C++ 2011

Dietmar Kühl
Bloomberg L.P.

Copyright Notice

Overview

- C++ committee and its operation
- language improvements
- r-value references and move semantics
- auto, decltype, and lambda functions
- concurrency support
- library enhancements

C++ 2011 Availability

- most compilers support some new features
- typically some compile-time flag is used:
    - gcc: -std=c++11 <http://gcc.gnu.org/>
    - clang: -std=c++11 <http://clang.llvm.org/>
    - EDG: --c++0x
    - MS VC++: enabled by default

Language Improvements

- final/override, defaulted/deleted functions
- initialisation: uniform, in class definition, lists
- delegating constructors, explicit conversion
- nullptr, static assertion, constant expression
- strongly typed and forward declared enums
- integer types, string literals

Stopping Inheritance

- prevent a class from being a base class:
  struct not_a_base final { ... };
  - final is part of the class definition
- prevent overriding of a virtual function:
  struct base { virtual void f(); ... };
  struct derived: base { void f() final; ... };
- final is not a keyword

Overriding Functions

- catch mistakes when overriding functions

```
struct B { virtual void f(int); ... };
struct D1: B { void f(int) override; }; // OK
struct D2: B { void f(long) override; }; // error
struct D3: B { void f(long); //hiding intentional?
```

- override is not a keyword

Defaulted Functions

- define a function like it would be generated

```
struct S {
  S();
  S(int); // => no implicit default constructor
  virtual S& operator=(S const&) = default;
};
S::S() = default; // non-inline definition
```

Defaultable Functions

- default constructor: S::S()
- copy constructor: S::S(S *cv*&)
- move constructor: S::S(S&&)
- copy assignment: S::operator= (S *cv*&)
- move assignment: S::operator= (S&&)
- destructor: S::~S()

Deleted Functions

- prevent use of functions:

```
struct B { void foo(); };
struct S: B {
  S(S const&) = delete;
  void operator= (S const&) = delete;
  void foo() = delete;
  void just_for_the_fun_of_it() = delete;
};
bool operator== (S, S) = delete;
```

Uniform Initialisation

- Braces {} can be used for ctor arguments:

```
struct S { S(int i = 0): m{i} {} int m; };
void f() {
  S s0{};
  S s1{1};
};
struct A { A(): a{0, 1, 2, 3} {} int a[4]; };
```

Uniform Initialisation

- narrowing conversion is an error, e.g.:

```
long   l{17};
short s{2}; // ERROR: narrowing conversion
```

- doesn't suffer from most vexing parse:

```
typedef istream_iterator<int> in_it;
vector<int> f(in_it(cin), in_it());
vector<int> v{in_it(cin), in_it()};
```

- f is a function, v is a vector<int>

Initialiser Lists

- let functions take argument lists:

```
#include <initializer_list>
void f(std::initializer_list<int> list);
f({ 0, 1, 2, 3 });
std::vector<int> v{ 0, 1, 2, 3 };
std::list<int>       l{ 0, 1, 2, 3 };
```

- std::initializer_list<T> is similar to container

std::initializer_list<T>

- same type for different sizes (unlike arrays)
- all members are const and noexcept
- usual container typedefs:
  value_type, reference, const_reference,
  size_type, iterator, const_iterator
- default constructor
- begin(), end(), size()

Member Initialisation

- members can be initialised where defined
- member initialiser list takes precedence

```
struct S {
  std::string s_ = “hello”;
  S() {}                         // s_ == “hello”
  S(std::string s): s_{s} {}     // s_ == s
};
```

Delegating Constructors

- constructors can delegate to others:

```
struct S {
  S(int i);
  S(std::string s): S{atoi(s.c_str())} {}
  S(): S{“17”} {} // yes, a bit silly...
};
```

Inheriting Constructors

- ctors can be inherited from the direct base
- the class can still add members and ctors

```
struct B { B(int); B(std::string); ... };
struct D: B {
  using B::B;
  D(): B{17}, v_(0) {}
  int v_ = 1;
};
```

Explicit Conversion

- conversion operators can be made explicit

```
struct some_ptr {
  explicit operator bool() const;
  ...
};
some_ptr p = ...;
if (p) ...                // OK: explicitly bool
int v(p << 3);            // no implicit conversion
```

Null Pointer

- nullptr converts only to pointer types
- nullptr is a keyword
- will become the spelling for null pointers
- nullptr_t defined in <cstddef>

```
int* p = nullptr;
```

Static Assert

- produce conditional compile-time errors
- condition is a constant expression
- most useful in template code
- useful to detect restrictions early on

```
static_assert(sizeof(int) < sizeof(long),
              "int isn't big enough");
```

Constant Expressions

- constant expressions extended to functions
- have to be declared as constexpr
- only one statement: a return
- can only use constant expressions
- arguments CEs => result CE
- definition has to be visible at point of use

Constexpr Function

```
enum BM { a = 0x01, b = 0x02 };
BM constexpr operator| (BM x, BM y) {
  return BM(int{x} | int{y});
}
void something(BM x) {
  BM y = a | b;
  switch (x) { case a | b: ...; }
}
```

Constexpr Value

- values can be constant expressions

```
struct S {
    static int constexpr       iv = 17;
    static double constexpr dv = 3.14;
};
```

- requires initialization (unlike const)
- requires a definition when address is used

Constexpr Constructor

- objects of user type can be constexpr

```
struct S {
  constexpr S(int v): v_{v} {}
  int v_;
};
S constexpr o(17);
enum { value = o.v_ };
```

Strongly Typed Enums

- enums have loads of problems:
  - they litter the enclosing namespace
  - implicitly convert to integer types
  - use an unpredictable underlying type
- the fix: strongly typed enums

Strongly Typed Enums

- declaration:
  enum class E { v1, v2, v3 };
  enum class F: int { v1, v2, v3 };
- no implicit conversion to integers
- values need qualification, e.g. E::v1 or F::v1
- the underlying integral type can be chosen (it is an error if the values don't fit)

Half-Hearted Enums

- for legacy uses strong enums may not work
- a migration path is supported, however:
  - the underlying integral type can be chosen
  - the scoped names work for all enums
  - nothing else changes

Forward Declare Enums

- enums can be forward declared
- requires the underlying type:
  ```
  enum class E: int;
  E e_val{17};
  enum class E: int { e1, e2 };
  ```
- works with strong and legacy enums

Integral Types

- 64 bit integers: long long/unsigned long long
- more character types (we had only 4):
  - UTF-16 encoded: char16_t
  - UTF-32 encoded: char32_t
  - <uchar.h>, <uchar> for some functions

Character Types

Type	encoding	char	string	class
char	?	‘a’	“a”	string
char	UTF-8	-	u8”a”	string
wchar_t	?	L’a’	L”a”	wstring
char16_t	UTF-16	u’a’	u”a”	c16string
char32_t	UTF-32	U’a’	U”a”	c32string

Raw String Literals

- all string literals have a “raw” version
- inside the raw literal no escapes matter
- start and end use same delimiter:
  R”abc(content)abc” - abc is the delimiter
  R”(content)” - no delimiter

R-Value References

- references to objects about to disappear
- move construction and moving objects
- using moves vs. exceptions

First: L-Values

- l-values are objects you can refer to
  - anything which has a name (including its aliases i.e. references)
  - objects being pointed to
- l-values may be const or volatile
- not all objects are l-values

Not L-Values

- unnamed temporary objects are not l-values
  - any unnamed object created
  - the object returned from a function
  - intermediate objects in an expression
- about to disappear when getting hold of them

R-Value References

- T&& x is an r-value reference for type T
- r-value references can only bind to r-values
  - the object is about to go away
  - x has name
  - .. and hence x binds to l-value references!

Move Construction

- consider the constructor: T(T&& obj)
- the referenced object is about to go away
- it is safe to move its state to *this
- ... and leave obj in a destructible state
- ... which doesn't release anything

Move Example

```
struct S {
    S(int i): p_{new int{i}} {}
    S(S&& other): p_{other.p_} { other.p_ = 0; }
    ~S() { delete this->p_; }
    ...
private:
    int* p_;
};
```

Overload Resolution

Overload	l-value	const l-value	r-value
	T x = ...; f(x);	T const x = ...; f(x);	f(T{...});
f(T&)	preferred	never	never
f(T const&)	OK	OK	OK
f(T&& x)	never	never	preferred

Move Constructor Use

```
T f() { return T{}; }            // move construction
T g() { T x{ ... }; return x; } // move construction

void h() {
    T r{f()};                    // move construction
}
```

(at least conceptually - probably elided, though)

std::move()

- std::move() allows moving l-values
- std::move() doesn't actually move
- creates r-value reference for r- and l-values
- returning an l-value from a function:

```
T x = ...;T y = ...;
return std::move(cond? x: y);
```

std::move() Caveat

- std::move(x) promises that x isn't used again
    - x is about to be destroyed or
    - x is about to be assigned to
- either way its state can be moved
- lying about this may be fatal
- thus: use std::move(x) carefully!

Move Assignment

```
struct S {
    S(S&& o): p_{o.p_} { o.p_ = 0; }
    S& operator= (S&& o) {
        delete this->p_;
        this->p_ = o.p_; o.p_ = 0; return *this;
    }
    ...
private:
    int* p_;
};
```

Moving Objects

- e.g. make/fill space in a std::vector<T>:

```
for (; i != v.size(); ++i)
    v[i - 1] = std::move(v[i]);
```

- uses move assignment
- std::move(begin, end, to) does this
- but what happens if moving throws?

Move vs. Copy

- moving or copying resources may fail
- copying objects often supports roll-back
- moving objects doesn’t support roll-back: the moved from objects are unusable
- only non-throwing move instead of copy
- in most cases move doesn’t throw

Moving vs. Exceptions

- moving is an optimisation in many places
- copying yields strong exception guarantee
- moving needs has to give same guarantee
- necessary to tell if something might throw
- need to declare if something might throw

noexcept Declaration

- mark that a function never throws, e.g.:
  `void f() noexcept;`
- optionally uses a bool constant expression:
  ```
  void f() noexcept(true);   // never throws
  void g() noexcept(false);  // might throw
  void h() noexcept(sizeof(int) <= 2);
  ```
- similar to throw() but no run-time overhead

noexcept Expression

- test if an expression will never throw e.g. if (noexcept(f())) ...
- this is a constant expression
- the argument is not evaluated
- tests if the overall expression might throw
- this includes "invisible" operations (e.g. construction/destruction of temporaries)

noexcept Expression

- consider noexcept(T{std::move(x)})
- x won't throw
- std::move(x) is declared be noexcept
- T{...} actually does two things:
  - (move?) constructs a T object
  - destroys the T object just created

noexcept Example

```
template <typename T>
void swap(T& t0, T& t1)
  noexcept(noexcept(T{std::move(t0)})
        && noexcept(t0 = std::move(t1))) {
  T tmp{std::move(t0)};
  t0 = std::move(t1);
  t1 = std::move(tmp);
}
```

noexcept Caveats

- noexcept is not statically enforced
- noexcept function throws => terminate
  - similar to using throw()
  - no guarantee on stack unwinding
- destructors implicitly become noexcept!
  - throwing dtor needs noexcept(false)

Uses of Moving

- for non-copyable or expensive to copy types
  - putting objects into containers
  - returning objects
  - passing objects around
- optimising reorganisations (when noexcept)

Generated Move

- move operation may be compiler generated
  - if no user defined copy, dtor, or move
  - if all the members are movable
  - user implementation can use = default
    (if all members are movable)
- generated operations move all subobjects

Automatic Types

- deduce types of initialised variables:
  auto var1(some_expression);
  auto var2 = some_expression;
- breaks some existing code:
  auto int i;
  auto int j(17);
- the fix: remove auto or the type

Automatic Types

- can be used where defining a variable
- cannot be used for members or arguments
- derived types can be deduced, too:

```
auto        x = 17;
auto const& c = 17;
auto*       p = &c;
```

- same rules as for template arguments

Type Deduction

	auto	auto&	auto const&	auto&&
S x; ... a = x;	S	S	S	S&
S const c; ... a = c;	S	error	S	S const&
... a= S{};	S	error	S	S

Type of Expression

- decltype(expression)
- can be used anywhere a type can be used
- does not evaluate the expression
- yields the exact type of the expression
- use type traits to manipulate the type

Type Traits

- compile-time inspection of types
- declared in <type_traits>
  - made accessible via a library interface
- various Boolean values to check properties
- various type transformation
- used e.g. to improve algorithm behaviour

Example Type Traits

is_abstract	is_integral	is_enum
is_object	is_fundamental	is_trivial
is_constructible	is_polymorphic	is_class
is_destructible	is_copy_constructible	is_pod
is_base_of	has_virtual_destructor	is_same
remove_const	add_lvalue_reference	make_signed

decltype Examples

```
typedef decltype(0) int_type;
decltype('c') f();

struct S { decltype(f()) m_; };

std::vector<decltype(f())> v(10, f());

// below doest not work (x isn't declared, yet):
template <typename T>
decltype(x * x) g(T x) { return x * x; }
```

Function Declaration

- alternate function declaration syntax:
  auto name(type arg) -> result;
- auto main(int ac, char* av[]) -> int { ... }
- template <typename T>
  auto square(T x) -> decltype(x * x) { ... }

Lambda Functions

- [capture](arguments) mutable -> type { ... }
- capture: how are variables referenced
- opt: arguments: normal argument declaration
- opt: mutable: can mutate captured variables?
- opt: type of the result

Lambda Example

```
std::string v("hello, world");
std::transform(v.begin(), v.end(), v.begin(),
 [](unsigned char c){ return toupper(c); });
```

Return Type

- defined using new style syntax
- the return type can be omitted sometimes:
  - if there is no return statement ⇒ void
  - if there is one return only ⇒ deduced
  - otherwise return needs to be specified

Lambda Arguments

- no arguments ⇒ can omit their declaration:
  - auto lambda1 = []( ){ return 17; }
  - auto lambda2 = []{ return 17; }
- argument types can not be deduced:
  - each type needs to be spelled out

Capture

- only names from the local scope!
- specify storage of objects in the closure
- [] ⇒ no variables are to be captured
- [&] ⇒ default capture by reference
- [=] ⇒ default capture by value
- [&r, v] ⇒ r captured by reference, v by value

Capture Example

```
int n{17}; string* s{new string};
auto a=[=]{ return n; };            // int n
auto b=[&]{ return n; };           // int& n
auto c=[=]{ return s->size(); }; // string* s
n = 18; *s = “hello”;
int a_result{a()};   // -> 17
int b_result{b()};   // -> 18
auto c_result = c(); // -> 5
```

Capture Concept

- captured variables are effectively a closure
- a lambda capturing any variables is like a function object with the variables as members
- capture specification: value/reference
- type of the members is deduced

Mutable Closure

- by default the closure is constant:
  int a{18};
  auto lambda = [=]{ a = 17; }; <- ERROR
- closure can be made mutable:
  int a{18};
  auto lambda = [=]() mutable { a = 17; };
- not needed for reference capture [&]

Lambda Details

- members cannot be captured
  - the pointer this is captured
- the type of lambdas cannot be spelled out
  - ... but they convert to function pointers if there is no closure:
    `int (*lambda)() = []{ return 17; };`

Range-Based For

- execute a block for all elements in a range

```
for (auto x: range)
    use(x);
```

- this is equivalent to

```
auto&& r{range};
for (auto b(begin(r)), e(end(r)); b != e; ++b) {
    auto x(*b);
    use(x);
}
```

Going Native 2012

- conference at Microsoft in February 2012
- recorded and available on the net
- http://channel9.msdn.com/Events/GoingNative/GoingNative-2012

Concurrency

- threads
- mutex, locks, and condition variable
- futures
- atomics
- thread safety

Threads

- std::thread to kick off a thread
  - detached to finish off at some point
  - joinable to synchronise on results
- no support for thread pools, etc. (yet?)

Create a Thread

```
#include <thread>
void entry(T0 arg0, T1 arg1);

void f(T0 arg0, T1 arg1) {
    std::thread thread{entry, arg0, arg1};
    ... // <- this is dangerous!
    thread.join();
}
```

std::thread's Destructor

- a std::thread is joinable() until
  - join() is called to synchronise with it
  - detach() is called to make it independent
- destroying a joinable() thread:
  - calls std::terminate()

Futures

- used to wait for results

```
std::vector<int> foo(int bar) {
    std::future<std::vector<int> > future
        = std::async(std::launch::async,
                     some_function, bar);
    do_work();
    return future.get();
}
```

Promises

- std::future<T> objects may have other source
- std::promise<T> collects results from thread(s)
  - std::future<T>::get() to access results
  - one or more threads set the result
  - ... which may be an exception

Obtaining Exceptions

- std::exception_pointer to hold an exception
- std::current_exception() to get exception
  - null if there is no current exception
  - current exception or copy of it otherwise
- std::make_exception_ptr(e)
- std::rethrow_exception(ptr)

Mutex

- synchronise access to shared data
- support lock(), try_lock(), unlock() operations
- std::mutex
- std::recursive_mutex
- std::timed_mutex
- std::recursive_timed_mutex

Lock Objects

- std::lock_guard<Mutex> lock(mutex);
  - acquire lock in ctor: mutex.lock()
  - release lock in dtor: mutex.unlock()
- std::unique_lock<Mutex>
  - allow later lock acquisition, try_lock(), ...
  - support transfer of lock, etc.

Condition Variables

- std::condition_variable
  - requires std::unique_lock<std::mutex>
  - most efficient
- std::condition_variable_any
  - can be used with user-defined locks

Waiting for Conditions

- cv.wait(lock): wait until notified
- cv.wait(lock, predicate): wait until notified (possibly multiple times) and predicate()
- cv.notify_one(): notify one waiting thread
- cv.notify_all(): notify all waiting threads

Atomic Operations

- work only on a few basic types atomically
- atomic only for each individual object
- only synchronise access to this one object
- more expensive than normal data accesses
- ... but cheaper than full synchronisation
- ... on systems supporting atomic operations

std::atomic<T>

- is_lock_free() to test if doesn't use a lock
- store(T), load() to change/read the value
- compare_exchange_...(): various forms
- can also use assignment and conversion
- for integral types also integer operations
- function interface to ease C compatibility

Thread-Safety

- standard classes are thread-safe:
  - multiple concurrent readers
  - one writer, no readers
  - has to be guaranteed via external locks (except for the synchronisation classes)
- applies to containers, streams, etc.

Fire and Forget

- implicit locking has limited use
- only applicable to fire-and-forget interfaces:
  - allocate memory, release memory
  - push on/pop from queue
  - ... and similar abstractions
- doesn’t work well with intermediate state

Stream Objects

- std::cout, std::cin, etc. are not special
- don't use them directly without locking
- typically accessed via some logging facility
    - combines result of formatting operations
    - makes the actual send atomic

Library Enhancements

- general improvements of the library
- support for function objects
- smart pointers
- added containers

Using the Language

- where possible, objects are moved
- appropriately defaulted or deleted functions
- take initialiser-lists where appropriate
- constructors objects in-place in containers
- functions are noexcept where appropriate

Overall Library Policies

- allow const iterators for non-const objects
- allocators are cleaned up
- allocators are used consistently throughout

std::function<Sig>

- function<R (T0, T1, T2)> f{x};
- holds an object callable with the signature
  - accepts arguments of type T0, T1, T2
  - result can be returned as R
- similar to R(*)(T0, T1, T2) with an object
- internally holds a polymorphic entity

std::mem_fn()

- function object from member function
- first parameter is the object
  - pointers or references can be used
- std::for_each(begin, end, mem_fn(&S::f));

std::bind()

- binds function arguments to positions
- allows composition of function objects
- reference_wrapper for pass by reference
- references and pointers for members
- replaces bind1st, ptr_fun, mem_fun, ...

bind() and Placeholders

```
#include <functional>
using namespace std::placeholders;
void f(int a, int b);
int main() {
  bind(f, _1, _2)(1, 2);
  bind(f, _2, _1)(1, 2);
  bind(f, _2, 3)(1, 2, 3, 4, 5);
  bind(f, _2, 3)(1); // placeholder index too big
  bind(f, _1)(1);    // too few arguments
}
```

bind() and Composition

```
int g(int a);
void f(int b, int c);

int main() {
  bind(g, _1)(1);                     // g(1)
  bind(f, bind(g, _1), _2)(1, 2);  // f(g(1), 2);
  bind(f, bind(g, _1), _1)(1);      // f(g(1), 1);
}
```

bind() and References

```
void f(int& r);

int main() {
  int v{0};
  bind(f, v)();          // ERROR: not a reference
  bind(f, ref(v))();  // OK
  bind(f, _1)(v);     // OK
}
```

reference_wrapper<T>

- T x...; reference_wrapper<T> rw{x};
- implicitly converts to T& but is a value type
- if T is function, rw can be used as function
- ref(x) → reference_wrapper<T>
- cref(x) → reference_wrapper<T const>
- for_each(begin, end, ref(x));

Smart Pointers

- std::unique_ptr<T> moves ownership
  - std::auto_ptr<T> is deprecated
- std::shared_ptr<T> reference counted
  - std::weak_ptr<T> to break cycles
- std::exception_ptr for exception objects

Added Containers

- note: “container” used here informally
- mandatory template arguments are marked
- std::array<T, N>
- std::forward_list<T, A>
- std::unordered_set<K, H, E, A>
- std::unordered_map<K, V, H, E, A>

std::tuple<typename...>

- container for heterogenous elements
- like std::pair<T0,T1> but more general:
  - there can be any number of elements
  - reference members are supported
- elements are referred to by index

Tuple Example

```
tuple<int, double, string> t{1, 3.14, “hello”};

int i{get<0>(t)}; double d{get<1>(t)}; string s;
get<0>(t) = i;

t = tuple<long, float, char const*>{2, 2.71, “x”};

tie(i, d, s) = make_tuple(3, 4.0, string(“y”));
```

Lexicographic Compare

- std::tuple<...> defines relational operators
- these can be used for other classes:

```
struct V { int i; double d; string s; };

bool operator< (V0 const& v0, V1 const& v1) {
    return tie(v0.i, v0.d, v0.s) < tie(v1.i, v1.d, v1.s);
}
```

Conclusions

- lots of new features
- C++ should be easier to use
- C++ has become more complex
- ... but should be easier to use