C++ History and Rationale

Shuo Chen

giantchen@gmail.com @bnu_chenshuo

blog.csdn.net/Solstice

Agenda

- Quiz
- Timeline
- Three Constraints
- Imperfections
- Why we use C++
- Improvements

Quiz

- Are virtual dtors necessary for base classes?
- Do you disable copy ctor and operator=() ?
- Ever worried about binary compatibility?
- Pimpl as common practices?
- Is auto_ptr recommended?
- Polymorphic array?
- Is std::copy as fast as memcpy for char[] ?

Quiz con't

- Is list::size() O(1) or O(*n*) ? list::empty() ?
- How about vector::push_back()? O(1) ?
- Break words in to vector<string>

```
string line;  // "cmd arg1 arg2"
vector<string> result;

istringstream iss(line);
istream_iterator<string> begin(iss);
istream_iterator<string> end;
copy(begin, end, back_inserter(result));
```

One system language every 10 years

- C 1970s stable since 1974
- C++ 1980s stable since 1996 (CD2)
- Java 1990s stable since 2004 (Java 5)
- ???? 2000s
- Go?? 2010s

- C#? well, it is a system-specific language

C++ in the 20th century

- C with class 1980 inherit w/o virtual
- First impl. 1983 virtual function
- CFront E/1.0 1985-02 overloading
- CFront 1.1 1985/86 reference
- CFront 1.2 1987-02 `protected`
- CFront 2.0 1989-06 Multiple-inheritance
- CFront 3.0 1991-10 Templates
- HP C++ 1992 Exceptions
- C++ 98/03 1996 namespaces/STL

Java in the 21th century

- Java 1.0 1996-01 Initial release
- Java 1.1 1997-02 JDBC
- Java 1.2 1998-12 Collections
- Java 1.3 2000-05 HotSpot JVM
- Java 1.4 2002-02 NIO
- Java 1.5 2004-09 Concurrent/generics/...
- Java 1.6 2006-12 Performance
- Java 1.7 2010-?? Closures

Major improvements every 2 years

C# in the 21th century

- C# 1.0 VS 2002/3 ≈ Java 5 - generics
- C# 2.0 VS 2005 Generics, partial class
- C# 3.0 VS 2008 LINQ, lambda
- C# 4.0 VS 2010 More dynamic

- 10 years ago *Java for C++ Developers*
- Now *C++ for Java Developers*

The second standard of a language is irrelevant

- C89 and C99
  - Does C99 change the way we write C programs?
- COBOL 60 and 2002
  - COBOL 2002 adds object-oriented, who cares?
- Fortran 77 and 90/95/03
  - 90/2003 adds object-oriented and generics
- IPv4 and IPv6, JPEG and JPEG2000
- C++98 and C++0x (Finalized on March 13, 2010)

What does C++ look like?

But, why?

480k EDG frontend vs. 250k HotSpot VM

Three constraints

- Compatible with C
- Zero-overhead principle
- Value semantics

Birthdays

- C was born on PDP-11 with 64k address space
  - The 1st C compiler involves two or three passes
  - Cc1 parse C source, generates intermediate code
  - Cc2 read in, generate machine code
  - Then link object files to executable
  - Why C has header files, compiler needs it
- C++ was born on 32-bit VAX with 1M memory
  - But it still uses header filer and expose to ODR violation, and to be binary compatible with C
  - Java compiler is smarter to find class definitions

C++ was born in Bell Labs

- Where Unix and C were born
- Same speed as C, why CFront compiles to C?
  - Otherwise no one would use it in the first place
- Same footprint as C, easily verifiable if C is the target
  - Class is almost same as struct, same size/layout
  - No virtual dtor by default, struct in sockets, mktime
- Compatible with C, a political pressure
  - Legal C code must be legal C++ code
  - Preprocessor/macros to compile Unix headers

Why class Foo{};

- Can be difficult to diagnose for novice
  - #include "foo.h" // missing ';' at the end
  - #include "bar.h" // strange errors in first lines
- C# and Java doesn't need the ';' for classes
  - C++ namespace {} doesn't neither, so why
- C allows defining unnamed struct
  - struct {int ask; int bid;} bidask; bidask.ask = 0;
  - struct {int x; int y;} getPoint(); // new type in return
- C++ doesn't allow unnamed class*, but has to follow the same syntax, ';' as delimiter.

Politics ? Yes, we're human

- ABI – compiler neutral inter-operations
  - It is said that to allow completive impls.
  - Every architecture except x86-32 has one ABI
    AMD64, ARM, MIPS, Itanium, PowerPC, SPARC
  - Truth: vendors didn't want to change their code
- abstract vs. =0
  - Adding new keyword would break existing code?
  - After CFront 2.0, we added `template`, `namespace`, `throw`, `catch`, …, in C++0x, will add `nullptr`
  - Truth: Too close to release 2.0, no time to add kw

Origination of features

- Why do we need object-oriented?
  - Because OO is the killer feature at 1980s
  - C++ made OO affordable for PCs, a major success factor
- Why do we need template?
  - Because Ada supports generic programming
- Why do we need exception?
  - Because Ada supports exception handling
- Why does Ada matter, anyway?
  - It's the chosen language of DoD,
    a big buyer of Bell Labs

Zero-overhead principle

- As close to machine as possible, same as C
- A minimal C++ runtime only needs stack to be setup, same as C.
- The sequence of evaluation is unspecified
- Variables on stack are not initialized
  - But there is a rule says you'd always init as define
- The default ctor doesn't bzero() the POD
- Virtual is not by default (which is good)

Value semantics

- User-defined types (string) vs. built-ins (int)
  - Pass by value for class types
  - Allocate class object on stack
  - Return class object by value
  - Composite

```
struct TcpHeader { IpHeader ip; ports }; // C
struct TcpHeader : IpHeader { ports }; // C++
```

- It violates reference semantics in OO
  - Making a copy of Printer != having two printers
- Auto generated copy constructors/operator=
- User-defined bi-direction implicit conversion

Reference types

- Introduced for operator overloading
  - Matrix operator+(Matrix a, Matrix b) vs.
  - Matrix operator+(Matrix& a, Matrix& b)
  - BigInteger& BigInteger::operator++()
- A hole in the value semantic framework
  - When you hold a reference or a pointer as member, you worry about its life time.
  - Dynamic binding only works on ptr/reference
  - Not object-oriented, but pointer-oriented or reference-oriented. (kidding)

There are only two kinds of languages:

- The ones people complain about and
- The ones nobody uses

Imperfections

WTFs

C++ syntax is not context-free

- `Foo<T> a;`
- A few possibilities
  - Foo is a class template, T is a type
  - Foo is a class template, T is a const int
  - Foo is an int, T is an int
- Don't forget
  - operator< is overloadable, Foo and T can be objects
- To understand one line of C++ code
  - One must read through all header files

Template syntax

- To initial an integer to zero
  - int x(0); int x = 0; int x = int();
- To convert integer x to double
  - (double) x, double(x), static_cast<double>(x)
- Why C++ supports all of them? Template!

```
template <typename T>
class Sync {
 public:
  explicit Sync(const T& v) : value_(v)  {}
  Sync() : value_()  {}
 pricate:
  Mutex mutex_;
  T value_;
};
```

```
template<typename To, typename From>
inline To explicit_cast(const From &f) {
  return To(f);
}
```

Inconsistency

- Class vs. struct, class is private by default, but
  - The operator=() and copy-ctor are public
  - Makes C++ an unsafe language by default, for any non-trivial class, unless you explicitly disable them (Item 6, EC)
  - It's a bad decision, class and struct shouldn't be so close
- Class uses private inheritance by default
  - Contrast to common OO practices, (is-a, LSP)
- More? Yes!
  - ~100 rules to remember while coding
  - Most of them say of "thou shalt not"

C++ is a mess

- Four small languages meshed together
  - immanent contradictions
- Unnecessarily flexibility
  - Why would the language allow returning a reference or pointer to stack local variable?
- Complex scoping rule and overload resolution
  - Free functions defined at global or namespace level
  - Plus implicit type conversion provided by constructors and conversion operators.
  - Look up and look around (for function definitions)

Prefer library solution over language solution

- Not always cleaner, too many “idioms”
- To force checking return value
  - Loki::CheckReturn<T, Action>
  - int foo() __attribute__ ((warn_unused_result));
- To make a class not derivable
  - extra base class hiding ctor + `friend`
  - keyword `final`

Not a good OO language

- Much weaker than modern languages: Java/C#
  - no reflection, no dynamic creation or class loading
  - Object slicing, object life time management
- Essentially, OO programming is try and error
  - Complex syntax stops us building refactoring tools
  - No reflection stops easy mocking
  - Compiles slowly, who else do distributed compilation?
- Ellipsis (...) parameter works fine in C/Java/C#, but is discouraged in C++
  - auto boxing, single root hierarchy, toString()

Exceptions are bad in C++

- Not because exception handling is bad, it is essential in Java and many other languages
- Exception is not designed with the language
  - It is added 10+ years later after the language shaped, ie. a late patch
  - Inconsistence with value semantics
- “throw 1” or “throw 1.0” make no sense at all
- No established good practices
  - Many guys/teams fall back to C’s return value / error code approach

Can we catch in ctor?

- Yes, of course
- How about initialization list?
- Do you still want to parse C++?

```
class Person
{
 public:
  Foo() : name_("Shuo")
  {
    try {
    // ...
    } catch (...) {
    }
  }

 private:
  std::string name_;
};
```

```
class Person
{
 public:
  Foo()
  try
    : name_("Shuo")
  { // ...
  }
  catch (...) { // read name_ ?
  }

 private:
  std::string name_;
};
```

Exception vs. destructing

- Throw an exception in function will
  - Destructs all previous constructed objects
- Throw an exception in constructor will
  - Destructs all member objects and base object(s)
- Throw an exception in initialization list will
  - Destructs member objects constructed so far
- Throw an exception in array constructing
  - Destructs objects constructed before this one
- How about multiple and virtual inheritances

Even worse, threading

- Multithreading appeared early 1990s
  - Solaris 2.2/Windows NT 3.1 both in 1993
  - Breaks lots of C function
    strtok, itoa, errno, singals, fork
  - Java was born in 1996, designed with threads
- No practical memory order for years
- No reference impl. -> slow evolution
  - Tons of meeting, arguing, debate, paper work
  - Java took 3 years to fix "double checked locking"

Not cooperative

- Perl/Python/Ruby/Lua/Erlang are all in C
  - Ironically, object-oriented scripting languages expose API interfaces in C
- Libraries from two vendors can't be mixed
  - Not true for C/Java/Python, why?
  - Varied style/taste/resource management
- Binaries from two compilers can't be mixed
  - Combination explosion, if you supply .so

yourlibrary_gcc32_boost_1.33 yourlibrary_gcc32_boost_1.36
yourlibrary_gcc41_boost_1.33 yourlibrary_gcc41_boost_1.36
yourlibrary_gcc41-64_boost_1.33 yourlibrary_gcc41-64_boost_1.36

Not affordable for small companies

- Boost, QT, Poco, ACE, apr
  - all provide all-in-one solution, but incomplete
  - Difficult to mix any two of them
    - Due to diff philosophies behinds libraries
  - Every big company re-invents wheels (as I know)
- Java (not to mention Spring/Hibernate/Tomcat)
  - logback, xerces, xalan, joda-time, mina, guice, trove4j
- Dependency management
  - Java – Ant+Ivy, I can setup my repository in 1 hr
  - C++ - GNU Make? CMake? SCons? Autotools?

Why we use C++

C++ is deterministic

- Destructing is determinate
  - Arguably *the* most important feature of C++
- The performance is predictable
  - C++ is fast, only when the coders are experts
  - Java is almost as fast, much higher productivity
- Although less deterministic than C
  - When stepping through code with debugger, a function call jumps to constructors
  - The optimized machine code is unreadable due to inlining

Determinism?

- Is it O(N) or O(1) ?
  - std::string a("a string");
  - std::string b = a;
- Is it O(N) or O(N*M) ?
  - vector<string> vs; // N-length
  - vector<string> newvs = vs;
- Allocate memory from heap? (No COW)
  - std::string c = a;
  - short string optimization

A better C + data abstraction

- Use C++ as of in 1985: C + concrete class + STL
- Only use templates for saving typing
- Only use OO for replacing switch-case
  - Eg. IO-Multiplex, select/poll/epoll/kqueue
  - Never design a base case to be derived by others, it's hard to do it correctly in C++
  - Use boost::bind/boost::function for dynamic bindings
- Frameworks are bad, libraries are good
  - One size never fits all, we're using C++ for purposes (latency, throughput, footprint, etc.)
  - Flexibility? Specificity!

Misuses

- Impersonate other languages
  - Boost.Proto vs. embedded interpreter (eg. Lua)
  - Boost.Sprit vs. ANTLR
  - Boost.Preprocessor vs. code generator
  - SystemC vs. Verilog
- overemphasize reusability or flexibility
  - too many customizable possibilities (how to test?)
- overemphasize portability
  - Good example: Lua, bad example: ACE

Improvements?

With out compromise performance

Many thanks to RoachCock@smth

Core language

- Module system, No headers, No ODR
  - compiler looks up modules by namespaces
  - Modern package dependency management
- Enum introduces scope
  - Color::RED vs. Color_RED
- Fixed integer sizes and (un)sign of char
- Disallow hiding variable of outer scope
- No diff to new/new[] -- or disallow new[]
- Not convert bool to int; nullptr for NULL ptr

Core language

- Allow ctor calling ctor
- override – @Override
- final – not designed as a base class
- finally – try {} catch () {} finally {}
- abstract – must override/inherit
- Copy-ctor/operator= make private for class
- Default values for data members
- Stack trace on error
- variadic template/macros

Library

- Unnecessary flexibility
  - Remove allocator template parameter
    - A vector<int, MyAlloc> is not a vector<int>
  - Remove locales and facets
    - iostream should be faster than scanf and printf
  - Remove or deprecate error-prone/bad-designed
    - auto_ptr, valarray, vector<bool>
- Add more
  - Networking, threading, XML, date time, logging
  - You name it!

Conclusion

- C++ was designed a person who works with people who invented Unix and C
- C++ is not owned by a person or company
  - pros and cons
- C++ is a success
- C++ doesn't fit all
  - Know when and when *not* to use C++
  - Know how and how *not* to use C++

Thanks for your time