# The Biggest Mistakes in the C++11 Library

Nicolai M. Josuttis
IT-communication.com

03/14

C++

1
josuttis | eckstein
IT communication

# Our Biggest Mistakes in the C++11 / C++14 Core and Library

Nicolai M. Josuttis
IT-communication.com

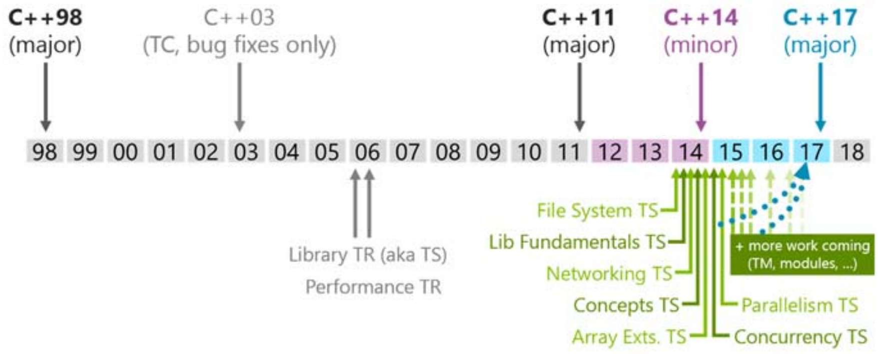
03/14

C++

2
josuttis | eckstein
IT communication

## C++ Timeframe

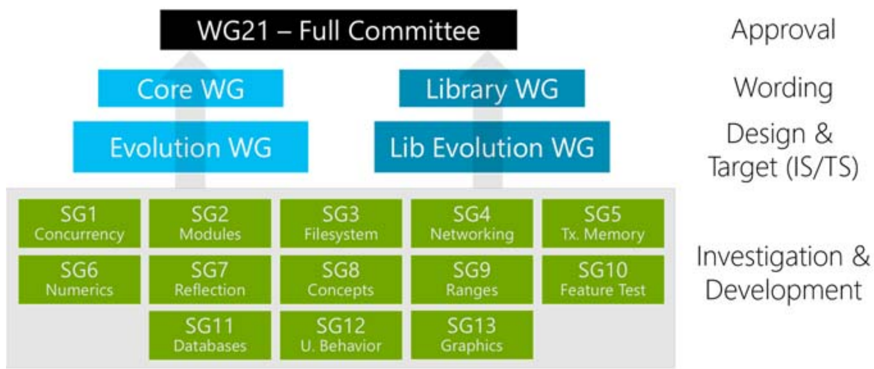
- **http://isocpp.org/std/status:**
  - Library and Language Specifications (TS's, 2014-) and New Standards (C++14, C++17)






## C++ Working and Study Groups

- **http://isocpp.org/std/the-committee:**



## iota()

**What is the output of this program:**

```
#include <vector>
#include <numeric>
#include <iostream>

int main()
{
  std::vector<int> coll = { 1, 2, 3, 5, 7, 11, 13, 17, 23 };

  std::iota (coll.begin(), coll.end(),   // range
             42);                        // value

  for (const auto& elem : coll) {
    std::cout << "elem: " << elem << std::endl;
  }
}
```

```
elem: 42
elem: 43
elem: 44
elem: 45
elem: 46
elem: 47
elem: 48
elem: 49
elem: 50
```



## Range-Based for Loops

**„do something with each element"**

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

// modify each elements of coll (using a range-based for loop):
for (auto elem : coll) {      // OOPS
    elem *= 2;
}
```

Note:
Without being declared as reference elem is a copy of the original element

## Next Generation of Range-Based for Loops

- **N3853:** ***Next Generation of Range-Based for Loops***
  - http://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3853.txt
- **Possibly in C++17**
  - Feb 2014:
    accepted by Evolution Working Group
    moved to Core Working Group

```
// print all elements of coll (using a range-based for loop)
// without copying the element:
for (elem : coll) {
    std::cout << elem << std::endl;
}
```

Note: `elem` has **no type**

C++
©2014 by IT-communication.com 7
josuttis | eckstein IT communication

## Next Generation of Range-Based for Loops

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

```
for (elem : coll) {
    std::cout << elem << std::endl;
}
```

```
for (auto&& identifier : coll ) {
    statement
}
```

```
for (auto&& elem : coll) {
    std::cout << elem << std::endl;
}
```

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

```
{
 for (auto&& _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos ) {
     auto&& elem = *_pos;
     std::cout << elem << std::endl;
 }
}
```

C++
©2014 by IT-communication.com 8
IT communication

## Overload Resolutions for Constructors with Initializer Lists

- **( ) Initialization calls ordinary constructors only**
- **{} Initialization calls initializer list or ordinary constructors**
  - Initializer list constructors have higher priority
  - but the default constructor has highest priority

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

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

P d {};           // calls  P::P(int)  (calls  P::P(initializer_list) without default constructor)
P e { 77 };       // calls  P::P(initializer_list)
P f { 77, 5 };    // calls  P::P(initializer_list)

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

**Consequence:**

`vector<int> v1(3,42);` → | 42 | 42 | 42 |

`vector<int> v2{3,42};` → | 3 | 42 |

C++





josuttis | eckstein IT communication

## Keyword `explicit`

- **Constructors for one argument and type conversion operators define implicit type conversions**
- **By using `explicit` you can disable these implicit conversions**
  - Useful if conversions change the semantics
- **Note:**
  - Initializations with **=** count as **`explicit`** conversions

```
class Collection
{
  public:
    explicit Collection (int);   // initial size
    …
};

void foo (const Collection&);

fp(42);                  // ERROR (would be OK without explicit)
fp(Collection(10));      // OK, explicit conversion

Collection c1(10);                  // OK
Collection c2 = 10;                 // ERROR due to explicit
Collection c3 = Collection(10);     // OK
```

C++





josuttis | eckstein IT communication

## Initializer Lists and explicit

- **`explicit` now also has an impact on constructors with 0, 2, or more arguments**
  - `explicit` inhibits automatic conversions from initializer lists

```
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 x(77);                 // OK
P y{77};                 // OK
P v = 77;                // OK
P w = {77};              // ERROR due to explicit
```



## Overload Resolutions for Constructors with Initializer Lists

- **( ) Initialization calls ordinary constructors only**
- **{} Initialization calls initializer list or ordinary constructors**
  - Initializer list constructors have higher priority
  - but the default constructor has highest priority

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

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

P d {};            // calls P::P(int) (calls P::P(initializer_list) without default constructor)
P e { 77 };        // calls P::P(initializer_list)
P f { 77, 5 };     // calls P::P(initializer_list)

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

## Library Issue 2193

- **It's not a good idea to ha**
  **with a non-explicit initial**

**Resolution for C++14:**
**Split constructors:**

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

```
template <class T, class Alloca
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(2), g(1,2,3), ... but ERROR for g()
}
```

Thanks to Jonathan Wakely and Marshall Clow for this example

C++ josuttis | eckstein IT communication





## 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**

C++ josuttis | eckstein IT communication

## Keyword constexpr

- **The motivating example:**

```
template<> class numeric_limits<int> {
  public:
    static constexpr int max() throw() { return __INT_MAX__; }
    …
};

const int x = std::numeric_limits<int>::max();
int arr[x];   // Error with C++98/C++03,  OK with C++11
```



## Variadic Templates

- **Templates for a variable number of template arguments**
  – varargs for templates
- **For classes and functions**
- **To end the recursion you usually need a non-template function**

```
void print ()
{
}

template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
   std::cout << firstArg << std::endl;
   print(args...);
}
```

## Variadic Templates

```
print ( 7.5, "hello", std::string("s") )

=> print<double> ( 7.5, "hello", std::string("s") )
     std::cout << 7.5 << std::endl;
     print ( "hello", std::string(s) )

     => print<const char*> ( "hello", std::string("s") )
          std::cout << "hello" << std::endl;
          print ( std::string(s) )

          => print<std::string> ( std::string("s") )
               std::cout << std::string("s") << std::endl
               print ( )
```

```
void print ()
{
}

template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args )
{
   std::cout << firstArg << std::endl;
   print( args... );
}
```

C++


josuttis | eckstein
IT communication

## Variadic Templates Overloads

- **In C++11 strictly speaking ending the recursion with an overload without the variadic argument is invalid**
  - Ambiguity because both functions match when calling `print()` with one argument (however compilers handle it as meant)
- **But this is "fixed with C++14**

```
template <typename T>
void print (const T& arg)
{
   std::cout << arg << std::endl;
}

template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
   print(firstArg);
   print(args...);
}
```

C++


josuttis | eckstein
IT communication

## C++14: Variable Templates

- **C++14 provides the ability to define simple generic variables/objects/references ("variable templates")**
  - can have default template parameters
  - can be partially specialized
  - can't be overloaded

```
template <typename T>
T pi{3.1415926535897932385};

std::cout << pi<double> << std::endl;
```

---

```
template <typename T = long double>
constexpr T pi = T(3.1415926535897932385);

std::cout << pi<> << std::endl;   // OK
std::cout << pi << std::endl;     // error
```



## C++14: Variable Templates

- **C++14 provides the ability to define simple generic variables/objects/references ("variable templates")**
  - can have default template parameters
  - can be partially specialized
  - can't be overloaded

```
template <typename T>
T pi{3.1415926535897932385};

std::cout << pi<double> << std::endl;
```

---

```
template &lt;typename T = long double&gt;
constexpr T pi = T(3.1415926535897932385);

std::cout << pi<> << std::endl;   // OK
std::cout << pi << std::endl;     // error
```

**"Variadic Template"**
**≠**
**"Variable Template"**

## `[]` and `data()` for `basic_string<>`

- **C++98 / C++03:**

  `operator[]`
  - Returns: If *pos* < size(), returns data()[*pos*].
    Otherwise, **if *pos* == size(), the const version returns charT()**. Otherwise, the behavior is undefined.

  `data()`
  - Returns: If size() is nonzero, the member returns a pointer to the initial element of an array whose first size() elements equal the corresponding elements of the string controlled by *this.

- **C++11/C++14:**

  `operator[]`
  - *Requires:* pos <= size().
  - *Returns:* *(begin() + pos) if pos < size(),
    **otherwise** a reference to an object of type T with value charT()

  `data()`
  - *Returns:* A pointer p such that p + i == &operator[](i) for each i **in [`0,size()`]**.

C++
©2014 by IT-communication.com
21
josuttis | eckstein IT communication

## `[]` and `data()` for `basic_string<>`

- **C++98 / C++03:**

  `operator[]`
  - Returns: If *pos* < size(), returns dat
    Otherwise, **if *pos* == size(), the co**
    the behavior is undefined.

  `data()`
  - Returns: If size() is nonzero, the
    element of an array whose fir
    elements of the string contr ed by

**Thus: Since C++11:**

- **For non-const strings:**
```
string s;
s[size()] == '\0'
```
- **For const/non-const strings:**
  **Besides `c_str()` also `data()` is guaranteed to end with `'\0'`**

- **C++11/C++14:**

  `operator[]`
  - *Requires:* pos <= size().
  - *Returns:* *(begin() + pos) if pos < size(),
    **otherwise** a reference to an object of type T with value charT()

  `data()`
  - *Returns:* A pointer p such that p + i == &operator[](i) for each i **in [`0,size()`]**.

C++
©2014 by IT-communication.com
22
josuttis | eckstein IT communication

## `[]` and `data()` for `basic_string<>`

- **C++98 / C++03:**
  `operator[]`
  – Returns: If *pos* < s
  Otherwise, **if *pos***
  the behavior is un
  `data()`
  – Returns: If size()
  element of an arra
  elements of the s
- **C++11/C++14:**
  `operator[]`
  – *Requires:* po
  – *Returns:* *(b
  **otherwise** a re
  `data()`
  – *Returns:* A pointer

**But for string_view (N3849 => C++17?):**

`operator[]`
– Note: Unlike **`basic_string::operator[]`**, **`basic_string_view::operator[](size())`** has undefined behavior instead of returning charT().

`data()`
– Note: Unlike **`std::string::data()`** and string literals, **`data()`** may return a pointer to a **buffer that is not null-terminated**.

**Thus, for kind-of-string objects in general:**
- **calling [size()] is undefined**
- **Do not use data() when you need a null terminated raw string**

C++ 23 josuttis | eckstein IT communication


## Disclaimer

- **The following slides are not provided to blame somebody**
  – We are all interested in the quality of C++
  – We all make mistakes
  – Especially if there is no existing practice

C++ 24 josuttis | eckstein IT communication

## async() and Futures

- **async(*args*)**
  - starts asynchronous task
    - either in background or deferred until outcome is requested
  - returns a future
    - a handle for the future outcome (return value or exception)
  - *args* might be any *callable* plus optional arguments
- **Calling get() on the future blocks until end of execution and yields the outcome**
  - starts task if not started yet and waits for its end

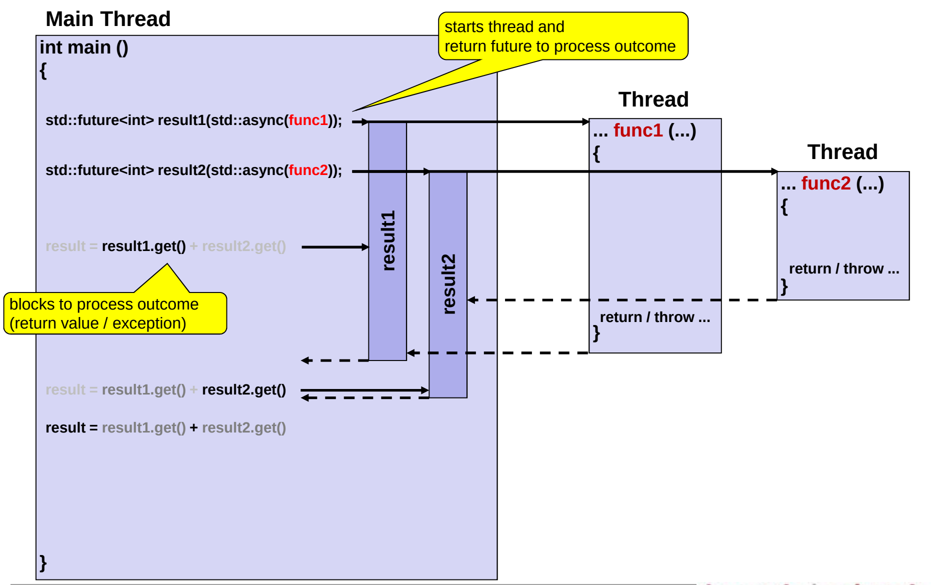
```
int result = func1() + func2()
```

⇨

```
// start both tasks asynchronously
std::future<int> result1(std::async(func1));
std::future<int> result2(std::async(func2));
…
// use outcome of both tasks
int result = result1.get() + result2.get();
```

## async() with 2 Threads Available



## Speculative Execution

```
int accurateComputation();  // an accurate result, which might take a while
int quickComputation();     // a fast result but not as accurate

int bestResultInTime()
{
   // define maximum time slot to return result:
   auto tp = std::chrono::system_clock::now() + std::chrono::minutes(1);

   // start both a quick and an accurate computation:
   auto f = std::async (std::launch::async, accurateComputation);
   int guess = quickComputation();

   // give accurate computation the rest of the time slot:
   std::future_status s = f.wait_until(tp);

   // return the best computation result we have:
   if (s == std::future_status::ready) {
      return f.get();
   }
   else {
      return guess;
   }
}
```

**There is a bug!**

C++





josuttis | eckstein IT communication

## async() History

**Mai 2011:**

- **What does the following program do:**

```
int main()
{
    std::async(std::launch::async,foo);
    bar();
}
```

**=>The standard has to be read as "destructor blocks"**

- async(): "the completion of the function foo is sequenced before (1.10) the shared state is made ready."

**=> Fix in Apples implementation;**
**conforms with gcc and VC++**

C++





josuttis | eckstein IT communication

## async() History

**Sep 2012: N3451 (Herb Sutter): We have some problems:**

**1)**

```
async(foo);
async(bar);
```

**is different from:**

```
auto f1 = async(foo);
auto f2 = async(bar);
```

**2)**

```
async(launch_policy::async,foo);
async(launch_policy::async,bar);
```

**is running foo() and bar() sequentially**

**3)**

```
void func() {
  future<int> f = start_some_work();
  ...     // no f.get()
}
```

**might or might not block depending on how f was created**

C++





josuttis | eckstein IT communication

## async() History

- **Without a decision, Microsoft changed its implementation**
  - destructor of future never blocks
    - because blocking is evil such as for a GUI thread
- **Lots and Lots and Lots of heated discussions:**
  - We want to be able to decide based on the type whether the destructor might block
  - Destructor must not block
  - We have to be backward compatible
  - We can add a member to know how it was created
    - Breaks binary compatibility
  - Let's change the return type of async()
    - ABI changing break
  - Let's introduce waiting_future and non_waiting_future
  - Let's introduce detach() for a future
  - ...

C++





josuttis | eckstein IT communication

## async() History

- **Oct 1, 2013:**
  - Heated discussion again
  - As a "solution":
    - **Let's deprecate async()**
    - Two straw polls inside Concurrency Working Group:

| | | |
|---|---|---|
| – Strongly in favor: | 13 | 12 |
| – Weakly in favor: | 6 | 4 |
| – Neutral: | 1 | 2 |
| – Weakly against: | 1 | 0 |
| – Strongly against: | 1 | 4 |

- **N3777:** ***Wording for Deprecating async()***

C++ ©2014 by IT-communication.com 31 josuttis | eckstein IT communication

## async() History

- **N3780:** ***Why Deprecating async() is the Worst of all Options***
  - not solving the problem (you have to stay backward compatible)
  - bad message to the community
  - Tweet:

    @gpakosz: @stefanusdutoit
    The fact that top men standardized something already broken tells me
    I'm not smart enough to use C++11 or 14 in production

- **All except 2 national bodies would vote against it**
  - No consensus, risk for C++2014

  **=> leave it as it is**

C++ ©2014 by IT-communication.com 32 josuttis | eckstein IT communication

## async() History

- **Clarifications added to C++14:**
  - **N3776:** ***Wording for ~future***:
    - ... and add:
      [*Note:* If a future obtained from std::async is moved outside the local scope, other code that uses the future must be aware that the future's destructor may block for the shared state to become ready.—*end note*]
- **Consequence in practice:**
  - Calling **`async()`** without calling **`get()`** for the returned future is not portable

**t.b.c.**



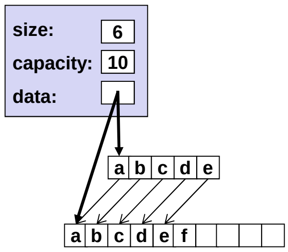
## Exception Safety Guarantees in the Standard Library

- **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
- **No-throw guarantee**
  - An operation does not throw any exception

## Exception Safety Guarantee for Vector's push_back()

- **Since C++98, the C++ Standard requires the strong 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."
- **In C++98/C++03 that's 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
    - free old memory
    - update size and capacity

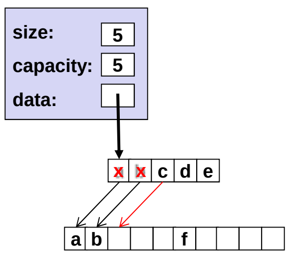



C++
©2014 by IT-communication.com
35
josuttis | eckstein
IT communication

## Exception Safety Guarantee for Vector's push_back()

- **With move semantics the strong guarantee is no longer possible**
  - Moving elements is not a reversible step but might throw
- **For this reason:**
  **Vectors use move semantics only if the elements can't throw**
  - Which requires a guarantee by the element's type in form of a declaration not to throw
  - However, the element's type might be a template, so that the guarantee might depend on another type

    => It's worth to have a conditional guarantee not to throw




C++
©2014 by IT-communication.com
36
josuttis | eckstein
IT communication

## Keyword noexcept

- **Keyword `noexcept` can be used**
  - in function declarations
    - to specify whether a function can't (or is not prepared to) throw
  - as operator
    - which yields **`true`** if an expression can't 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



## `noexcept` Policy for the Standard Library

- **According to N3279** (partly literally quoted)**:**
  - Each library function ... that ... cannot throw and does not specify any undefined behavior - for example, caused by a broken precondition - 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, conditionally throws, it should be marked as conditionally **`noexcept`**. No other function should use a conditional **`noexcept`** specification.
  - No library destructor should throw. It must use the implicitly supplied (nonthrowing) exception specification.
  - Library functions designed for compatibility with C code ... may be marked as unconditionally **`noexcept`**.

## `noexcept` Policy for the Standard Library

- **Thus:**
  - If possible, to support move semantics declare move operations as (conditionally) **`noexcept`**
- **The standard library does that mostly**

**For a simple example:**

According to Library Issue 2319
http://cplusplus.github.io/LWG/lwg-active.html#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

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

C++

39
josuttis | eckstein IT communication

## `noexcept` Policy for the Standard Library

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

**For example:**

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

C++

40
josuttis | eckstein IT communication

## Keywords

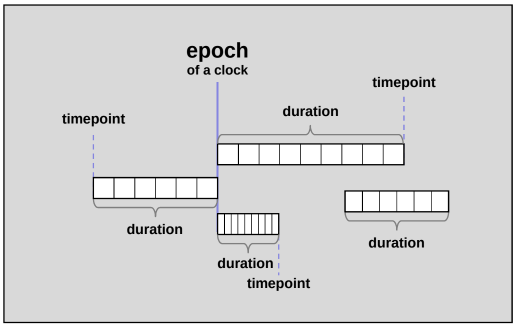
- **Is this a valid compilation unit?:**

```
class final final
{
    void override() override;
};


class Base
{
    virtual void override();
};

class final final : public Base
{
    void override() override;
};
```



## Timepoints and Durations

- **Duration**
  - number of ticks over a time unit (ratio<> in seconds)
- **Timepoint**
  - duration since the Epoch
- **Clock**
  - defines Epoch



## Example for Using Durations

```
#include <chrono>
#include <iostream>
using namespace std;

int main()
{
    chrono::milliseconds threeMilliseconds(3);
    chrono::seconds      secs(24);
    chrono::hours        aDay(24);

    cout << secs.count() << endl;
    cout << aDay.count() << endl;

    cout << chrono::seconds(aDay).count() << " seconds" << endl;
    cout << chrono::hours(secs).count() << " hours" << endl;    // compile time ERROR

    auto d1 = aDay - chrono::hours(1);      // OK: unit is hours
    cout << d1.count() << endl;
    auto d2 = aDay - chrono::minutes(10);   // OK: unit is minutes
    cout << d2.count() << endl;

    aDay -= chrono::hours(1);               // OK
    cout << aDay.count() << " hours" << endl;

    aDay -= chrono::minutes(10);            // ERROR: would lose information
}
```

**Output:**

```
24
24

86400 seconds

23

1430

23 hours
```



## Duration Initialization

- **Durations have no zero initialization defined for the default constructor**

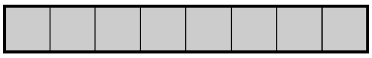
```
template <class Rep, class Period = ratio<1>>
class duration {
  public:
    // default default constructor (constexpr doesn't force value initialization):
    constexpr duration() = default;
    …
};
```

- **Thus, default constructed values may have an undefined value:**

```
std::chrono::duration<int> d;   // d has undefined value

chrono::minutes m;     // m has undefined value
chrono::minutes m{};   // m.count() == 0
chrono::minutes m();   // a very clever declaration for geek bars
```

Thanks to Howard Hinnant for pointing this out

## Array<>



- **Fixed sized array of elements**
- **Container category:**
  - fulfills general container requirements, except:
    - default constructed array is not empty
    - default constructed array may have undefined values
    - swap has no constant complexity
    - after swap iterators and reference refer to different values (and not to different containers)
  - fulfills requirements of reversible container

```
array<int,10> a = { 11, 22, 33, 44 };   // create array with 10 ints

a.back() = 9999999;                     // modify last element
a[a.size()-2] = 42;                     // modify element before last element

// process sum of all elements
cout << "sum: " << accumulate(a.begin(),a.end(),0) << endl;
```

C++





josuttis | eckstein IT communication

## Array<>

- **Array<> is an aggregate**
  - no constructors defined
  - initialization only via initializer lists (and copying)
  - without initialization FDT values are undefined (default initialized)
  - smaller initializer lists result into value initialization (zero initialization for FDTs)

```
std::array<int,5> c1 = { 1, 2, 3, 4, 5 };      // OK: array with 5 elements
std::array<int,5> c2 = { 1, 2 };               // OK: array with: 1, 2, 0, 0, 0
std::array<int,5> c3 = { 1, 2, 3, 4, 5, 6 };   // Error at compile time

std::array<int,5> c4;                          // OOPS: undefined values
std::array<int,5> c5 = {};                     // OK: all 0 (initialize with int())

std::array<int,5> c6( { 1, 2, 3, 4, 5 } );     // ERROR: no constructor for initializer list
std::vector<int>  cv( { 1, 2, 3, 4, 5 } );     // OK for all other containers
```

C++





josuttis | eckstein IT communication

## Hash Functions

- **In C++11 there is no generic hash function**
  - which is good: good hash functions are hard to implement
  - which is bad: no hash function might be worse (Java has one)
- **For any non-trivial type you have to provide a hash function**
  - Function object
  - Lambda
- **For example:**

```
class Customer {
   …
};

class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return …
    }
};

std::unordered_set<Customer,CustomerHash> custset;
```



## Generic Hash Function

```
template <typename T>
inline std::size_t get_hash (const T& val)
{
    return hash<T>()(val);  // equivalent to: hash<T>().operator()(val)
}

template <typename T, typename... Types>
inline std::size_t get_hash (const T& val, const Types&... args)
{
    return hash<T>()(val) + get_hash(args...);  // poor hash function!
}

class Customer {
  private:
    string firstname;
    string lastname;
    int    year;
  public:
    …
    friend class CustomerHash;
};

class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
        return get_hash(c.firstname,c.lastname,c.year);
    }
};
```

**Better approach:**
- get_hash()
- hash out of a tuple
- hash out of a range
- accumulate() with hash

**based on a generic hash_combine (see boost):**

```
template <typename T>
void hash_combine (size_t& seed, const T& v)
{
  seed ^= hash_value(v) + 0x9e3779b9
          + (seed << 6) + (seed >> 2);
}
```

## Generic Hash Function

```
#include <functional>

// from boost (functional/hash); see http://www.boost.org/doc/libs/1_35_0/doc/html/hash/combine.html
template <class T>
inline void hash_combine (std::size_t& seed, const T& val)
{
    seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}

// create a hash value using a seed
template <typename T>
inline void hash_val (std::size_t& seed, const T& val)
{
    hash_combine(seed,val);
}
template <typename T, typename... Types>
inline void hash_val (std::size_t& seed, const T& val,
                      const Types&... args)
{
    hash_combine(seed,val);
    hash_val(seed,args...);
}

// create a hash value out of a heterogeneous list of arguments
template <typename... Types>
inline std::size_t hash_val (const Types&... args)
{
    std::size_t seed = 0;        // initial seed
    hash_val (seed, args...);    // create hash value with this seed
    return seed;
}
```

**Usage example:**

```
class Customer {
  private:
    std::string fname;
    std::string lname;
    int         year;
  public:
    ...
    friend class CustomerHash;
};

class CustomerHash
{
  public:
    std::size_t operator() (const Customer& c) const {
      return hash_val (c.fname, c.lname, c.year);
    }
};
```

C++





josuttis | eckstein IT communication

## Emplace Functions

- **STL containers now provide emplace() functions**
- **They allow to pass the values for initialization instead of an initialized object to avoid unnecessary copies**

```
class Customer {
  private:
    string first;
    string last;
    int    year;
  public:
    Customer (const string& fn, const string& ln, int y) : first(fn), last(ln), year(y) {
    }
    friend ostream& operator << (ostream& strm, const Customer& c) {
        return strm << "[" << c.first << "," << c.last << "," << c.year << "]";
    }
};

int main()
{
    vector<Customer> cv1;                           // C++03 style:
    cv1.push_back(Customer("nico","josu",42));     // creates customer and copies it into cv1
    PRINT_ELEMENTS(cv1);                            // [nico,josu,42]

    vector<Customer> cv2;                           // with emplace functions:
    cv2.emplace_back("nico","josu",42);             // creates new customer inside cv2
    PRINT_ELEMENTS(cv2);                            // [nico,josu,42]
}
```

C++





josuttis | eckstein IT communication

## Emplace Functions

- **Emplace functions are:**
  - emplace_front(args...), emplace_back(args...)
    - correspond with push_front(), push_back()
  - emplace_after()
    - for forward_list
  - emplace(args...), emplace(pos,args...), emplace_hint(pos,args...)
- **There is an inconsistency between insert() and emplace():**
  - insert(pos,val) is a general function to insert val at pos
    - for associative containers pos is taken as a hint
  - emplace(pos,args...) is a mess:
    - For sequential containers there is provided:
      - emplace(pos,args...)
    - For associative containers there is provided:
      - emplace(args...)
      - emplace_hint(pos,args...)
  - Can't implement a generic function with emplace() for all containers



## Emplace Functions

- **Can't implement a generic function with emplace() for all containers**
  - For sequential containers the first argument is the position
  - For associative containers the first argument is the first value to initialize the element

```
template <typename T>
void doEmplace (T& cont)
{
    cont.emplace(cont.begin(),"nico","josuttis",42);     // dangerous!
}
```

**Sometimes this is the position for the new element.**
**Sometimes this is the first argument to initialize the element**

## 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;

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

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

**Class pair<> has only 8 constructors**

C++ 53 josuttis | eckstein IT communication



## Contact

**Nicolai M. Josuttis**

**www.josuttis.com**
**nico@josuttis.com**


C++ 54 josuttis | eckstein IT communication