A Zephyr Overview of C++11

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

C++ now

May 13, 2013

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
- Part IV. Concurrency
- Part V. Standard library additions
- Most new features are at least mentioned

2

About the Code Examples

- As much as possible, I show specific deficiencies/issues in Old C++ and then introduce the C++11 solutions
- Most code has been tested using:
  - TDM gcc 4.6.1 w/`just::thread` 1.7.3 (Preview)
  - TDM gcc 4.5.2 w/`just::thread` 1.7.0 (released)
- Code excerpts shown on slides are not 100% self-contained programs
  - Read the code *as if* the requisite `#include`s, `using`s, `std::`s etc. were there
    - The slides are less cluttered without them….

3

A Brief History of C++

- 1979: Bjarne invents *C With Classes*
- 1998: First ISO C++ Standard (C++98)
- 2003: Bug Fix update (C++03)
  - In this presentation, I use the term *Old C++* to mean "C++98 and C++03"
- 2005: TR1 specifies new library components
- 2005-2011: "C++0x" evolves
- 2011: C++11 ratified (August)
- Next on the agenda
  - C++14 ("bug fix update" + a few new lang. features)
  - More TRs (Filesystem, Networking, Concepts Lite…)

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)
- Plus some others

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, nullptr };

    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

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;
}
```

11

`decltype` and Trailing Return Type

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

```
// 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;
}
```

12

`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

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…
}
```

15

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);            // calls f(long)
      f(nullptr);       // fine, calls f(char *)
      f(0);             // still ambiguous
}
```

16

`findNull` in C++11
(Final 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;
}
```

17

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;
              // (Note: Using C++11-only brace initialation)
    vector<int> vi { 10, 20, 100, 200, -500, 999, 333 };

    for (int i = 0; i < vi.size(); ++i)
       vi[i] += 100000;

    for (int i = 0; i < vi.size(); ++i)
       cout << vi[i] << " ";
}
```

18

Improvement: Range-Based `for` Loop

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

    for (auto i : ai)
        cout << i << " ";   // Don't need size
    cout << endl;
    vector<int> vi { 10, 20, 100, 200, -500, 999, 333 };
    for (auto &i : vi)
        i += 10000;         // Modify in place

    for (auto i : vi)
        cout << i << " ";
    cout << endl;

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

19

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*

20

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 compiler support for *compile-time* invariant validation and diagnosis:

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

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

21

`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!
}
```

22

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*

23

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)

24

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 vector<Widget>;
```

25

Problem: Exception Specifications

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

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

26

The C++11 Way: `noexcept`

- Exception specifications (even empty ones) can impact performance
- C++11 replaces exception specifications (now deprecated) with the `noexcept` keyword:

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

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

27

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

28

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);
};
```

29

Now For Average

- Another variadic function template can leverage the `sum()` template to give us average:

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

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

30

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";
```

31

Other Language Features

- `enum class`
  - Strongly scoped and typed `enum`s
  - Can specify underlying (integral) type
- `constexpr`
  - Enables compile-time evaluation of constant expressions *and functions* (including operators)
- `long long`
  - 64-bit (at least) ints

32

Other Language Features

- 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 };
```

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

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

- `alignas` / `alignof`
  - query/ force boundary alignment

33

Yet More Language Features

- Attributes
  - Replaces #pragmas, __attribute__, __declspec, etc.
  - E.g., `[[noreturn]]` to help compilers detect errors
- Inline Namespaces
  - Facilitates versioning; implicity "hoists" stuff from a sub-namespace into its enclosing namespace
- Generalized Unions
  - E.g., union members can now have constructors

34

Yet More Language Features

- Generalized PODs
  - E.g., "Standard Layout Types" (PODs) can now have ctors
- Garbage Collection ABI
  - Sets ground-rules for gc; specifes an ABI. [Note: No actual gc is required to exist.]
- 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;
```

35

Part II: Features Specific to Class Design

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

36

Problem: How to Disable Copying?

- There are two old C++ approaches to disallow the copying of objects
  - Either make the copy operations private:

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

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

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

  - Both are problematic.

37

C++11: =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!
}
```

38

Problems With Overriding

- When limited to old C++ syntax, the "overriding interface" is quite ambiguous

```
class Base {
public:
      virtual void f(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 h(int);       // overrides Base::h? Or… ?
};
```

39

override / final

- C++11 lets you say what you really mean:

```
class Base {
   public:
      virtual void f(int);    // Nothing more needed;
      virtual int g() const;  // Here, either
      void h(int) final;      // Invariant over special-
};                            //                 ization

class Derived : public Base {
   public:
      void f(int) override;   // Base::f MUST be virtual
      int g() override;       // Error!
      void h(int);            // Error! GOOD THING!!
};

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

40

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++) {}
//	...
private:
	complex<double> capacity;
	int id;
	static int nextId;
	void validate();
};
```


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(); }

private:
	complex<double> capacity;
	int id;
	static int nextId;
	void validate();
};
```

In-Class Initializers

- 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;
	static int nextId = 0;            // ERROR!
	complex<double> capacity = 100;   // ERROR!
	Cell FluxCells[num_cells];        // OK
};
```

43

C++11 In-Class Initializers

- Now, *any* data member can be (default) 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;
	static int nextId = 0;            // Now OK!
	complex<double> capacity = 100;   // Now OK!
	Cell FluxCells[num_cells];        // Still OK
};
```

44

Inheriting Constructors

- C++11 derived classes may "inherit" all ctors from their base class:
  - Simply extends the old `using Base::name` syntax to ctors (where they used to be arbitrarily excluded)
  - 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
};
```

45

Explicit Conversion Operators

- In Old C++, only constructors (a type of user-defined conversion) could be declared `explicit`
- User-defined conversion *operators* (e.g., `operator long()`) could not
- C++11 remedies that

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

46

Part III: Larger Language Features

- Initialization
  - Initializer lists
  - Uniform initialization
  - Prevention of narrowing
- Lambdas
- Rvalue references and "move" semantics

47

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);
}
```

48

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 need the =

v2 = { 10, 20, 30, 40, 50 };      // not 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>);
    …
```

49

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;
}
```

50

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!
}
```

51

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};           // ILLEGAL with braces
                            //    (as it should be)
    double x = 10e19;
    int j{x};               // ERROR: Narrowing when
                            //   using {}s is illegal
    int i{5.5};             // ERROR, fortunately!
}
```

52

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!
}
```

53

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;
}
```

54

Lambda Expressions

- A *lambda expression* creates 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;
}
```

55

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 sometimes called a *closure* (but I haven’t seen that term used consistently)

```
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
```

56

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]`

57

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!
```

58

Sneak Peek: 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
```

59

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+ ?
```

60

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…

61

But First…Some Terminology

- 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

62

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
}
```

63

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" still satisfying its invariants (sample implementations in a bit…)

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

// Note: Both really should be noexcept
```

64

"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();                             // default ctor
   ~Big();                            // dtor
   Big(int x);                        // (non-canonical)

   Big(const Big &);                  // copy ctor
   Big &operator=(const Big &);       // copy assignment
   Big(Big &&);                       // move ctor
   Big &operator=(Big &&);            // move assignment
private:
   Blob b;              // some resource-managing type
   double x;               // other data…
};
```

65

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
```

66

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!

67

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 we've forced move operations to be considered first, falling back on copy operations

68

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…

69

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;
};
```

70

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

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

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

auto &&x = 3.1415;        // x is an rvalue
auto &&y = pi;            // y is an lvalue

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

71

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 &)
```

72

Efficient Sub-object Initialiation?

- Consider constructors when there are several potentially 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;
    string s;
};
```

73

Perfect Forwarding

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

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

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

74

Move Operations and the Standard Library

- 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)

75

"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*
- The *copy* operations are still generated by default if needed--however, this behavior is *deprecated in C++11!*

76

Interlude: Some Omissions, Some Remedies

- The Old C++ Standard ignored several useful facilities of modern software design:
  - GUIs
  - Garbage Collection
  - `finally` blocks in exception handling
  - Concurrency
- There's *still* no GUI or `finally` support
- An ABI does exist in C++11 to support GC
- However, the most far-reaching *high-level* aspect of C++11 (IMO) is support for *concurrency*

77

Part IV: Essentials of Concurrency

- Multi-threading is *complicated*
- As with exception handling:
  - The language/lib support for concurrency is significant
  - Understanding best practices / idioms requires both study and experience
    - Reading at least one good book on the subject , such as *C++ Concurrency In Action* (by Anthony Williams, Manning Press) can help
    - Right now, that's the *only* book!
  - All we have time to do is scratch the surface

78

Concurrency Topics Covered:

- Threads
- Passing arguments to threads
- Synchronization with mutexes and locking
- Returning values from threads using futures and `async`
- Atomics
- (*en passant*: a peek at a few of the new time and random number library facilities)

79

Threads

- `main` runs in one single thread of execution
  - Pre-C++11, that single thread of execution was all the Standard recognized
    - One set of registers, one stack, one memory space, etc.
- In C++11, additional concurrent threads are launched by instantiating a `std::thread`
  - Each thread has its own stack for local data, but code and non-local data is shared
  - On multi-core / multi-processor systems, multiple threads can be truly concurrent
  - On single-core systems, they are time-sliced
  - Both scenarios are coded similarly

80

Starting a New Thread, 1st Attempt

```
void hello()
{
    cout << "Hello from new thread\n";
}

int main()
{
    thread t0(hello);
    cout << "Hello from main!\n";
}

// what happens if thread t0 is still
// running when main completes? (UB)
```

81

Starting a New Thread, 2nd Attempt

```
void hello()
{
    cout << "Hello from new thread\n";
}

int main()
{
    thread t0(hello);
    cout << "Hello from main!\n";
    t0.join();     // wait 'til t0 done
}
```

82

Functors, Lambdas as Threads

```
void hello();                     // function, as before

class Hello {                     // function object (functor)
public:
   void operator()() { cout << "Hello from functor\n"; }
};

int main() {
   thread t1(hello);              // function pointer

   Hello aHello;
   thread t2a(aHello);            // named function object
   thread t2b{Hello()};           // anonymous functor

   thread t3([]{ cout << "Hello from lambda!\n"; });

   t1.join(); t2a.join();
   t2b.join();
   t3.join();
}
```

83

Arguments and Threads: `bind`

```
void hello(const string &greeting, int n) {
      cout << greeting << "," << n << endl;
}

class Hello {
public:
      void operator()(const string &g)
      { cout << "Hello from " << g << endl; }
};

int main() {
      thread t1(bind(hello, "hello from function", 42));

      Hello aHello;
      thread t2a(bind(aHello, "named functor"));
      thread t2b(bind(Hello(), "anonymous functor"));

      t1.join(); t2a.join(); t2b.join();
}
```

84

Variadic `thread` Constructor

```
int main()
{
//    thread t1(bind(hello, "hello from function", 42));

                                    // Look Ma, no bind!
      thread t1(hello, "hello from function", 42);

      Hello aHello;

//    thread t2a(bind(aHello, "hello from named functor"));
      thread t2a(aHello, "hello from named functor");

//    thread t2b(bind(Hello(), "anonymous functor"));
      thread t2b(Hello(), "Hello from anon. functor");

      t1.join(); t2a.join(); t2b.join();
}
```

85

A Synchronization Issue

- Running either of the previous two examples reveals a problem
- Statements such as

  ```
  cout << greeting << "; n = " << n << endl;
  ```

  are composed of multiple interdependent expressions / function calls
- A thread context switch can occur anywhere within that statement, mixing output up between different lines in separate threads

86

Mutexes

```
mutex m;

void hello2(const string &greeting, int n)
{
   m.lock();              // "critical" section
   cout << greeting << "; n = " << n << endl;
   m.unlock();
}

class Hello {
   public:
      void operator()(const string &g)
      {     m.lock();   // critical section
            cout << g << endl;
            m.unlock();
      }
};

// BUT…what about exceptions in critical sections?
```

87

lock_guard

```
mutex m;

void hello2(const string &greeting, int n)
{
   lock_guard<mutex> lck(m);  // example of RAII
   cout << greeting << "; n = " << n << endl;
}                         // guaranteed unlocking

class Hello {
   public:
      void operator()(const string &g)
      {
            lock_guard<mutex> lck(m);
            cout << g << endl;
      }                   // guaranteed unlocking
};
```

88

Returning values from threads

- Consider a system for predicting the weather. We begin with a class to represent weather conditions:

```
class Condition {
   public:
      Condition (int n) : cond_(n) {}
      string describe() const { return conditions[cond_]; }
      static size_t last() { return conditions.size() - 1; }
   private:
      static vector<string> conditions;
      int cond_;
};

vector<string> Condition::conditions = {
   "hurricane", "nor'easter", "tropical_storm", "heavy_rain",
   "light_rain", "cloudy", "partly_cloudy", "sunny" };

ostream &operator<<(ostream &os, const Condition &c) {
   return os << c.describe(); }
```

89

Predicting the Weather
(Single-Threaded)

```
Condition predict_weather(system_clock::time_point t)
{   // using the C++ random number generator facilities...
    static uniform_int_distribution<int>
                    dist(0, Condition::last());
    static mt19937 engine;
    int n = dist(engine);

    return Condition(n);
}

int main()
{
       cout << "Forecast for 96 hours from now is: " <<
           predict_weather(system_clock::now() + hours(96))
              << endl;     // Above, C++11 time facilities
}

// But how do we launch predict_weather in a sub-thread
// and get the forecast result back into THIS thread?
```

90

Futures and async()

```
int main()
{
   future<Condition> theForecast =
      std::async(predict_weather,
             system_clock::now() + hours(96));

   cout << "Doing stuff while predicting" << endl;
   cout << "Doing more stuff while predicting" << endl;

   cout << "Weather prediction is for: "
             << theForecast.get() << endl;
}
```

91

Atomics

- We've seen how critical sections of code have to be synchronized
- The same principle applies to operations on primitives if they're shared among threads…

```
int global_int = 10;
atomic<int> ai(10);

int function()
{
    ++global_int;           // OK only if NOT shared

    ai.fetch_add(1);        // thread-safe (instead of ++ai)

    cout << ai << endl;     // prints 11
}
```

92

Part V: New Library Components

- New Function/Function Object Facilities
  - `std::function`
  - `std::bind`
- Smart Pointers
  - `std::unique_ptr`
  - `std::shared_ptr`
- Fixed-length Array
  - `std::array`
- Hash-based Containers
  - `std::unordered_*`
- Performance enhancements
- Note: Most new components originated in Boost!

93

Representing Function Objects

- We know templates can be written to support anything that "acts like a function":
  - ```
    template<typename In, typename Pred>
    In find_if(In begin, In end, Pred p);
    ```
    - `p` can be a function pointer
    - `p` can be a function object (including a lambda)
- But how do we extend this genericity to any object, not just to function template parameters?

94
    ```

std::function

```
size_t str_length(const string &s) { return s.length(); }

int main()
{
    string s("Hello, Dolly!");
    cout << s.length() << endl;

    function<int (const string &)> fn;

    fn = str_length;             // non-member function
    cout << fn(s) << endl;

    fn = &string::length;        // member function
    cout << fn(s) << endl;
                                 // lambda:
    fn = [](const string &s) { return s.length();};
    cout << fn(s) << endl;
}
```

95

Old C++ Binders

- Special-purpose 1-off functions are lame:

```
bool greaterThan5(int n) { return n > 5;}
… = find_if(v.begin(), v.end(), greaterThan5);
```

- Old C++ had `bind1st`, `bind2nd` to "fix" one argument of a binary function:

```
… = find_if(v.begin(), v.end(),
     bind2nd(std::greater_than<int>(), 5))
```

  - Some of the drawbacks to `bind1st` / `bind2nd`:
    - Limited to two arguments (one each)
    - Requires "adaptable" function object

96

`std::bind`

- C++11 provides the more flexible `std::bind`:

```
… = find_if(begin(v), end(v),
        bind(greater<int>(), _1, 5));
```

- However, lambdas are often preferable:

```
… = find_if(begin(v), end(v),
      [](int n) { return n > 5; });
```

97

Problem: Resource Leaks

- Memory and other resources managed by raw pointers are easily "leaked":

```
Widget *getWidget();
void crunch()
{
        int *ia = new int[1000];   // dyn. array of int
        Widget *wp = getWidget();  // Widget factory

        //  if code here throws, or otherwise
        //  returns from the function prematurely…

        delete wp;                 // Release the Widget
        delete[] ia;               // Release array of ints
}
```

98

Solution: Smart Pointers

- *Smart Pointers* are objects that
  - are initialized with a resource (the *RAII* idiom)
  - are used with the syntax of pointers
  - release that resource automatically upon destruction
- Typically, they are class templates specialized on the type of resource being managed
- Old C++ provided a single, zero-cost, smart pointer template, `auto_ptr`:

```
{
        auto_ptr<int> api(new int);
        *api = 10;
        // …
} // int pointer deleted automatically
```

99

Applying auto_ptr ?

```
Widget *getWidget();
void crunch()
{
        auto_ptr<Widget> wp(getWidget());      // Fine.
        auto_ptr<int> ia (new int[1000]);      // Mistake!

    //  Regardless of exceptions and/or returns out of
    //  this section of code, Widget automatically
    //  released…
    //  Unfortunately, undefined behavior for the array!
}
```

- `auto_ptr` also has strange semantics – *copying* an `auto_ptr` means *transferring* the resource!
  - Thus, `auto_ptr` has been deprecated in favor of…

100

The C++11 Solution: `unique_ptr`

```
Widget *getWidget();
unique_ptr<Widget> getWidget2();

void crunch()
{
   unique_ptr<Widget> wp(getWidget());   // init from ptr

   unique_ptr<Widget> wp2;    // copying from another
   wp2 = getWidget2();        // unique_ptr means “move”
   wp = wp2;                  // ERROR! (but rvalues only)

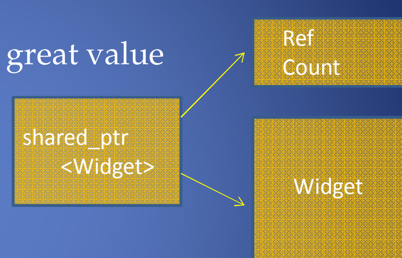
   unique_ptr<int[]> ia (new int[1000]); // arrays too!
   unique_ptr<int> ia (new int[1000]); // ERROR!

   unique_ptr<FILE, int (*)(FILE *)>    // custom deleter!
      fp(fopen("file.txt", "r"), fclose); // (not 0-cost)
}              // All resources released OK
```

101

Reference-Counted Smart Pointer: `shared_ptr`

- Introduced in TR1
  - Not “Zero Cost”, but still a great value




```
class Widget {
public:
      Widget(int, double);
};

void crunch()
{                              // initialize from ptr:
   shared_ptr<Widget> spw(new Widget(10, 2.23));

   vector<shared_ptr<Widget>> vw;
   list<shared_ptr<Widget>> lw;

   vw.push_back(spw);      // copy shared_ptr, NOT the Widget
   lw.push_back(spw);      // another copy of shared_ptr
}     // The ONE Widget is destroyed before return
```

102

An Optimization: make_shared

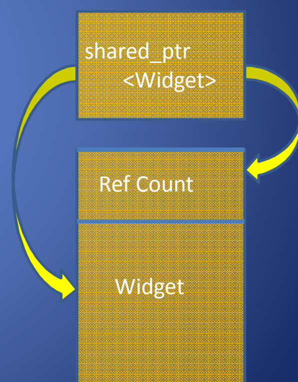
- A single memory allocation suffices for both the resource *and* the `shared_ptr`'s reference count:

```
class Widget {
public:
        Widget(int, double);
};

void crunch() // allocate Widget AND
{    // ref. count in one fell swoop:
    auto spw =
         make_shared<Widget>(10, 2.23);

    vector<shared_ptr<Widget>> vw;
    list<shared_ptr<Widget>> lw;

    vw.push_back(spw);
    lw.push_back(spw);
}
```



103

The array template: Arrays as First-Class Objects

- Another component introduced in TR1
  - The "nail in the coffin" of built-in arrays?

```
void f1(int a[]);
void f2(vector<int> v);
void f3(array<int, 5> a);

int main()
{
        int ai[] {5, -3, 25, 0, -2};
        vector<int> vi { 3, -19, 0, 6, 5};
        array<int, 5> ai2 {35, -5, 13, -20, 6};

        f1(ai);       // just passing pointer
        f2(vi);       // passing vi by value
        f3(ai2);      // passing ai2 by value
}
```

104

Templates, However, Can Still Be Quite Generalized!

```
template<class C>          // C is any container or array
auto min_elt(const C &cont) -> decltype(begin(cont))
{
       return min_element(begin(cont), end(cont));
}

int main() {
       vector<int> vi { 3, -19, 0, 6, 5};
       int ai[] {5, -3, 25, 0, -2};
       array<int, 5> ai2 {35, -5, 13, -20, 6};

       cout << "min val in vi = " <<
              *min_elt(vi) << endl;      // -19

       cout << *min_elt(ai) << endl;     // -3

       cout << "min val in ai2 = " <<
              *min_elt(ai2) << endl;     // -20
}
```

105

Hash-based Associative Containers

- Original associative containers
  - `set`, `multiset`, `map`, `multimap`
  - b-tree based, self-sorting
  - Insert/delete/lookup speed is $O(\log_2 N)$
- TR1 / C++11 hash-based associative containers
  - `unordered_set`, `unordered_map`, etc.
  - based on hash tables
  - *No* inherent sort/traversal order
  - Insert/delete/lookup speed *typically* faster…
    - …But not always. Issues can be complex. Rule of thumb: the larger the size of the container, the more likely a hash-based version will yield better overall performance.

106

Library Performance Improvements

- Containers' interfaces benefit from move operations and variadic templates:
  - `push_back` overloaded for rvalue refs
  - `emplace_back` accepts ctor argument list
- Internally, sequence containers employ move operations in lieu of copying
  - E.g., `vector` memory reallocation
- Algorithms, e.g. `sort` win by moving
- Initializer lists, lambdas streamline the use of algorithms

107

Some Library Components We Didn't Cover

- Larger Library Components
  - Regular expressions
  - Tuples
- Smaller Library Components
  - `std::weak_ptr`
  - `std::forward_list`
  - `std::result_of`
  - Wrapper references
  - Type traits (for TMP)
  - String conversion functions (`stof`, `stoi`, `stol`, etc.)
  - New algorithms
    - `copy_if, all_of`, `any_of`, `none_of`
    - `iota` (anyone remember APL?)
    - A bunch of others…

108

C++11 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`

109

Overviews of C++11

- Bjarne Stroustrup's C++11 FAQ:
  `www2.research.att.com/~bs/C++0xFAQ.html`
- Wikipedia C++11 page:
  `en.wikipedia.org/wiki/C++0x`
- 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`

110

On Specific C++11 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`

111

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 2012 (@ µSoft) Talks
  - Bjarne, Herb, Andre, "STL", many others:
    `http://channel9.msdn.com/Events/GoingNative/GoingNative-2012`

112

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`

113

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`

114

"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 all links cited, please visit:
`www.bdsoft.com/links.html`

115