Concepts Requirements

Bjarne Stroustrup

Texas A&M University

http://www.research.att.com/~bs

2009 Reflections

- Concepts for C++0x: We were so close
  - We had/have
    - Design
    - Experimental (only) implementation (ConceptGCC)
    - Text voted into C++0x WP
    - Standard library “conceptualized”
- What went wrong?
- What must we consider to avoid a 2nd failure

2009 Reflections

- Concepts for C++0x were defeated by
  - Fear of the new, fear of the unknown and unknowable
  - Fear of compilation overheads
  - Inflexibility relative to “status quo” definition/implementation of concepts
  - Fear that the specification was unstable
  - Fear of complexity (specifications and use)
  - Fear that the “status quo” design would not scale (program size and usability)
- “Schedule” was IMO only a small part of the problem
- We have to reevaluate and redesign from scratch
  - Of course we now have much more experience
  - We must work harder on simplicity (teachability and scalability)
  - We must work harder on validation
  - Implementation is no substitute for design (and vise versa)

Template argument checking

- Problem recognized early
  - Three pages of D&E
- Lobbying by Alex Stepanov
- Experiments with and use of constraints checking in C++98
  - E.g., Stroustrup, Siek, Boost
- Languages analysis: Garcia, et.al (several papers over the years)
- Bjarne Stroustrup and Gabriel Dos Reis: *Concepts – Design choices for template argument checking*
- Bjarne Stroustrup: *Concept checking – A more abstract complement to type checking*
  - October 22, 2003
  - Not a design – instead: a set of requirements

2003 Design aims

1. A system as flexible as current templates
2. Enable better checking of template definitions
3. Enable better checking of template uses
4. Better error messages
5. Selection of template specialization based on attributes of template arguments
6. Typical code performs equivalent to existing template code
7. Simple to implement in current compilers
8. Compatibility with current syntax and semantics
9. Separate compilation of template and template use
10. Simple/terse expression of constrains
11. Express constraints in terms of other constraints
12. Constraints of combinations of template arguments
13. Express semantics/invariants of concept models
14. The extensions shouldn't hinder other language improvements

1. A system as flexible as current templates

- Templates and dynamically typed languages have demonstrated far greater flexibility and ability to sustain reuse than systems based on types and interfaces.
  - specifying the type of arguments for operations requires exact agreement between template writer and template user – that's more foresight that we can realistically expect.
- Constraints should not require explicit declaration of argument types and be non-hierarchical.
  - built-in types, such as int and Shape* should remain first-class template arguments, requiring no workarounds.

“Overloading”

- 5. Selection of template specialization based on attributes of template arguments. It must be possible to define several templates (e.g. template functions or template classes) and to select the one to be used based on the actual template arguments. Furthermore, we must be able to specify preferences among related types and among separately developed types (class is the minimally constraining concept).

“Axioms”

- 13. Express semantics/invariants of concept models. For example, a user should be able to state that his/her array class has the property that its elements occupy contiguous storage, or that an increment followed by a decrement of his/her bidirectional iterator is a no-op.
- suggested in Dos Reis: *Generic Programming in C++: The next level*. ACCU2002]:
  - **{ Op(x, x) } == { true }**
  - **{ Op(x, y) == true } -> { Op(y, x) == true }**
  - **{ Op(x, y) == true && Op(y, z) == true } -> { Op(x, z) == true }**

“Concepts are not types”

- Conventional static (compile-time) type checking performs two roles:
  - it controls the way a programmer can invoke and combine operations (e.g., it verifies that you can add two integers and you can add two complex numbers)
  - it provides sufficient information for code to be generated (e.g., it ensures that a compiler has the information it needs to generate the exact code sequences to add two integers and to add two complex numbers)

  The need to provide the information required for code generation when writing code overconstrains the programmer’s expression of solutions.
- Implication: concept checking needs to check that code can be generated, not how

Simplify through generalization

- Consider:

```
X operator+(X,X);
X operator+(const X&, const X&);
X& operator+(X&,X&);
const X X::operator+(const X);
X X::operator+(const X&);
X X::operator+(X) const;
```

- Basically, there are 3*4*4*4==192 ways of expressing a function taking two arguments and returning a value,

The key issues are not syntactic

- “Basically, such “pseudo signatures” can express what usage-pattern can (and vise versa). Thus, the two approaches can be seen as different syntactic representations of the same idea”

- Though syntax matters immensely for the usability of concepts

Missing in 2003

- Concept_maps
  - Instead a more limited mechanism for adding attributes was envisioned:
    - **int* has_member typedef int value_type;**
  - or
    - **template<class T> operator:: (T*, value_type) = T;** // "::" as binary op. on names.
  - Alternatively, one might consider a general "rewriting system" whereby some patterns may be rewritten based on constraints/concepts**.**
  - Must deal with
    - **x.f(1);** // ok
    - **f(x,1);** // ok

2009 Reflections

- No major increase in compilation speed
  - Implied in 2003 “simple to implement” (but underestimated)
  - Implies that concept programs cannot be dramatically longer than non-concept template programs
    - Probably implies built-in concepts
  - Major vendors will insist
    - Yes, really
  - Conjecture: “major” means “<10%”

2009 Reflections

- Simple to use
  - Most C++ programmers don’t know type theory
    - For “type theory” think ‘anything written with Greek letters”
    - And will not learn it
  - Many C++ programmers dislike type theory
    - Even if they don’t know it
  - What the end user have to write must not increase significantly
    - Real Programmers™ ***hate*** writing what they consider redundant
    - It must be easy to give a reason for anything a user must write
  - Beware of the cleverness and novelty needed to publish academic papers
    - I suspect we need a lot of refinement of ideas (engineering)

2009 Reflections

- Scale
  - Composition must be easy
    - How modular?
    - Don’t damage run-time performance
  - Think in terms of million-line programs
  - Compilation, linking, and run-time speed

2009 Reflections

- We were obsessed with the STL and monoids
  - Fundamental programming ideas must be validated in several application domains
  - Grad students are not a good substitute for “Real programmers”

Heard in Bergen

- Mix concepts code and “old template” code
- Tool support is essential
  - Debugging
  - Intellisence
- Concepts as predicates?
  - Predicates on function arguments in requires specification?
  - How about exception is concept predicates?
  - How to define equivalence for concept predicates?
- Interface to theorem prover?
- The compiler know a lot about types that we can’t get to
- We need to reduce the amount of template hacking

Heard in Bergen

- The main purpose of “concepts” is not to get academic publications!
- C++’s image is a huge problem
  - Financial, high-performance, mobile
- Incremental adoption is essential
  - Concept and old-template libraries working together
  - Must be able to start small, e.g. individual class or algorithm

2009 Reflections

- Concepts for C++0x were defeated by
  - Fear of the new, fear of the unknown and unknowable
  - Fear of compilation overheads
  - Inflexibility relative to “status quo” definition/implementation of concepts
  - Fear that the specification was unstable
  - Fear of complexity (specifications and use)
  - Fear that the “status quo” design would not scale (program size and usability)
- “Schedule” was IMO only a small part of the problem
- We have to reevaluate and redesign from scratch
  - Of course we now have much more experience
  - We must work harder on simplicity (teachability and scalability)
  - We must work harder on validation
  - Implementation is no substitute for design (and vise versa)