Elegant and efficient code

Bjarne Stroustrup
Texas A&M University
http://www.research.att.com/~bs
Overview

• I'll present our general approach to getting code that is more elegant and more efficient than current code. Our approach have three prongs:
  – Improvements to ISO Standard C++
  – Improvements to library design and implementation techniques
  – A new framework for the analysis and transformation of C++ source code
Traditional hard choices

• Pick any two
  – Quality
  – Speed
  – Features

• In computer science you can base your argument of one of two
  – Elegance
  – Speed

• I hate to choose between elegance and speed
  – Programs should be elegant and fast

• How do we dodge the horns of this dilemma?
What do we mean?

• Elegance
  – Maintainable
  – Simple to read and write
  – Simple to reason about
  – Simple to analyze
    • Humans and programs

• Performance (efficiency)
  – Equal or better than the current gold standard in any area
    • Fortran for scientific programming
    • C (and sometimes even assembler) for systems programming

• For industrial programs
  – Supported by industry standard tools
    • Tool chains are critical
  – Not just academic exercises
The dilemma is real

• The most elegant languages
  – (e.g. Smalltalk, ML, Haskell, Common Lisp)
  – Are not applicable for many important application areas
  – Are not efficient for many important problems
  – Are apparently not manageable for most programmers

• The most efficient languages
  – (e.g. C, C++, Fortran)
  – Encourage low-level messing/hacking
  – Have problems expressing some important problems elegantly
  – Are hard to “restrain” for reasoning/analysis/translation

• Many language are neither fish nor fowl
  – (e.g. Java and C#)
  – Neither elegant nor efficient
  – Not applicable for many important application areas
Our approach – three prongs

• Improve ISO Standard C++
  – Performance
  – Ability to express high level concepts
  – Applicable for the widest range of applications

• Add libraries
  – Based on a combination of generic and object-oriented techniques

• Add analysis and transformation support tools
  – Optimization
  – Dialect definition through selective restriction
  – More (we don’t yet know how far this may go)

• Why C++?
  – It is closest to our ideals in the largest range of application areas
Consider an example

- World’s most powerful marine diesel engine
  - fuel injector and engine control
  - 132,000Hp
  - Service time 20-30 years
  - Delay or injection error is not an option
  - No single point of failure
  - Heat and vibration hardened
  - Hot standby replicated systems (incl. networks)

This we can do today (MAN B&W using modern C++)

how can we make it easier and cheaper to build such systems?
C++

• We support/lead the ISO C++ standards effort
  – Add features for direct expression of guarantees
    • Concepts: a type system for template parameters
      – Separate compilation
      – No hierarchical constraints
      – No performance penalty
    • Notational support for more general expression
      – Generalized constructors
      – Generalized initializers
      – More effective use of type deduction: Decltype/auto
    • Better support for concurrency
      – Memory model
      – Threads
      – Lot’s of “usual library stuff”
        • E.g. regular expressions
Dialect support

• Express a domain specific language as a dialect of ISO standard C++
  – Enhanced with (typically high-level) libraries supporting the application domain
    • Note: Libraries usually written in standard C++
    • E.g. STAPL for scientific numeric parallel computation
  – Restricted from (low-level) features that could violate guarantees for the domain
    • Note: the restricted features can be used in the library implementations
    • E.g. arrays, unsafe use of pointers, unsafe use of unions
Advantages of this approach

• The full C++ tool chains are available
  – Compilers, linkers, debuggers, system simulators, libraries, etc.
• The full C++ language is available where needed
• Unless there is a domain specific reason we can link to any C++, C, Fortran, etc. library

• Private, specialized, and unpopular languages die
  – Poor tool chains (e.g. Ada)
  – Excessive education/training costs
  – Too narrow user community
  – …
The problem

How do we write programs that really understand C++ source code?

• The original problem (inspiration)
  – Poor support for CORBA and for high-level parallel and distributed programming techniques

• There are no widely-available and general static analysis and transformation support for C++

• There are competing many incomplete tools

• The community is fractured
  – Few dare to rely on other groups tools

• None of the existing tools deal with the higher levels of C++ (templates, specialization, concepts)
  – Those are the aspects of C++ that are crucial for advanced optimization, validation of safety, enforcement of dialects, support of advanced libraries
The Pivot
A framework for static analysis and transformation of C++
The Pivot parts

- IPR (Internal Program Representation)
  - And **fully general** typed abstract syntax tree representation of **all of** C++ (with the exception of macros)
  - Unified type system
  - Prepared for C++0x facilities; notably concepts
  - Potentially standard

- XPR (eXternal Program Representation)
  - Compact, persistent, user-readable, portable representation of IPR

- Compiler to IPR generators
- IPR $\leftrightarrow$ XPR parsers
- Traversal and transformation tools
- Specific tools
  - E.g. IPR $\leftrightarrow$ XPR, IPR $\leftrightarrow$ IDL, style checker
The Pivot Project

• Phase 1: Infrastructure
  – Build internal representation from compiler info
  – Support persistence of internal representation

• Phase 2: acceptance and low-hanging fruit
  – Present the Pivot to Compiler providers, the C++ standards committee, and static analysis users
    • Build alliances
  – Style checkers
    • E.g. for safety-critical embedded systems work
  – Simple optimizations
  – …

• Phase 3: Research
  – Exception safety validation
  – Support for domain specific libraries
    • Validation and optimization
  – Generate flow-control-oriented representation and tools
  – Tools for integrated symbolic and numeric calculation