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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Java</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Objective-C</td>
<td>3</td>
<td>36</td>
<td>45</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C++</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>26</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PHP</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Python</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>29</td>
<td>22</td>
<td>-</td>
</tr>
<tr>
<td>JavaScript</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>20</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Transact-SQL</td>
<td>10</td>
<td>29</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lite</td>
<td>14</td>
<td>20</td>
<td>15</td>
<td>12</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Black Duck Open Source Active-Project Language Use Data

<table>
<thead>
<tr>
<th>Language</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>28.30</td>
</tr>
<tr>
<td>Java</td>
<td>9.96</td>
</tr>
<tr>
<td>Javascript</td>
<td>9.48</td>
</tr>
<tr>
<td>PHP</td>
<td>7.18</td>
</tr>
<tr>
<td>Autoconf</td>
<td>6.57</td>
</tr>
<tr>
<td>Ruby</td>
<td>3.31</td>
</tr>
<tr>
<td>Python</td>
<td>2.07</td>
</tr>
<tr>
<td>Shell</td>
<td>1.97</td>
</tr>
<tr>
<td>C#</td>
<td>1.58</td>
</tr>
<tr>
<td>Make</td>
<td>1.39</td>
</tr>
<tr>
<td>Perl</td>
<td>1.27</td>
</tr>
<tr>
<td>Assembler</td>
<td>1.27</td>
</tr>
<tr>
<td>XML Schema</td>
<td>0.91</td>
</tr>
<tr>
<td>SQL</td>
<td>0.83</td>
</tr>
</tbody>
</table>

D’s position: 26

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++. 
Fun with Initialization

```cpp
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

```cpp
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

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

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

```cpp
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

```cpp
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; }
  ...
};
```

For
```cpp
const int cx = 0;
```
type deduction for `cx` yields:

<table>
<thead>
<tr>
<th>Context</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto and lambda init capture</td>
<td>int</td>
</tr>
<tr>
<td>decltype</td>
<td>const int</td>
</tr>
<tr>
<td>template(T parameter)</td>
<td>int</td>
</tr>
<tr>
<td>template(T&amp; parameter)</td>
<td>const int</td>
</tr>
<tr>
<td>template(T&amp;&amp; parameter)</td>
<td>const int&amp;</td>
</tr>
<tr>
<td>lambda (by-value capture)</td>
<td>const int</td>
</tr>
<tr>
<td>lambda (init capture)</td>
<td>int</td>
</tr>
</tbody>
</table>
Fun with Type Deduction

```cpp
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

```cpp
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?
    }
};
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)
(Ofticially 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))
Fun with APIs

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

```cpp
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)”
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

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.
The Last Thing D Needs

Essential Complexity

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

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

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

What is the type of `cp.x`?

C++ solution:

```cpp
decltype(cp.x) == int
dcltype((cp.x)) == const int&
```

---

```cpp
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.

- *ints* are *sometimes* initialized to 0.
- By-value lambda capture *sometimes* retains the *const*ness 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`.
My Last Quarter Century

- Effective Modern C++
  - 42 Guidelines
  - 42 Guidelines

- Effective STL
  - 50 Guidelines

- More Effective C++
  - 55 Guidelines

- The Last Thing D Needs...

Too much tool use, too little tool application.

The Last Thing D Needs...

...is someone like me.

Image: Daniel Lobo @ Flickr ("No trabajas aquí!")
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](http://aristeia.com/Licensing/licensing.html)
- **Personal use**: [http://aristeia.com/Licensing/personalUse.html](http://aristeia.com/Licensing/personalUse.html)

Courses currently available for personal use include:
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