The Last Thing D Needs

Scott Meyers, Ph.D.
Software Development Consultant
http://aristeia.com
smeyers@aristeia.com

Last Revised: 5/21/14

Missive from Walter

Having a relevant picture in your presentation can make it more interesting and visually appealing.

C++'s Success Speaks for Itself

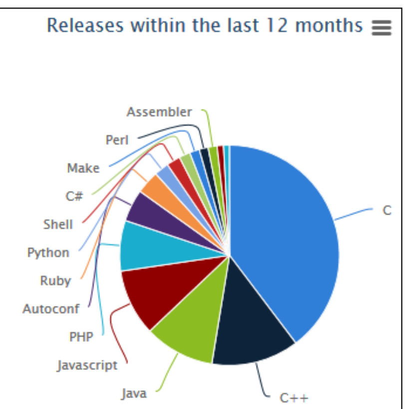
Tiobe Programming Community Index

Programming Language	2014	2009	2004	1999	1994	1989
C	1	2	2	1	1	1
Java	2	1	1	15	-	-
Objective-C	3	36	45	-	-	-
C++	4	3	3	2	2	2
C#	5	7	8	26	-	-
PHP	6	5	6	-	-	-
(Visual) Basic	7	4	5	3	3	7
Python	8	6	10	29	22	-
JavaScript	9	9	9	20	-	-
Transact-SQL	10	29	-	-	-	-
Lisp	14	20	15	12	6	3

D's position: 26

Black Duck Open Source Active-Project Language Use Data

Language	%
C	38.30
C++	12.46
Java	9.96
Javascript	9.43
PHP	7.18
Autoconf	4.57
Ruby	3.31
Python	2.07
Shell	1.97
C#	1.58
Make	1.39
Perl	1.27
Assembler	1.27
XML Schema	0.91
SQL	0.83



I'm not here to bash.

My Job

- Not D.
- Not designing languages.
- Not developing software.
- Occasionally consulting.
- Often training.
- **Always explaining.**
  - ➡ My perspective: easy to explain ≡ good.

All code in this presentation is in C++.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2014 Scott Meyers, all rights reserved.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2014 Scott Meyers, all rights reserved.

Fun with Initialization

`int x1;`	`// value? why?`	unknown
`int x2;`	`// (at global scope) value? why?`	0
`static int x3;`	`// value? why?`	0
`int *px = new int;`	`// value of *px? why?`	unknown
`{ int x4; ... }`	`// value? why?`	unknown
`int a1[100];`	`// values? why?`	unknown
`int a2[100];`	`// (at global scope) values? why?`	0
`static int a3[100];`	`// values? why?`	0
`std::vector<int> v(100);`	`// values? why?`	0

Fun with Type Deduction

```
const int cx = 0;
auto my_cx1 = cx;                     // type? why?          int

decltype(cx) my_cx2 = cx;             // type? why?          const int

template<typename T>
void f1(T param);
f1(cx);                               // T's type? why?      int

template<typename T>
void f2(T& param);
f2(cx);                               // T's type? why?      const int

template<typename T>
void f3(T&& param);
f3(cx);                               // T's type? why?      const int&
```

Fun with Type Deduction

```
const int cx = 0;
auto lam = [cx] { cx = 10; };           // error!

class UpToTheCompiler {
private:
  ??? cx;                               // type? why?     const int
  ...
};
```


Fun with Type Deduction

```
const int cx = 0;
auto lam = [cx = cx] { cx = 10; };      // error! why?

class UpToTheCompiler {
private:
  ??? cx;                               // type?   int (but acts like const int)
public:
  void operator()() const               // why const?
  { cx = 0; }
  ...
};
```

Fun with Type Deduction

```
const int cx = 0;
auto lam1 = [cx = cx] mutable { cx = 10; };     // error! why?
auto lam2 = [cx = cx]() mutable { cx = 10; };
class UpToTheCompiler {
private:
  ??? cx;                                        // type?   int (and acts like it)
public:
  void operator()()
  { cx = 0; }
  ...
};
```


Fun with Type Deduction

For

```
const int cx = 0;
```

type deduction for `cx` yields:

Context	Type
`auto` and `lambda init capture`	`int`
`decltype`	`const int`
`template(T parameter)`	`int`
`template(T& parameter)`	`const int`
`template(T&& parameter)`	`const int&`
`lambda (by-value capture)`	`const int`
~~`lambda (init capture)`~~	~~`int`~~

Fun with Type Deduction

```
int x1 = 0;
int x2(0);
int x3 = { 0 };
int x4 { 0 };

auto x1 = 0;              // type? why?     int
auto x2(0);               // type? why?     int
auto x3 = { 0 };          // type? why?     initializer_list<int>
auto x4 { 0 };            // type? why?     initializer_list<int>

template<typename T>
void f(T param);
f({0});                   // type? why?
```

error! “`{0}`” has no type

Fun with Inheritance

```
class Base {
public:
  void doBaseWork();
};
class Derived: public Base {
public:
  void doDerivedWork()
  {
    doBaseWork();                   // okay?
  }
};
```

Fun with Inheritance

```
template<typename T>
class Base {
public:
  void doBaseWork();
};

template<typename T>
class Derived: public Base<T> {
public:
  void doDerivedWork()
  {
    doBaseWork();                    // okay?
  }
};
```


Fun with Inheritance

```
template<typename T>
class Base {
public:
  void doBaseWork();
};

template<typename T>
class Derived: public Base<T> {
public:
  void doDerivedWork()
  {
    doBaseWork();                    // okay?
  }
};

template<>
class Base<int> {};                  // no doBaseWork


Derived<int> d;

d.doDerivedWork();                   // fail!
```

Fun with Inheritance

```
template<typename T>
class Base {
public:
  void doBaseWork();
};

template<typename T>
class Derived: public Base<T> {
public:
  void doDerivedWork()
  {
    this->doBaseWork();                          // okay!
  }
};
```


Fun with Computational Complexity

```
std::vector<int> v;
...
std::sort(v.begin(), v.end());                  // compiles? why? O(???)?
```

O(n lg n)
(Officially $O(n^2)$)

```
std::list<int> li;
...
std::sort(li.begin(), li.end());                // compiles? why? O(???)?
```

error!

```
auto it1 =
  std::binary_search(v.begin(), v.end(), 10);    // compiles? why? O(???)?
```

O(lg n)

```
auto it2 =
  std::binary_search(li.begin(), li.end(), 10);  // compiles? why? O(???)?
```

O(n)
(Officially O(lg n))

Image: WINNING INFORMATION @ Flickr ("stickynote")

Fun with APIs

Container member function to eliminate all copies of a value or map key?

```
std::set<int> si;
...
si.erase(14);                 // eliminate all 14s from si
```

- `set` `erase`
- `multiset` `erase`
- `map` `erase`
- `multimap` `erase`
- `unordered_set` `erase`
- `unordered_multiset` `erase`
- `unordered_map` `erase`
- `unordered_multimap` `erase`
- `list` `remove`
- `forward_list` `remove`

Fun with APIs

Sorts can be stable or unstable. Which are guaranteed to be stable?

- `sort` not guaranteed
- `stable_sort` guaranteed
- `list::sort` guaranteed

Fun with Specifications

"Table 100 – Sequence container requirements (in addition to container)"

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X a(n, t)`		*Requires:* T shall be `CopyInsertable` into X. post: `distance(begin(), end()) == n` Constructs a sequence container with n copies of t
`X(i, j)` `X a(i, j)`		*Requires:* T shall be `EmplaceConstructible` into X from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be `MoveInsertable` into X. Each iterator in the range `[i,j)` shall be dereferenced exactly once. post: `distance(begin(), end()) == distance(i, j)` Constructs a sequence container equal to the range `[i, j)`
`X(il);`		Equivalent to `X(il.begin(), il.end())`
`a = il;`	`X&`	*Requires:* T is `CopyInsertable` into X and `CopyAssignable`. Assigns the range `[il.begin(),il.end())` into a. All existing elements of a are either assigned to or destroyed. *Returns:* `*this`.
`a.emplace(p, args);`	`iterator`	*Requires:* T is `EmplaceConstructible` into X from `args`. For `vector` and `deque`, T is also `MoveInsertable` into X and `MoveAssignable`. *Effects:* Inserts an object of type T constructed with `std::forward<Args>(args)...` before p.
`a.insert(p,t)`	`iterator`	*Requires:* T shall be `CopyInsertable` into X. For `vector` and `deque`, T shall also be `CopyAssignable`. *Effects:* Inserts a copy of t before p.
`a.insert(p,rv)`	`iterator`	*Requires:* T shall be `MoveInsertable` into X. For `vector` and `deque`, T shall also be `MoveAssignable`. *Effects:* Inserts a copy of rv before p.
`a.insert(p,n,t)`	`iterator`	*Requires:* T shall be `CopyInsertable` into X and `CopyAssignable`. Inserts n copies of t before p.

Expression	Return type	Assertion/note pre-/post-condition
`a.insert(p,i,j)`	`iterator`	*Requires:* T shall be `EmplaceConstructible` into X from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be `MoveInsertable` into X and `MoveAssignable`. Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: i and j are not iterators into a. Inserts copies of elements in `[i, j)` before p
`a.insert(p, il);`	`iterator`	`a.insert(p, il.begin(), il.end())`.
`a.erase(q)`	`iterator`	*Requires:* For `vector` and `deque`, T shall be `MoveAssignable`. *Effects:* Erases the element pointed to by q
`a.erase(q1,q2)`	`iterator`	*Requires:* For `vector` and `deque`, T shall be `MoveAssignable`. *Effects:* Erases the elements in the range `[q1, q2)`.
`a.clear()`	`void`	Destroys all elements in a. Invalidates all references, pointers, and iterators referring to the elements of a and may invalidate the past-the-end iterator. post: `a.empty()` returns `true`. *Complexity:* Linear.
`a.assign(i,j)`	`void`	*Requires:* T shall be `EmplaceConstructible` into X from `*i` and assignable from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be `MoveInsertable` into X. Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: i, j are not iterators into a. Replaces elements in a with a copy of `[i, j)`.
`a.assign(il)`	`void`	`a.assign(il.begin(), il.end())`.
`a.assign(n,t)`	`void`	*Requires:* T shall be `CopyInsertable` into X and `CopyAssignable`. pre: t is not a reference into a. Replaces elements in a with n copies of t.

Fun with Specifications

Five sequence containers:

- `array` No
- `deque` Yes
- `forward_list` No (fulfills 1 of 16)
- `list` Yes
- `vector` Yes

Which ones fulfill the sequence container requirements?

Essential and Accidental Complexity


No Silver Bullet

Essence and Accidents of Software Engineering

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

Fashioning complex conceptual constructs is the essence; accidental tasks arise in representing the constructs in language. Past progress has so reduced the accidental tasks that future progress now depends upon addressing the essence.

Brooks, F.P., Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987.

Fred Brooks' terms. My view for languages:

- Essential complexity: due to inherent design tensions.
  - Simplicity and regularity vs. expressiveness.
  - Abstraction and portability vs. efficiency.
  - New approaches vs. compatibility with legacy systems.
  - Expressiveness vs. ability to issue good diagnostics.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Essential Complexity

```
struct Point {
  int x, y;
};
```

What is the type of `Point::x`?

```
Point p;
const Point& cp = p;
```

What is the type of `cp.x`?

C++ solution:

```
decltype(cp.x) ≡ int
decltype((cp.x)) ≡ const int&
```


Essential Complexity

```
template<typename T>
class Base {
public:
  void doBaseWork();
};

template<typename T>
class Derived: public Base<T> {
public:
  void doDerivedWork() { doBaseWrk(); }       // okay?
};
```

Assume typo and diagnose now?

- Wrong if later specialization offers `doBaseWrk`.

Assume later specialization and defer lookup until instantiation?

- If typo, imposes diagnostics for library errors on clients.

C++ solution:

- Template author has control.
  - `doBaseWrk()` ⇒ lookup name when parsing template.
  - `this->doBaseWrk()` ⇒ lookup name when instantiating template.

Accidental Complexity

Due to arbitrary design decisions.

- `int`s are *sometimes* initialized to 0.
- By-value lambda capture *sometimes* retains the `const`nesss of what's captured.
- `mutable` lambdas must declare a parameter list, but non-`mutable` lambdas don't.
- Braced initializers (e.g., "`{ 0 }`") *sometimes* have a type.
- Computation complexity guarantees *usually* meaningful.
- Eliminating all container elements with a given value *usually* means calling `erase`.
- `sort` is *sometimes* stable.
- Container "requirements" are *sometimes* required.
- Et freaking cetera (in C++).

C++ vs. D

C++:

- Too complicated to fix.
- Too constrained by legacy code compatibility requirements.
  - ➡ "Most C++ code is yet to be written" no longer heard.
- **No real interest by user community or standardization committee.**

D:

- Younger language and library.
- Smaller user community and legacy code base.
- Still time to embrace a *holistic* "easy to explain ≡ good" philosophy.

Piecemeal vs. Holistic Design Philosophies

Piecemeal:

- Each language rule easy to explain/justify in isolation.

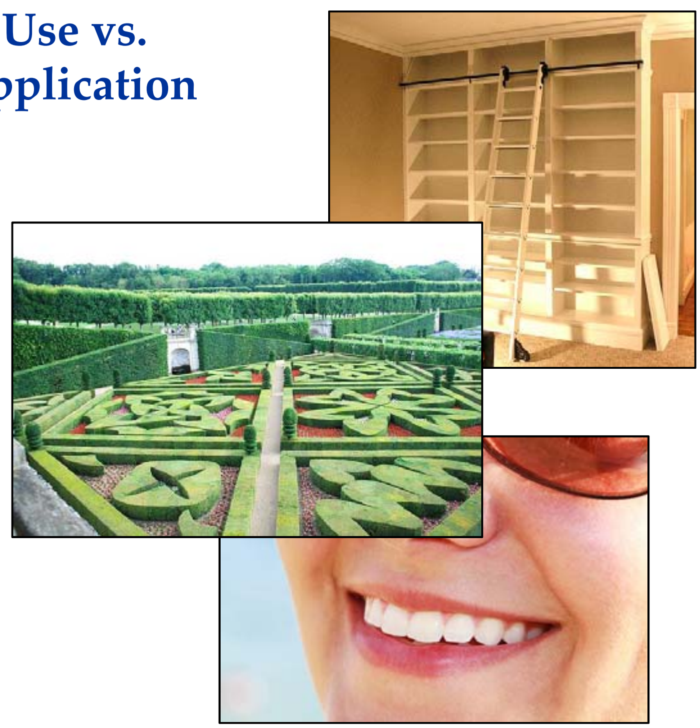
Holistic:

- Each language rule easy to explain/justify in isolation *and in context of other rules*.

C++ approach decidedly piecemeal. Popular entries in the justification-o-rama:

- Compatibility with C.
- Maximize efficiency.
- You don't pay for what you don't use.
- Trust the user.
- Prevent likely user errors.
- Make features general.
- Don't constrain compilers.
- Favor users over compiler writers.
- Retain backwards-compatibility.
- Be consistent with other features.

Tool Use vs. Tool Application



Images: Karin Dalziel @ Flickr ("Table Saw"), Martin Thomas @ Flickr ("Car boot sale tools"), D.C.Atty @ Flickr ("tools of torture"), Robin Zebrowski @ Flickr ("Cutom Bookcase Finished!"), Casper Moller @ Flickr ("Formal garden at Villandry"), Rupert Taylor-Price @ Flickr ("Smile").

Tool Use

Table saw:

- How to attach and remove the blade.
- How to raise/lower and angle the blade.
- How to avoid cutting off your fingers or poking an eye out with that thing.

C++:

- Rules governing compiler-generated “special” functions.
- How type deduction works (all six forms, sigh).
- How reference-collapsing works.
- Dependent vs. independent names in templates.

Tool Application

Table saw:

- How to keep wood from splintering as you cut it.
- How to make a tapered cut.
- How to mix sawdust with putty for color-matched hole filler.

C++:

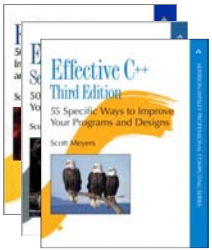
- How to replace virtual functions with templates in a physics simulator.
- How to use TMP to generate code for dispatch tables in an embedded automotive system.
- How to use lambdas to initialize `const` data structures.
- How to use `nth_element` and `sort` to outperform `partial_sort`.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

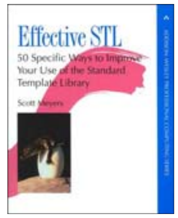
Scott Meyers, Software Development Consultant

http://www.aristeia.com/

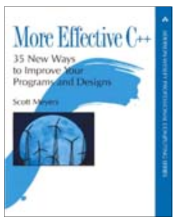
My Last Quarter Century



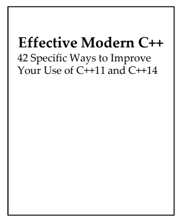
55 Guidelines



50 Guidelines



35 Guidelines



42 Guidelines

182 Guidelines!

Too much tool use, too little tool application.

The Last Thing D Needs...

...is someone like me.

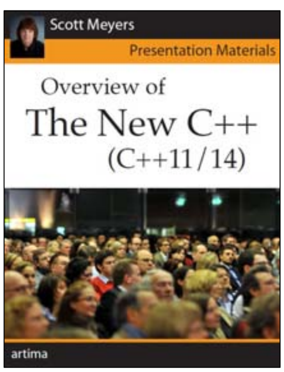
Image: Daniel Lobo @ Flickr ("No trabajas aquíí")

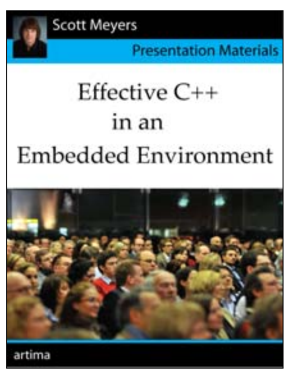
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:





Scott Meyers, Software Development Consultant
http://www.aristeia.com/

About Scott Meyers

Scott Meyers is one of the world's foremost authorities on C++. His web site,

http://aristeia.com/

provides information on:

- Technical training services
- Upcoming presentations
- Books, articles, online videos, etc.
- Professional activities blog

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2014 Scott Meyers, all rights reserved.