An Overview of C++11/14

Leor Zolman
BD Software
www.bdsoft.com
leor@bdsoft.com

September 8th,2014

Agenda

- C++ timeline
- Goals for the new C++
- Part I. Simpler language changes
- Part II. New facilities for class design
- Part III. Larger new language features
  - Initialization-related improvements
  - Rvalue references, move semantics and perfect forwarding
  - Lambdas
- Most new language features are at least mentioned

2

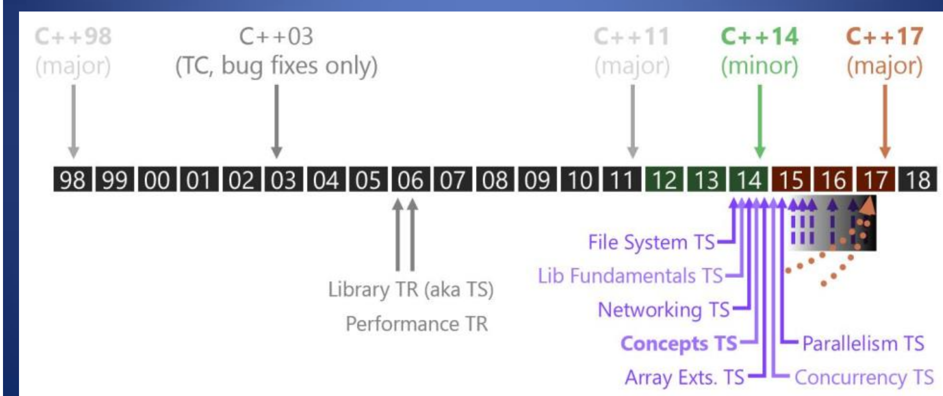
About the Code Examples

- When practical, I show specific problems/issues in Old C++ and then introduce the C++11/14 solutions
- Examples are not all 100% self-contained
  - Read the code *as if* the requisite `#includes`, `using`s, `std::`s etc. were there ☺

3

ISO C++ Timeline

(From Herb Sutter's Fall '03 ISO C++ Standards Meeting Trip Report, 10/3/13)



4

Goals for C++11

- Make C++ easier to teach, learn and use
- Maintain backward-compatibility
- Improve performance
- Strengthen library-building facilities
- Interface more smoothly with modern hardware

5

"The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever."

-Bjarne Stroustrup [from his C++11 FAQ]

6

Part I: The Simpler Core Language Features

- `auto`, `decltype`, trailing return type
- `nullptr`
- Range `for`
- `>>` in template specializations
- `static_assert`
- `extern template`
- `noexcept`
- Variadic templates (OK, maybe *not* so simple)
- `constexpr`, template alias, and more…

7

Problem: Wordy declarations

```
// findNull: Given a container of pointers, return an
// iterator to the first null pointer (or the end
// iterator if none is found)

template<typename Cont>
typename Cont::const_iterator findNull(const Cont &c)
{
	typename Cont::const_iterator it;
	for (it = c.begin(); it != c.end(); ++it)
		if (*it == 0)
			break;

	return it;
}
```

8

Using findNull in Old C++

```
int main()
{
    int a = 1000, b = 2000, c = 3000;
    vector<int *> vpi;
    vpi.push_back(&a);
    vpi.push_back(&b);
    vpi.push_back(&c);
    vpi.push_back(0);

    vector<int *>::const_iterator cit = findNull(vpi);
    if (cit == vpi.end())
        cout << "no null pointers in vpi" << endl;
    else
    {
        vector<int *>::difference_type pos = cit - vpi.begin();
        cout << "null pointer found at pos. " << pos << endl;
    }
}
```

9

Using findNull in C++11

```
int main()
{
    int a = 1000, b = 2000, c = 3000;
    vector<int *> vpi { &a, &b, &c, 0 };

    auto cit = findNull(vpi);

    if (cit == vpi.end())
        cout << "no null pointers in vpi" << endl;
    else
    {
        auto pos = cit - vpi.begin();
        cout << "null pointer found in position " <<
                     pos << endl;
    }
}
```

10

Problem: What's the Return Type?

- Sometimes a return type simply cannot be expressed in the usual manner:

```
// Function template to return  product of two
// values of unknown types:

template<typename T, typename U>
??? product(const T &t, const U &u)
{
      return t * u;
}
```


`decltype` and Trailing Return Type

- In this case, a combination of `auto`, `decltype` and *trailing return type* provide the only solution for C++11:

```
// Function template to return  product of two
// values of unknown types:

template<typename T, typename U>
auto product(const T &t, const U &u) -> decltype (t * u)
{
      return t * u;
}
```

`findNull` in C++11
(First Cut)

```
// findNull: Given a container of pointers, return an
// iterator to the first null pointer (or the end
// iterator if none is found)

template<typename Cont>
auto findNull(const Cont &c) -> decltype(c.begin())
{
	auto it = c.begin();
	for (; it != c.end(); ++it)
		if (*it == 0)
			break;
	return it;
}
```

13

Non-Member `begin/end`

- New forms of `begin()` and `end()` even work for native arrays, hence are more generalized

```
bool strLenGT4(const char *s) { return strlen(s) > 4; }

int main()
{                           // Applied to STL container:
    vector<int> v {-5, -19, 3, 10, 15, 20, 100};
    auto first3 = find(begin(v), end(v), 3);

    if (first3 != end(v))
       cout << "First 3 in v = " << *first3 << endl;
                            // Applied to native array:
    const char *names[] {"Huey", "Dewey", "Louie"};
    auto firstGT4 = find_if( begin(names), end(names),
                                  strLenGT4);
    if (firstGT4 != end(names))
       cout << "First long name: " << *firstGT4 << endl;
}
```

14

Non-Member begin/end Variations in C++14

- Return const_iterators:
  - cbegin/cend
- Return reverse_iterators:
  - rbegin/rend
- Return const_reverse_iterators:
  - crbegin/crend

```
template <typename Container>
void process_container(Container &c)     // Note: no const
{
    typename C::const_iterator ci = begin(c);  // C++11

    auto ci2 = cbegin(c);                      // C++14

    . . .
}
```

15

Problem: Null Pointers

- In Old C++, the concept of “null pointers” can be a source of confusion and ambiguity
  - How is NULL defined?
  - Does 0 refer to an int or a pointer?

```
void f(long) { cout << "f(long)\n"; }
void f(char *) { cout << "f(char *)\n";}

int main()
{
     f(0L);             // calls f(long)
     f(0);              // ERROR: ambiguous!
     f(static_cast<char *>(0)); // Oh, OK…
}
```

16

nullptr

- Using `nullptr` instead of 0 disambiguates:

```
void f(long) { cout << "f(long)\n"; }
void f(char *) { cout << "f(char *)\n";}

int main()
{
      f(0L);            // as before, calls f(long)
      f(nullptr);       // fine, calls f(char *)
      f(0);             // still ambiguous
}
```

17

findNull in C++11 (Final C++11 version)

```
template<typename Cont>
auto findNull(const Cont &c) ->
                    decltype(begin(c))
{
     auto it = begin(c);
     for (; it != end(c); ++it)
          if (*it == nullptr)
               break;

     return it;
}
```

18

Generalized Function Return Type Deduction in C++14

- C++14 allows return type to be *deduced* from the return expression(s) used:

```
template<typename Cont>
auto findNull(const Cont &c)    // don't need decltype!
{
      auto it = begin(c);
      for (; it != end(c); ++it)
            if (*it == nullptr)
                  break;

      return it;         // return type deduced HERE
}
```

19

C++14: `auto` vs. `decltype(auto)`

- There are actually two approaches to function return type deduction in C++14
  - Functions declared to return `auto` (or "decorated" `auto`)
    - Employs *template* type deduction rules
      - Discards references, `const`, `volatile` from return expression's type (may add it back when `auto` is decorated)
  - Function declared to return `decltype(auto)`
    - No decoration permitted
    - Employs `decltype` type deduction
      - Expression's actual type is the return type

20

Iterating Over an Array or Container in Old C++

```
int main()
{
    int ai[] = { 10, 20, 100, 200, -500, 999, 333 };
    const int size = sizeof ai / sizeof *ai;    // A pain

    for (int i = 0; i < size; ++i)
        cout << ai[i] << " ";
    cout << endl;

    list<int> li ( ai, ai + size);
            // Note opportunities for typos, off-by-1's, etc.
    for (list<int>::iterator it =
                    li.begin(); it != li.end(); ++it)
        *it += 100000;
                                    // Same rigmarole here:
    for (list<int>::const_iterator it =
                    li.begin(); it != li.end(); ++it)
        cout << *it << " ";
}
```

21

Improvement: Range-Based `for` Loop

```
int main()
{
    int ai[] { 10, 20, 100, 200, -500, 999, 333 };

    for (auto i : ai)       // Don't need size
        cout << i << " ";
    cout << endl;

    list<int> li (begin(ai), end(ai));
    for (auto &elt : li)    // Using a ref allows modifying
        elt += 10000;

    for (const auto &elt : li) // Note: const & not nec-
        cout << elt << " ";     // essarily better for ints
    cout << endl;

    for (auto i : { 100, 200, 300, 400 })
        cout << i << " ";
}
```

22

The “>> Problem”

- Old C++ requires spaces between consecutive closing angle-brackets of nested template specializations:

```
map<string, vector<string> > dictionary;
```

- C++11 permits you to omit the space:

```
map<string, vector<string>> dictionary;
```

- That’s one less *gotcha*

23

Compile-Time Assertions: `static_assert`

- The C library contributed the venerable `assert` macro for expressing run-time invariants:

```
int *pi = …;
assert (pi != NULL);
```

- C++11 provides direct language support for *compile-time* invariant validation and diagnosis:

```
static_assert(condition, “message”);
```

- Conditions may only be formulated from *constant* (compile-time determined) expressions

24

static_assert

```
static_assert(sizeof(int) >= 4,
      "This app requires ints to be at least 32 bits.");

template<typename R, typename E>
R safe_cast(const E &e)
{
   static_assert(sizeof (R) >= sizeof(E),
            "Possibly unsafe cast attempt.");
   return static_cast<R>(e);
}

int main()
{
   long lval = 50;
   int ival = safe_cast<int>(lval);   // OK iff long & int
                                      //    are same size
   char cval = safe_cast<char>(lval); // Compile error!
}
```

25

Problem: Object File Code Bloat From Templates

- The industry has settled on the "template inclusion model"
  - Templates fully defined in header files
  - Each translation unit (module) `#includes` the header: all templates are instantiated in *each* module which uses them
  - At link time, all but one instance of each redundant instantiated function *is discarded*

26

The Failed Solution: `export`

- Old C++ introduced the `export` keyword
- The idea was to support *separately compiled templates*
- But even when implemented (AFAIK only EDG accomplished this), *it didn't really improve productivity*
  - Templates are just too complicated
    - …due to two-phase translation

27

The C++11 Solution: `extern template`

- Declare a class template specialization `extern` and the compiler will not instantiate the template's functions in that module:

```
#include <vector>
#include <Widget>
extern template class vector<Widget>;
```

- For `vector<Widget>`, the *class definition* is generated if needed (for syntax checking) but member functions are not instantiated
- Then, in just *one* (`.cpp`) module, *explicitly instantiate* the class template:

```
template class vector<Widget>;
```

28

Problem: Dynamic Exception Specifications

- In Java, all exception specifications are *enforced*
- Old C++ functions specify exceptions they might throw via *dynamic exception specifications*…but callers need not acknowledge them!
- Plus, how can function *templates* possibly know what exceptions might be thrown?
- Thus the only dynamic exception specification used in the Old C++ standard library is the *empty* one:

```
template<typename T>
class MyContainer {
public:
        …
        void swap(MyContainer &) throw();
    …
```

29

The C++11 Way: `noexcept`

- Dynamic exception specifications (even empty ones) can impact performance
- C++11 *deprecates* dynamic exception specifications and introduces the `noexcept` keyword:

```
template<typename T>
class MyContainer {
public:
     …
     void swap(MyContainer &) noexcept;
     …
```

- If an exception tries to escape from a `noexcept` function, the program immediately terminates.

30

Conditional `noexcept`

- `noexcept` clauses can be conditional on the "noexcept" status of sub-operations

```
// From the g++ standard library's implementation
// of std::pair (simplified):
template<class T1, class T2>
class pair
{
   // Defines data members 'first', 'second', etc.

   void swap(pair& __p)
      noexcept(noexcept(swap(first, __p.first))
           && noexcept(swap(second, __p.second)))
   {
       using std::swap;
       swap(first, __p.first);
       swap(second, __p.second);
   }
}
```

31

Problem: How Do You Write a Function to Average N Values?

- You can use C variadic functions:

```
int averInt(int count, ...);
double averDouble(int count, ...);
```

  - Must write one for each type required
  - Must provide the argument count as 1st arg
  - Type safety? Fuggedaboudit…
- Can't use C++ default arguments
  - Because we can't know the # of actual args
- Could use overloading and templates
  - That's ugly too

32

Variadic Templates

```
// To get an average, we 1st need a way to get a sum…

template<typename T> // ordinary function template
T sum(T n)           // for the "terminal" case
{
      return n;
}
                     // variadic function template:
template<typename T, typename... Args>
T sum(T n, Args... rest)       // "parameter packs"
{
      return n + sum(rest...);
}

int main() {
      cout << sum(1,2,3,4,5,6,7);
      cout << sum(3.14, 2.718, 2.23606);
};
```


A Subtle Problem With `sum`

- Note that if `sum` is called with a mixture of different argument types, the wrong return type may result:

```
template<typename T, typename... Args>
T sum(T n, Args... rest) // T is the FIRST type
{
      return n + sum(rest...);
}     // sum(1, 2.3) yields 3 (instead of 3.3)
```

- To fix this, first we need `auto`, trailing return type, etc.:

```
template<typename T, typename... Args>
auto sum(T n, Args... rest) -> decltype(n + sum(rest...))
{
      return n + sum(rest...);
}
```

- Unfortunately, *that* doesn't compile ☹ ☹

An Even *More* Subtle Problem

- The recursive reference to `sum` in the `decltype` expression is illegal because the compiler doesn't know the full type of `sum` (haven't reached the end of its header line yet…)
- A surprising exception to this restriction: when the function name is of a *member* function (I kid you not) it is OK if it is still "incomplete"
- So to make it all work, just make the `sum` function templates be `static` functions of a struct!

35

Finally, a Working `sum`

```
struct Sum
{
	template<typename T>
	static T sum(T n)
	{
		return n;
	}

	template<typename T, typename... Args>
	static auto sum(T n, Args... rest) ->
		decltype(n + sum(rest...))
	{			// NOW the recursion is OK…
		return n + sum(rest...);
	}
};
```

36

And, At Last: Average

- Another variadic function template can leverage the `sum()` templates, and variadic `sizeof...` operator, to give us average:

```
template<typename... Args>
auto avg(Args... args) ->
              decltype(Sum::sum(args...))
{
   return Sum::sum(args...) / (sizeof... (args));
}

cout << avg(2.2, 3.3, 4.4) << endl; // works!
cout << avg(2, 3.3, 4L) << endl;    // works too!
```

37

constexpr

- Enables compile-time evaluation of functions (including operators and constructors) when expressed in terms of *constant* expressions

```
const int SIZE = 100;             // simple constant

template<typename T>
constexpr T square(const T&x)     // constexpr function--
{                                 //  serves "double duty"
    return x * x;
}

Widget warray[SIZE];              // OK, as always
Widget warray2[square(SIZE)];     // OK (compile-time)

int val = 50;                     // NOT constant expr.
int val_squared = square(val);    // OK, run-time expr.
Widget warray3[square(val)];      // ERROR (not const.)
```

38

C++11 `constexpr` vs. C++14's

- Bodies of C++11 `constexpr` functions are essentially limited to a single return expression
- No other control structures allowed
  - No `if…else`
    - But `?:` can serve as a substitute
  - No loops
    - But recursion is supported
- C++14 relaxes most of C++11's restrictions
  - "Roughly: a `constexpr` function can contain anything that does not have side effects outside the function." *[--B. Stroustrup]*
  - Still forbidden: `goto`, `try` blocks, calling `non-constexpr` functions, a few other minor things

39

Template Alias

- The "template typedef" idea, w/clearer syntax:

```
template<typename T>
using setGT = std::set<T, std::greater<T>>;

setGT<double> sgtd { 1.1, 8.7, -5.4 };
// As if writing:
// std::set<double, std::greater<double>> sgtd {…
```

- `using` aliases also make a "better typedef":

```
typedef void (*voidfunc)();        // Old way
using voidfunc = void (*)();       // New way
```

40

Some String-Related Features

- Unicode string literals
  - UTF-8: `u8"This text is UTF-8"`
  - UTF-16: `u"This text is UTF-16"`
  - UTF-32: `U"This text is UTF-32"`
- Raw string literals
  - Can be clearer than lots of escaping:

```
string s = "backslash: \"\\\", single quote: \"'\"";
string t = R"(backslash: "\", single quote: "'")";
// Both strings initialized to:
//       backslash: "\", single quote: "'"

string u = R"xyz(And here's how to get )" in!)xyz";
```

41

Inline Namespaces

- Facilitates versioning
- Names in an `inline` sub-namespace are implicitly "hoisted" into the enclosing (parent) namespace

```
namespace bdsoft {
      namespace v1_0            // Old version
      { … }
      inline namespace v2_0     // Current/default version
      {
        …    // Everything in here behaves as if
             // declared directly in namespace bdsoft
      }
      namespace v3_0            // Experimental version
      { … }
}
```

42

Attributes

- Replaces #pragmas, __attribute__, __declspec, etc.
- E.g., `[[noreturn]]` to help compilers detect errors
- New in C++14: `[[deprecated]]`
  - Compiler issues warning if labeled entity used
  - Can be used for: functions, classes, typedefs, enums, variables, non-static data members and template *complete specializations*

43

More Language Features

- Scoped `enum`s (a.k.a. *enum classes*)
  - Enumerators don't "leak" into surrounding scopes
  - Fewer implicit conversions
  - Can specify the underlying (integral) type
    - This is true for old (un-scoped) `enum`s as well
- `long long`
  - 64-bit (at least) ints
- `alignas` / `alignof`
  - Query/ force boundary alignment

44

Yet More Language Features

- Generalized Unions
  - E.g., members of unions are now allowed to have constructors, destructors and assignment
    - However, any user-defined ctor, dtor or copy op is treated as if it were declared `=delete` (and thus cannot be used with an object of the union type)
- Generalized PODs
  - E.g., "Standard Layout Types" (PODs) can now have constructors
  - C++98 POD types are now subdivided into: PODs, trivially copyable types, trivial types and standard-layout types

45

Yet More Language Features

- Garbage Collection ABI
  - Sets ground-rules for gc; specifies an ABI. [Note: Actual gc is neither required nor supplied]
- User-defined Literals
  - Classes can define *literal operators* to convert from literals with a special suffix into objects of the class type, e.g.,

```
Binary b = 11010101001011b;
```

46

New for C++14

- Binary Literals
  - The new prefix `0b` (or `0B`) designates intrinsic binary literal values:

```
auto fifteen = 0b1111;
```

- Single-quote as Digit Separator:

```
auto filler_word = 0xdead’beef;
auto aBillion = 1’000’000’000;
```

47

Part II: Features Supporting Better Class Design

- Generated functions: `default` / `delete`
- Override control: `override` / `final`
- Delegating constructors
- Inheriting constructors
- Increased flexibility for in-class initializers
- Explicit conversion operators

48

Problem: How to Disable Copying?

- There are at least two Old C++ approaches to prevent objects from being copied:
  - Make the copy operations private:

```
class RHC      // some resource-hogging class
{     …
private:
      RHC(const RHC &);
      RHC &operator=(const RHC &);
};
```

  - Inherit privately from a base class that does it for you:

```
class RHC : private boost::noncopyable
{
      …
```

  - Both are problematic.

49

Now: =default, =delete

- These specifiers control function generation:

```
class T {
public:
      T() = default;
      T(const char *str) : s(str) {}
      T(const T&) = delete;
      T &operator=(const T&) = delete;
private:
      string s;
};

int main() {
      T t;               // Fine
      T t2("foo");       // Fine
      T t3(t2);          // Error!
      t = t2;            // Error!
}
```

50

`=delete` and Func. Overloading

- `=delete` also lets you restrict arg types:

```
void doSomething(int);
void doSometehing(long) = delete;
void doSomething(double) = delete;

int main()
{
        doSomething(10);        // Fine (direct)
        doSomething('c');       // Fine (conversion)
        doSomething(10L);       // ERROR
        doSomething(2.3);       // ERROR
        doSomething(2.3F);      // ERROR (!)
}
```

51 51

Problems With Overriding

- When limited to Old C++ syntax, the "overriding interface" is potentially misleading / error-prone:

```
class Base {
public:
        virtual void f(int);
        virtual void ff(int);
        virtual int g() const;
        void h(int);
};

class Derived : public Base {
public:
        void f(int);        // is this a virtual func.?
        virtual int g();    // meant to override Base::g?
        void ff(int);       // How to prevent further over.?
        void h(int);        // overrides Base::h? Or… ?
};
```

52

override / final

- C++11 (mostly) lets you say what you really mean:

```
class Base {
    public:
        virtual void f(int);
        virtual void ff(int);
        virtual int g() const;
        void h(int); // final      // Unfortunately, can't
};                                 // use on a non-virtual fn

class Derived : public Base {
    public:
        void f(int) override;      // Base::f MUST be virtual
        void ff(int) final;        // Prevents overriding!
        int g() override;          // Error!
        void h(int);               // SHOULD be an error…
};                                 // … but no help here

// Note: These are "CONTEXTUAL" keywords! Cool!
```

53

final Classes

- An entire class can be declared `final`:

```
class Base {};

class Derived final : public Base {   // Derived is final
        . . .
};

class Further_Derived : public Derived {   // ERROR!
        . . .
};
```

54

Problem: Old C++ Ctors Can't Use the Class' Other Ctors

```
class FluxCapacitor
{
public:
        FluxCapacitor() : capacity(0), id(nextId++) {}
        FluxCapacitor(double c) : capacity(c),
                id(nextId++) { validate(); }
        FluxCapacitor(complex<double> c) : capacity(c),
                id(nextId++) { validate(); }
        FluxCapacitor(const FluxCapacitor &f) :
                id(nextId++) {} // can  you spot the
                                // silent logic error?
//      ...
private:
        complex<double> capacity;
        int id;
        static int nextId;
        void validate();
};
```

55

C++11 Delegating Constructors

- C++11 ctors may call other ctors (à la Java)

```
class FluxCapacitor
{
public:
        FluxCapacitor() : FluxCapacitor(0.0) {}
        FluxCapacitor(double c) :
                FluxCapacitor(complex<double>(c)) {}
        FluxCapacitor(const FluxCapacitor &f) :
                FluxCapacitor(f.capacity) {}
        FluxCapacitor(complex<double> c) :
                capacity(c), id(nextId++) { validate(); }
                // Note: Now harder to forget to set capacity
private:
        complex<double> capacity;       // BUT... There's a
        int id;                         // subtle performance
        static int nextId;              // hit in this part-
        void validate();                // icular example...
};
```

56

Problem: Very Limited Data Member Initialization

- In old C++, *only* const static integral members could be initialized in-class

```
class FluxCapacitor
{
public:
        static const size_t num_cells = 50;  // OK
        FluxCapacitor(complex<double> c) :
                capacity(c), id(nextId++) {}
        FluxCapacitor() : id(nextId++) {}     // capacity??
        …
private:
        int id;
        complex<double> capacity = 100;  // ERROR!
        static int nextId = 0;           // ERROR!
        Cell FluxCells[num_cells];       // OK
};
```

57

C++11 In-Class Initializers

- Now, any non-static data member can be initialized in its declaration:

```
class FluxCapacitor
{
public:
        static const size_t num_cells = 50;     // still OK
        FluxCapacitor(complex<double> c) :
                capacity(c), id(nextId++) {}     // capacity c
        FluxCapacitor() : id(nextId++) {}        // capacity 100
        …
private:
        int id;
        complex<double> capacity = 100;          // Now OK!
        static int nextId = 0;                   // Still illegal
        Cell FluxCells[num_cells];               // Still OK
};
```

58

Inheriting Constructors

- C++11 derived classes may "inherit" most ctors (just not the default ctor) from their base class(es):
  - Simply extends the old `using Base::name` syntax to ctors (arbitrarily excluded in C++98)
  - New ctors may still be added
  - Inherited ones may be redefined

```
class RedBlackFluxCapacitor : public FluxCapacitor
{
public:
      enum Color { red, black };
      using FluxCapacitor::FluxCapacitor;
      RedBlackFluxCapacitor(Color c) : color(c) {}
      void setColor(Color c) { color = c; }
private:
      Color color { red };     // Note: default value
};
```

59

Explicit Conversion Operators

- In Old C++, only constructors (one form of user-defined conversion) could be declared `explicit`
- *Operator* conversion functions (e.g., `operator long()`) could not
- C++11 remedies that, *but…*

```
class Rational {
public:
      // …
      operator double() const;          // Iffy…
      explicit operator double() const; // Better…
      double toDouble() const;          // Best?
private:
      long num, denom;
};
```

60

Part III: Larger Language Features

- Initialization
  - Initializer lists
  - Uniform initialization
  - Prevention of narrowing
- Lambdas
- Rvalue references, move semantics, universal references and perfect forwarding (they're all related…)

61

Problem: Limited Initialization of Aggregates in Old C++

```
int main()
{                                   // OK, array initializer
      int vals[] = { 10, 100, 50, 37, -5, 999};

      struct Point { int x; int y; };
      Point p1 = {100,100};         // OK, object initializer

      vector<int> v = { 5, 29, 37};       // ERROR in Old C++!

      const int valsize = sizeof vals / sizeof *vals;

                                    // range ctor OK
      vector<int> v2(vals, vals + valsize);
}
```

62

Initializer Lists

- C++11's `std::initializer_list` supports generalized initialization of aggregates
- It extends Old C++'s array/object initialization syntax to *any* user-defined type

```
vector<int> v = { 5, 29, 37 };   // Fine in C++11...
vector<int> v2 { 5, 29, 37 };    // Don't even need the =

v2 = { 10, 20, 30, 40, 50 };     // not really just for
                                 // "initialization" !
template<typename T>
class vector {                // A peek inside a typical STL
public:                       // container's implementation…
    vector(std::initializer_list<T>);    // (simplified)
    vector &operator=(std::initializer_list<T>);
    …
```

63

More Initializer Lists

```
vector<int> foo()
{
        vector<int> v {10, 20, 30};
        v.insert(end(v), { 40, 50, 60 }); // use with algos,

        for (auto x : { 1, 2, 3, 4, 5 })   // with for loops,
                cout << x << " ";
        cout << endl;

        return { 100, 200, 300, 400, 500 }; // most anywhere!

}

int main()
{
        for (auto x : foo())               // note: foo()
                cout << x << " ";          // returns vector
        cout << endl;
}
```

64

Danger, Will Robinson!

- Constructors taking an `initializer_list` are preferred over ones that don't…

```
vector<int> v(10, 20);   // Unambiguous: v gets
                         // 10 elements each
                         // equal to 20

vector<int> v2{10, 20};  // Surprise! v2 gets
                         // two elements: the
                         // values 10 and 20
```

- Advice: going forward, avoid writing constructors that ambiguously overload with others taking an `initializer_list`.

65

Problem: Old Initialization Syntax Can Be Confusing/Ambiguous

```
int main()
{
    int *pi1 = new int(10); // OK, initialized int
    int *pi2 = new int;     // OK, uninitialized
    int *pi3 = new int();   // Now initialized to 0
    int v1(10);             // OK, initialized int
    int v2();               // Oops!

    int foo(bar);           // what IS that?

    int i(5.5);             // legal, unfortunately
    double x = 10e19;
    int j(x);               // even if impossible!
}
```

66

Fix: C++11 Uniform Initialization, Prevention of Narrowing

```
typedef int bar;

int main()
{
    int *pi1 = new int{10}; // initialized int
    int v1{10};             // same
    int *pi2 = new int;     // OK, uninitialized
    int v2{};               // Now it's an object!
    int foo(bar);           // func. declaration
    int foo{bar};           // ERROR with braces
                            //    (as it should be)
    double x = 10e19;
    int j{x};               // ERROR: No narrowing
                            //   when using {}s
    int i{5.5};             // ERROR, no truncation
}
```

67

Careful When Initializing Aggregates...

- Surprise! Narrowing/truncation of aggregates (arrays and structures) is *always* an error in C++11... even using *legal* Old C++ syntax !!
- This is a **breaking change** in C++11 (NP, IMO)

```
struct S
{
        int i, j;
};

int main()
{
        S s1 = {5,10};          // OK everywhere
        S s2{5, 10};            // OK in C++11
        S s3{5, 10.5};          // Error everywhere
        S s4 = {5, 10.5};       // ERROR in C++11...
                                //    but OK in Old C++!
        int a[] = {1, 2, 3.3};  // ERROR in C++11,
}                               //   OK in Old C++
```

68

More Danger, `auto` Style

- Another gotcha: when combining initializer lists with `auto`, the type deduced by `auto` is the type of the *intializer list itself*:

```
auto a(10);              // As expected…
cout << "a has type: " <<
      typeid(a).name() << endl;

auto b {10};             // Surprise!
cout << "b has type: " <<
      typeid(b).name() << endl;

// Output:
// a has type: i
// b has type: St16initializer_listIiE
```


Problem: Algorithms Not Efficient When Used with Function Pointers

- Inlining rarely applies to function pointers

```
inline bool isPos(int n) { return n > 0; }

int main()
{
    vector<int> v {-5, -19, 3, 10, 15, 20, 100};
                    // Calls to isPos probably NOT inlined:
    auto firstPos = find_if(begin(v), end(v), isPos);
    if (firstPos != end(v))
       cout << "First positive value in v is: "
              << *firstPos << endl;

                 // Old function object adaptors can eliminate
    firstPos = find_if(begin(v), end(v), // some functions,
       bind2nd(greater<int>(), 0) );    // but they're messy!
}
```

Function Objects Improve Performance, But Not Clarity

```
// Have to define a separate class to create function
// objects from:

struct IsPos
{
        bool operator()(int n) { return n > 0; }
};

int main()
{
        vector<int> v {-5, -19, 3, 10, 15, 20, 100};

        auto firstPos =
                     find_if(begin(v), end(v), IsPos());
        if (firstPos != end(v))
               cout << "First positive value in v is: "
                           << *firstPos << endl;
}
```

71

Lambda Expressions

- A *lambda expression* specifies an anonymous, on-demand function object
- Allows the logic to be truly localized
- Herb Sutter says: "Lambdas make the existing STL algorithms roughly 100x more usable."

```
int main()
{
    vector<int> v {-5, -19, 3, 10, 15, 20, 100};

    auto firstPos = find_if(begin(v), end(v),
              [](int n){return n > 0;} );

    if (firstPos != end(v))
       cout << "First positive value in v is: "
              << *firstPos << endl;
}
```

72

Lambdas and Local Variables

- Local variables in scope before the lambda may be *captured* in the lambda's `[]`s
  - The resulting (anon.) function object is called a *closure*

```
int main()
{
    vector<double> v { 1.2, 4.7, 5, 9, 9.4};
    double target = 4.9;
    double epsilon = .3;

    auto endMatches = partition(begin(v), end(v),
                [target,epsilon] (double val)
                { return fabs(target - val) < epsilon; });

    cout << "Values within epsilon: ";
    for_each(begin(v), endMatches,
                 [](double d) { cout << d << ' '; });
}                                  // output: 4.7 5
```

73

Different Capture Modes

- Lambdas may capture by reference:
  `[&variable1, &variable2]`
- Mix capturing by value and by ref:
  `[variable1, &variable2]`
- Specify a default capture mode:
  `[=]` (or) `[&]`
- Specify a default, plus special cases:
  `[=, &variable1]`

74

Only *Locals* Can Be Captured

- Capturing only applies to *non-static local variables* (including parameters)
- Within a member function, data members cannot be captured directly
  - They may be accessed (and modified) from within the lambda by capturing `this` (even if by value!)
  - Alternatively, they may be copied into a local variable
    - Then the lambda can capture that variable

75

"Avoid Default Capture Modes"

- A lambda may access any global/static data already in scope without any special fanfare
  - Thus, data outside the closure may be modified despite the fact it was not captured
- The potential confusion over what the lambda can modify, along with the potential for dangling references, has moved Scott Meyers to recommend (in his soon-to-be-published *Effective Modern C++*) that programmers "avoid default capture modes."

76

Lambdas as "Local Functions"

- Defining functions directly within a block is not supported in C++, *but…*

```
int main()
{
   double target = 4.9;
   double epsilon = .3;

   bool withinEpsilonBAD(double val)       // ERROR!
      { return fabs(target - val) < epsilon; };

   auto withinEpsilon = [=](double val)    // OK!
      { return fabs(target - val) < epsilon; };

   cout << (withinEpsilon(5.1) ? "Yes!" : "No!“);
}                                    // Output: Yes!
```

77

C++14 Generic Lambdas

```
vector<shared_ptr<string>> vps;

// Example #1:
sort(begin(vps), end(vps), []                        // C++11
(const shared_ptr<string> &p1, const shared_ptr<string> &p2)
   { return *p1 < *p2 } );

sort(begin(vps), end(vps), []                        // C++14
   (const auto &p1, const auto &p2) { return *p1 < *p2 });

// Example #2:
auto getsize = []                                    // C++11
   (const vector<shared_ptr<string>> &v) { return v.size();};

                                                     // C++14
auto getsize = []( auto const& c ) { return c.size(); };

// Note: Examples based on Herb's 4/20/13 Trip Report
```

78

Uses for Lambdas

- As we've seen, they're great for use with STL algorithms
  - Predicates for `*_if` algorithms
  - Algos using comparison functions (`sort`, etc.)
- Quick custom deleters for `unique_ptr` and `shared_ptr`
- Easy specification of predicates for condition variables in the threading API
- On-the-fly callback functions

79

Problem: Gratuitous Copying

- In Old C++, objects are (or might be) *copied* when replication is neither needed nor wanted
  - The "extra" copying can sometimes be optimized away (e.g., the RVO), but often is not or cannot

```
class Big { ... };           // expensive to copy

Big makeBig() { return Big(); } // return by value
Big operator+(const Big &, const Big&); // arith. op.

                             // This may cost up to 3
Big bt = makeBig();          // ctors and 2 dtors!

Big x(…), y(…);
Big sum = x + y;   // extra copy of ret val from op+ ?
```

80

Old C++ Solutions are Fragile

- The functions *could* be re-written to return:
  - References – but how is memory managed?
  - Raw pointers – prone to leaks, bugs
  - Smart pointers – more syntax and/or overhead
- But if we know the returned object is a *temporary*, we know its data will no longer be needed after "copying" from it
- The solution begins with a new type of reference. But first, some terminology…

81

Ancient Terms, Modern Meanings

- Lvalues
  - Things you can take the address of
  - They may or may not have a name
    - E.g., an expression `*ptr` has no name, but has an address, so it's an lvalue.
- Rvalues
  - Things you can't take the address of
  - Usually they have no name
    - E.g., literal constants, temporaries of various kinds

82

C++11 Rvalue References

- An *rvalue reference* is declared with &&
- Binds *only* to (unnamed) temporary objects

```
int fn();                    // Note: return val is rvalue
int main()
{
   int i = 10, &ri = i;      // ri is ordinary lvalue ref
   int &&rri = 10;           // OK, rvalue ref to temp
   int &&rri2 = i;           // ERROR, attempt to bind
                             //       lvalue to rvalue ref
   int &&rri3 = i + 10;      // Fine, i + 10 is a temp

   int &ri2 = fn();          // ERROR, attempt to bind
                             //       rvalue to lvalue ref
   const int &ri3 = fn(); // OK, lvalue ref-to-const

   int &&rri4 = fn();        // Fine, ret. val is a temp
}
```

83

Copy vs. Move Operations

- C++ has always had the "copy" operations--the *copy constructor* and *copy assignment operator*:

```
T::T(const T&);              // copy ctor
T &operator=(const T&);      // copy assign.
```

- C++11 adds "move" operations—the *move constructor* and *move assignment operator*:
  - These operations *steal* data from the argument, transfer it to the destination--leaving the argument an "empty husk"
    - This husk can be destroyed / assigned to, and not much else

```
T::T(T &&);              // move ctor
T &operator=(T &&);      // move assignment

// Note: Both would typically be noexcept
```

84

"Big" Class with Move Operations

- So there are now six canonical functions per class (used to be four) that class authors may define

```
class Big {
public:
   Big();                             // 1. default ctor
   ~Big();                            // 2. destructor
   Big(int x);                        // (non-canonical)

   Big(const Big &);                  // 3. copy ctor
   Big &operator=(const Big &);       // 4. copy assign.
   Big(Big &&);                       // 5. move ctor
   Big &operator=(Big &&);            // 6. move assign.
private:
   Blob b;          // e.g. some resource-managing type
   double x;        // other data…
};
```

85

Move Operations In Action

```
Big operator+(const Big &, const Big &);
Big munge(const Big &);
Big makeBig()  { return Big(); }

int main()
{
   Big x, y;             // Note: below, "created" really
   Big a;                //     means "not just moved"

   a = makeBig();        // 1 Big created *
   Big b(x + y);         // 1 Big created *
   a = x + y;            // 1 Big created *
   a = munge(x + y);     // 2 Bigs created *
   std::swap(x,y);       // 0 Bigs created!
}

// *: Return value's contents moved to destination obj
```

86

Move Operations: Not Always Automatic

- Consider the Old C++-style implementation of the `std::swap` function template:

```
template<typename T>
void swap(T &x, T &y)     // lvalue refs
{
        T tmp(x);         // copy ctor
        x = y;            // copy assignment
        y = tmp;          // copy assignment
}
```

- Even when applied to objects (e.g., `Big`) *with move support*, that support won't be used!

87

Forcing Move Operations

- Here's a C++11 version of `std::swap`:

```
template<typename T>
void swap(T &x, T &y)     // still lvalue refs
{
        T tmp(move(x));   // move ctor
        x = move(y);      // move assignment
        y = move(tmp);    // move assignment
}
```

- `move` is a zero-cost function meaning "cast to rvalue"
- Note: this `swap`'s signature is still the same as for old `swap`, but move operations are used *if available* (else falling back on copy operations)

88

Implementing Big's Move Operations

```
class Big {
public:
    …
    Big(Big &&rhs) :                          // move ctor
        b(move(rhs.b)), x(rhs.x) {}

    Big &operator=(Big &&rhs)         // move assignment op.
    {
        b = move(rhs.b);      // Note we NEED the moves, because
        x = rhs.x;            // rhs itself is an lvalue! (even
        return *this;         // though it has type rvalue ref)
    }

private:
    Blob b;
    double x;
};
```

- `Big`'s move operations simply delegate to `Blob`'s move ops, and assume they do the right thing…

89

Blob's Move Operations

- …so `Blob`'s move ops must do the "stealing":

```
class Blob {
public:
    …
    Blob(Blob &&rhs) {                    // move ctor
        raw_ptr = rhs.raw_ptr;            // "steal" pointer
        rhs.raw_ptr = nullptr;            // clear source
    }

    Blob &operator=(Blob &&rhs) {         // move assign. op
        if (this != &rhs) {
            delete [] raw_ptr;
            raw_ptr = rhs.raw_ptr;        // "steal" pointer
            rhs.raw_ptr = nullptr;        // clear source
        }
        return *this;
    }
private:
    char *raw_ptr;
};
```

90

When && "*Doesn't Mean Rvalue*"

- Scott Meyers coined the term *Universal References* for refs--declared using && in a *type deduction* context--that behave as <u>either</u> lvalue <u>or</u> rvalue references:

```
auto &&x = 3.1415;          // x is an rvalue

double pi = 3.14;
auto &&y = pi;              // y is an lvalue

template<typename T>        // Here, val can be
void f(T &&val);            // lvalue OR rvalue!

                            // functions instantiated:
f(3.14);                    // f(double &&);
f(x);                       // f(double &&);
f(pi);                      // f(double &);
```

91

Explanation: Reference "Collapsing"

- Refs-to-refs in a universal ref. (*deduction*) context:
  - `T & &   → T&`
  - `T && &  → T&`
  - `T & &&  → T&`
  - `T && && → T&&`

"Lvalue references are infectious"
-STL

```
template<typename t>        // Here, val can be
void f(T &&val);            // lvalue OR rvalue!
double pi = 3.14;

f(3.14);                    // f(double && &&); →
                            //    f(double &&);

f(pi);                      // f(double & &&); →
                            //    f(double &)
```

92

Efficient Sub-object Initialization?

- How many constructors would be required when there are many expensive-to-copy sub-objects?

```
class Big {
public:
    Big(const Blob &b2, const string &str) :   // copy both
        b(b2), s(str) {}

    Big(Blob &&b2, string &&str) :             // move both
        b(move(b2)), s(move(str)) {}

    Big(const Blob &b, string &&str) :         // copy 1st,
        b(b2), s(move(str)) {}                 // move 2nd

    Big(Blob &&b, const string &str) :         // move 1st
        b(move(b2)), s(str) {}                 // copy 2nd
private:
    Blob b; // what if we added other data members? Arghhh!
    string s;
};
```

93

Perfect Forwarding

- We'd prefer for each parameter to be copied or moved *as per its original lvalue-ness or rvalue-ness*

```
class Big {
public:
    template<typename T1, typename T2>
    Big(T1 &&b2, T2 &&str) :     // Universal refs
        b(std::forward<T1>(b2),  // std::forward preserves the
        s(std::forward<T2>(str)) // lvalue-ness or rvalues-ness
    {}                           // (and const-ness) of its arg

private:
    Blob b;
    string s;
};
```

94

When Move-Enable a Type?

- In the general case, move operations should be added only when *moving can be implemented faster than copying*
- Most C++11 library components are move-enabled
  - Some (e.g. `unique_ptr`, covered later) are *move-only*--they don't support conventional copy operations.
  - Internally, the implementations of many components, e.g. containers, employ moves whenever possible (rather than copying)

95

"The Rule of 5"

- The Old C++ "Rule of 3" now becomes the "Rule of 5":
- Good C++11 style dictates that if you declare any copy operation, move operation or destructor (even if only with `=default` or `=delete`), then you should declare *all 5*
- If you declare *any* of the 5, *no* move operations will be generated implicitly
  - The *copy* operations are still generated if needed
    - Note, however: this behavior is *deprecated in C++11!*

96

C++14 Generalized Lambda Capture

- C++11 lambdas can only capture variables by value or by reference
  - So those lambdas can't capture move-only types
  - C++14's can be initialized with *arbitrary expressions*
    - They can have their own names, but even if not, the captured values are distinct from the originals

```
// Example from C++14 Wikipedia page:
auto ptr = std::make_unique<int>(10);
auto lambda =
    [ptr = std::move(ptr)] {return *ptr;};
```

97

Epilogue: Is C++ Too Complicated?

- This is an oft-heard complaint
- But how does one measure "complexity"?
  - Pages in the *language* specification?
    - C++98 Language: 309 pp.
    - C++11 Language: 424 pp.
  - How does that compare to other languages?
    - Java SE 7 Edition: 606 pp.
    - C# ECMA Standard 4th Ed. June '06: 531 pp.

98

Complexity vs. "Complications"

- I'd like to suggest some different metrics for what makes a language "complicated to use":
  - To what extent does good code rely on (undocumented) idioms?
  - How many "gotchas" show up while debugging?
  - Can you "say it in code" or do you have to explain what you're doing in comments?
  - Does attaining high performance require jumping through hoops?
- By these criteria, I believe C++11/14 "measures up" pretty well!

99

C++ Resources

- For live links to resources listed here and more, please visit my "links" page at BD Software:
  `www.bdsoft.com/links.html`
- The C++ Standards Committee:
  `www.open-std.org/jtc1/sc22/wg21`
  (Draft C++ Standard available for free download)
- ISO C++ Site (spearheaded by Herb Sutter and the Standard C++ Foundation):
  `isocpp.org`

100

Overviews of C++11/14

- Bjarne Stroustrup's C++11 FAQ:
  `http://www.stroustrup.com/C++11FAQ.html`
- Wikipedia C++11 page:
  `en.wikipedia.org/wiki/C++11`
- Elements of Modern C++ Style (Herb Sutter):
  `herbsutter.com/elements-of-modern-c-style/`
- Scott Meyers' *Overview of the New C++ (C++11)*
  `http://www.artima.com/shop/overview_of_the_new_cpp`

C++14 Features

- *C++14 and early thoughts about C++17* (Bjarne Stroustrup):
  `https://parasol.tamu.edu/people/bs/622-GP/C++14TAMU.pdf`
- *A Look at C++14: Papers* (Meeting C++)
  `http://www.meetingcpp.com/index.php/br/items/a-look-at-cpp14-papers-part-1.html`
- C++14 Wiki
  `http://en.wikipedia.org/wiki/C++14`

On Specific New C++ Features

- *Rvalue References and Perfect Forwarding Explained* (Thomas Becker):
  `http://thbecker.net/articles/rvalue_references/section_01.html`
- *Universal References in C++* (Scott Meyers)
  - Article, with link to great video from C&B '12:
    `http://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers`
- *Lambdas, Lambdas Everywhere* (Herb Sutter)
  - These are the slides (there are videos out there too):
    `http://tinyurl.com/lambdas-lambdas`

103

On Specific New C++ Features: Using the Standard Smart Pointers

- *Guru of the Week #89: Smart Pointers* (Herb Sutter):
  `http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers`
- *Guru of the Week #91: Smart Pointer Parameters* (Herb Sutter):
  `http://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters`

104

Multimedia Presentations

- Herb Sutter
  - *Why C++?* (Herb's amazing keynote from *C++ and Beyond 2011*, a few days before C++11's ratification):
    `channel9.msdn.com/posts/`
    `C-and-Beyond-2011-Herb-Sutter-Why-C`
  - *Writing modern C++ code: how C++ has evolved over the years*:
    `channel9.msdn.com/Events/BUILD/`
    `BUILD2011/TOOL-835T`
- Going Native 201x (@ µSoft) Talks
  - Bjarne, Herb, Andre, "STL", many others:
    `http://channel9.msdn.com/Events/GoingNative`

105

Concurrency Resources

- Tutorials
  - Book: *C++ Concurrency in Action* (Williams)
  - Tutorial article series by Williams: *Multithreading in C++0x (parts 1-8)*
  - *C++11 Concurrency Series* (9 videos, Milewski)
- `just::thread` Library Reference Guide
  - `www.stdthread.co.uk/doc`

106

Where to Get Compilers / Libraries

- Twilight Dragon Media (TDM) gcc compiler for Windows
  `tdm-gcc.tdragon.net/start`
- Visual C++ Express compiler
  `http://www.microsoft.com/visualstudio/eng/downloads`
- Boost libraries
  `www.boost.org`
- Just Software Solutions (just::thread library)
  `www.stdthread.co.uk`
- If running under Cygwin, a Wiki on building the latest gcc distro under that environment:
  `http://cygwin.wikia.com/wiki/How_to_install_a_newer_version_of_GCC`

107

Testing Code on PCs

- Here are some good bets for trying out new C++ features on a Windows platform:
  - g++ 4.9.1 distro at *nuwen.net* (courtesy of Microsoft's own "STL"). Note: no threading support, but lots of C++14 language support:
    `http://nuwen.net/mingw.html`
  - clang 3.6 via the online compiler at *Wandbox* (full language/lib support):
    `http://melpon.org/wandbox`

108

"There are only two kinds of languages: the ones people complain about and the ones nobody uses."

-Bjarne Stroustrup

Thanks for attending!

Leor Zolman
`leor@bdsoft.com`

For many of the links cited here, and more, please visit: `www.bdsoft.com/links.html`

109