autocmd FileType cpp set mps+=<:>
```

Guidelines for Template Parameters?

- **If you know that the object is always cheap to copy then pass by value.**
- **If it might not be cheap to copy, you have to make a choice:**
  - If the expected type is likely to be an rvalue and is moveable, then you call by value so that the caller passes temporaries or uses move
  - If it's not cheap to copy and not moveable, then still take by value and let the caller use std::ref()
  - Otherwise use const lvalue reference
    - Think about whether and where to decay
- **If you return something in the argument, use a non-const lvalue reference**
- **If you have to pass move semantics into other parts of the called function, declare as universal reference and forward<>**
  - Think about whether and where to decay

josuttis | eckstein 50

Eric Niebler @ C++Now May 14, 2014:

aerix consulting

Passing and Returning in C++11

Category	C++11 Recommendation
Input	
small & “sink”	Pass by value
all others	Pass by const ref
Output	Return by value
Input/Output	Pass by non-const ref (?)

 51

Eric Niebler @ C++Now May 14, 2014:

aerix consulting

Passing and Returning in C++11

Category	C++11 Recommendation
Input	
small & "sink"	Pass by value
all others	Pass by const ref
Output	Return by value
Input/Output	Use a stateful algorithm object (*)

(*) Initial state is a **sink** argument to the constructor

 52

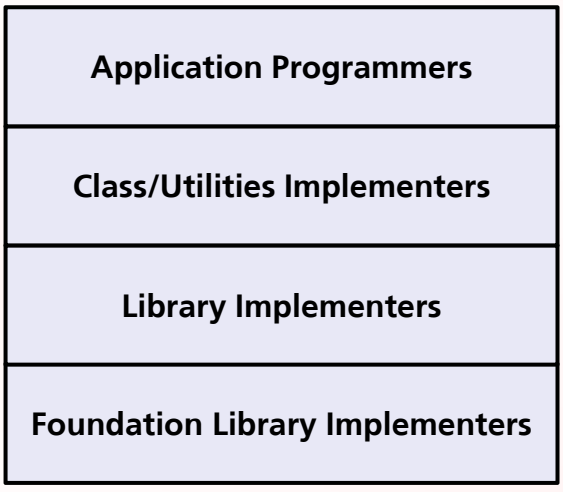
- **It is key to have guidelines for how to use C++ Core Features**
- **Ideally, before we have to adapt them in the library**
- **That's way before Scott or Herb write books about them!**

josuttis | eckstein 53

We need guidelines for

- **noexcept**
  - new version
- **explicit**
  - I plan to provide a first draft
- **constexpr**
- **template parameters**
- **...**

josuttis | eckstein 54



- **Same guidelines for different groups?**
- **Which group has to know and understand which detail?**

josuttis | eckstein 55

Beware of C++

- **Which problems of this talk should the ordinary C++ programmer know?**

Too much!

josuttis | eckstein 56

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com

josuttis | eckstein 57