Why C++ Sails...

...When the Vasa Sank

Scott Meyers, Ph.D.

C++ in 1992

Standardization of a fledging language.

- Endless stream of extension proposals.
- Concern by standardization committee.
  - STL hadn’t yet been proposed.

Every extension proposal should be required to be accompanied by a kidney. People would submit only serious proposals, and nobody would submit more than two.

— Jim Waldo

C++ is already too large and complicated for our taste...

Remember the Vasa!

— Bjarne Stroustrup
The Vasa

- Commissioned 1625 by Swedish King Gustav.
- Much redesign during construction.
  - Ultimately designed to be flagship of royal navy.
- Heavily armed and decorated.

The Vasa

Top-heavy and unstable, it sank on its maiden voyage.

- Sailed less than a mile.
- Dozens died.
### C++: Heavily Armed

<table>
<thead>
<tr>
<th>Classes (concrete and abstract)</th>
<th>Namespaces</th>
<th>Virtual member functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructors and Destructors</td>
<td>RTTI</td>
<td>Nested classes</td>
</tr>
<tr>
<td>Default parameters</td>
<td>Locales</td>
<td>Nonvirtual functions</td>
</tr>
<tr>
<td>Operator overloading</td>
<td>constexpr</td>
<td>Static member functions</td>
</tr>
<tr>
<td>new and delete</td>
<td>Bitwise operations</td>
<td>Pure virtual functions</td>
</tr>
<tr>
<td>Lambda expressions</td>
<td>Iostreams</td>
<td>Function overloading</td>
</tr>
<tr>
<td>Inheritance (public, private, protected, virtual, multiple)</td>
<td>Static data (in classes, functions, namespaces, files)</td>
<td>Templates (including total and partial specialization)</td>
</tr>
<tr>
<td>Nonvirtual member functions</td>
<td>Inline functions</td>
<td>Atomic data types</td>
</tr>
<tr>
<td>Exceptions/Exception Specifications</td>
<td>References (Arguably 3 kinds)</td>
<td>New Handlers</td>
</tr>
<tr>
<td>Private and protected members</td>
<td>Friends</td>
<td>Memory Models (3)</td>
</tr>
<tr>
<td>User-defined literals</td>
<td>Raw and smart Pointers</td>
<td>Promises and Futures</td>
</tr>
<tr>
<td>Type deduction (3 sets of rules)</td>
<td>The STL</td>
<td>Enums (2 kinds)</td>
</tr>
</tbody>
</table>

### C++: Ornately Decorated

What is `f` in this code?

```cpp
f(x); // “invoke f on x”
```

- Maybe a function (or function pointer or function reference).
- Maybe an instance of a class overloading `operator()`.
- Maybe an object implicitly convertible to one of the above.
- Maybe an overloaded function name.
  - At any of multiple scopes.
- Maybe the name of one or more templates.
  - At any of multiple scopes.
- Maybe the name of one or more template specializations.
- **Maybe several of the above!**
  - E.g., overloaded function name + overloaded template name + name of template specialization(s).
Too Complicated?

If you think C++ is not overly complicated, just what is a "protected abstract virtual base pure virtual private destructor," and when was the last time you needed one?
— Tom Cargill (1990)

class Base {
    private:
        virtual ~Base() = 0;
    }

class Derived: protected virtual Base { ... };

C++ Must have Done Something Right

![Source: http://tinyurl.com/3xutoh](http://tinyurl.com/3xutoh)
C++ Must have Done Something Right

Language use in active open source projects:

<table>
<thead>
<tr>
<th>Language</th>
<th>Releases within the last 12 months</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>12.62%</td>
</tr>
<tr>
<td>C++</td>
<td>38.18%</td>
</tr>
<tr>
<td>Java</td>
<td>9.91%</td>
</tr>
<tr>
<td>Javascript</td>
<td>9.46%</td>
</tr>
<tr>
<td>PHP</td>
<td>7.19%</td>
</tr>
<tr>
<td>Autoconf</td>
<td>4.58%</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>1.57%</td>
</tr>
<tr>
<td>Make</td>
<td>1.39%</td>
</tr>
<tr>
<td>Assembler</td>
<td>1.27%</td>
</tr>
<tr>
<td>Perl</td>
<td>1.24%</td>
</tr>
<tr>
<td>XML Schema</td>
<td>0.91%</td>
</tr>
<tr>
<td>SQL</td>
<td>0.83%</td>
</tr>
</tbody>
</table>

Change in past year:

<table>
<thead>
<tr>
<th>Language</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>-14.7%</td>
</tr>
<tr>
<td>Java</td>
<td>2.19%</td>
</tr>
<tr>
<td>Javascript</td>
<td>3.03%</td>
</tr>
<tr>
<td>PHP</td>
<td>2.83%</td>
</tr>
<tr>
<td>Shell</td>
<td>-0.52%</td>
</tr>
<tr>
<td>Autoconf</td>
<td>2.4%</td>
</tr>
<tr>
<td>Python</td>
<td>0.29%</td>
</tr>
<tr>
<td>Assembler</td>
<td>-0.5%</td>
</tr>
<tr>
<td>Ruby</td>
<td>1.72%</td>
</tr>
<tr>
<td>C#</td>
<td>0.55%</td>
</tr>
<tr>
<td>SQL</td>
<td>-0.02%</td>
</tr>
<tr>
<td>XML Schema</td>
<td>0.07%</td>
</tr>
<tr>
<td>Make</td>
<td>0.65%</td>
</tr>
</tbody>
</table>

Compatibility with

- **Programmers:**
  - Important in 1982.
  - Important now.
    - C longevity astonishing.

- **Source code:**
  - “As close as possible to C, but no closer.”

- **Object code:**
  - Direct access to universal assembly language.

- **Build chains:**
  - Editors, compilers, linkers, debuggers, etc.

C & C++ still often lumped together: “C/C++”.
Very General Features

Lead to surprising uses.

- **Destructors.**
  - RAII ⇒ general-purpose Undo.
  - Basis of all resource management.
  - Exception safety.

- **Templates.**
  - Generic programming (e.g., STL).
  - TMP.
  - Compile-time dimensional units analysis.

- **Overloading.**
  - Generic programming.
  - Smart pointers.
  - Function objects.
  - Basis for lambdas in C++11.

A language for library writers.

![Image](tinyurl.com/awmleyr)

\[
\frac{1}{X_0} = 4 \alpha r_e^2 \frac{N_A}{A} \left\{ Z^2 \left[ L_{rad} - f(Z) \right] + Z L'_{rad} \right\}
\]

Energy:: constEnergy();
Element:: const & material, Density:: const dens,
Length:: const thick, Energy:: const initEnergy()
{
  AtomicWeight:: const A = material->atomicWeight;
  Atomic:: Number:: const Z = material->atomicNumber;
  Number:: const L_rad = log(184.15 / (1000.0 - Z)) * 1000.0;
  Number:: const Lp_rad = log(1194.0 / (1000.0 - Z.Floor)) * 1000.0;
  Length:: const X_0 = 4.0 * alpha * r_e * r_e * N_A / A * 
    (Z - 2.0 * L_rad + Z - Lp_rad);
  return initEnergy / exp(thick / X_0);
}

Paradigm Agnosticism

- **Procedural programming.**
  - Free functions:
    - Basis of STL.
    - Naturally models symmetric relationships.
    - Facilitates natural type extension.

- **OOP.**
  - Including MI.

- **Generic programming.**
  I tried to implement STL in other languages and failed.
  C++ was the only language in which I could do it.
  — Alexander Stepanov

- **Functional programming.**
  - Function objects (including closures).
  - TMP.
Paradigm Agnosticism

- **“Unsafe” programming.**
  - Pointers, casts, unchecked arrays, etc.
  - “Trust the programmer.”

- **“Safe” programming.**
  - Vectors with bounds checking, checked iterators, etc.

Commitment to Systems Programming

Suitable for demanding systems applications.

- **Program speed.**
  - Many systems are never fast enough.
  - Some systems must conserve CPU cycles.

- **Program size.**
  - Static (including RTL).
  - Dynamic (i.e., image and working set size).

- **Data layout.**
  - Drivers.
  - Communications protocols.
  - Use with legacy code/systems.

**Zero overhead principle:**

- **You don’t pay (at runtime) for what you don’t use.**
Commitment to Systems Programming

Designed for wide platform availability.
- **Microcontrollers to supercomputers.**
  - Implementation-defined sizes for e.g., `ints` and pointers.
  - No assumptions about I/O capabilities.
- **Hosted and unhosted.**
  - No OS required.

C++ compilers available almost everywhere.

Dedication to Backwards-Compatibility

Legacy code keeps working.
- **Circa-1989 C++ code almost always valid C++14 code, too.**

Standardization safeguards investments in C++.
C++ vs. The Vasa

So far, C++ has escaped the fate of the Vasa; it has not keeled over and disappeared – despite much wishful thinking and many dire predictions.

— Bjarne Stroustrup, 1996

Why?

- Compatibility with C.
- Very general features.
- Paradigm agnosticism.
- Commitment to systems programming.
- Dedication to backwards-compatibility.

Complexity Revisited

Consider again:

\[ f(x); \quad \text{// “invoke f on x”} \]

- \( f \) may be a function (or function pointer or function reference).
- \( f \) may be an instance of a class overloading `operator()`.
- \( f \) may be an object implicitly convertible to one of the above.
- \( f \) may be an overloaded function name.
  - At any of multiple scopes.
- \( f \) may be the name of one or more templates.
  - At any of multiple scopes.
- \( f \) may be the name of one or more template specializations.
- \( f \) may be several of the above!
  - E.g., overloaded function name + overloaded template name + name of template specialization(s).
**Complexity Revisited**

Is this really confusing?

\[
\text{std::cout} \ll x; \quad \text{\textit{// call operator<ll\textit{(std::cout, x)}}}
\]

\texttt{operator\ll} is overloaded, templatized, and specialized.
- In multiple scopes.

Most C++ complexity hidden from most users most of the time.

---

**Complex for Whom?**

\textbf{User view} often pretty simple:

\[
\text{auto y = std::move(x);} \quad \text{\textit{// request that x's value be moved to y}}
\]

\textbf{Implementer view} often more complex:

\[
\begin{align*}
\text{namespace std} \{ \\
\text{\quad template<typename T>} \\
\text{\quad typename remove_reference<T>::type&&} \\
\text{\quad move(T&& param) noexcept} \\
\text{\quad \{} \\
\text{\quad \quad typedef remove_reference<T>::type&& ReturnType;} \\
\text{\quad \quad return static_cast<ReturnType>(param);} \\
\text{\quad \} \\
\text{\}}
\end{align*}
\]
Destined for Complexity?

C++ most suited for **demanding systems applications**.

- Less demanding software ⇒ other languages suffice.
- You choose C++ ⇒ the situation is already complicated.

Or a Reflection of its Users?

[Diagram]

Sweet Spot
A Language on the Move

C++ Standard evolving increasingly rapidly:

- **ARM C++ (1990):** 453 pages.
  - Includes explanatory annotations.
- **C++98:** 776 pages (71% bigger). \( \Delta t = 8 \text{ years} \)
- **C++11:** 1353 pages (75% bigger). \( \Delta t = 13 \text{ years} \)
- **C++14:** ~1370 pages (~1% bigger). \( \Delta t = 3 \text{ years} \)
- **C++17:** Probably much bigger than C++14. \( \Delta t = 3 \text{ years} \)

Bigger = more complicated.

Simplification Through Added Complexity

New features often simplify common tasks:

```cpp
std::vector<Widget> v;
for (std::vector<Widget>::const_iterator ci = v.begin(); // C++98
     ci != v.end();
     ++ci) {
    use ci (often *ci)
}
for (auto ci = v.cbegin(); ci != v.end(); ++ci) { // C++11
    use ci (often *ci)
}
for (const auto cw& : v) { // C++11
    use cw
}
```
Simplification Through Added Complexity

E.g., evolution of function object creation:

```cpp
std::vector<int> v;

class InValueRange { // C++98
public:
    InValueRange(int bottom, int top): b(bottom), t(top) {}
    bool operator()(int val) const
    { return bottom <= val && val <= top; }
private:
    int b, t;
};

std::vector<int>::iterator i = std::find_if( v.begin(), v.end(),
    InValueRange(10, 20));
```

```cpp
auto i = std::find_if(v.begin(), v.end(), // C++11
    [](int val) { return 10 <= val && val <= 20; });
```

```cpp
struct ValidateAndFwd { // C++11
    template<typename... Ts>
    auto operator()(Ts&&... params) const
    -> decltype(process(std::forward<Ts>(params)...))
    { validateCredentials();
      return process(std::forward<Ts>(params)...);
    }
};

auto f11 = ValidateAndFwd();

auto f14 = [](auto&&... params) -> decltype(auto) { // C++14
    validateCredentials();
    return process(std::forward<decltype(params)>(params)...);
};
```
Status of The Vasa and C++

Museum piece: Preserved for Posterity

Living Language: Still Growing and Maturing

Further Information

- “Vasa (ship),” Wikipedia.
- Primarily an interview with Bjarne Stroustrup.
Further Information


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/](http://aristeia.com/) provides information on:

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