ConceptClang Prototype Update

Larisse Voufo

Open Systems Lab
Comp. Sci. Program
SOIC, IU-Bloomington, USA

IWR - TU Dresden: 03/16/11

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Outline

1. **Concepts: Terminology and Historical Perspective**
   - Origin
   - In Relation to Generic Programming
   - Concepts-Oriented Programming
2. **Concepts:The Implementation Design Philosophies**
   - The Concepts Proposals
   - Deriving the Right Proposal
3. **ConceptClang**
   - Implementation Philosophy
   - The Prototype: Update

Concepts: Not a New Idea

- Tecton: D. Kapur, D. Musser & A. Stepanov. [1980s]
  - Alex Stepanov & Paul McJones. "Elements Of Programming". [2009]
  - Concept: groups types in terms of shared structures and properties
  - Programmer's awareness of mathematical properties
    - ==> Better programming discipline
    - ==> More code reusability and safety.
- Austern: Generic Programming and the STL [1998]
  - Documentation is Concepts-Oriented.
- J. Siek & A. Lumsdaine.
  - Boost Concepts Checking Library. [2000]
- Peter Gottschling
  - Property-Aware Programming
    - Facilitating the "exploitation" of the idea.
- In Practice: STL, BGL, MTL4, G Language (J. Siek's thesis), Adobe Open Systems, etc...

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: Not a New Idea

- Tecton: D. Kapur, D. Musser & A. Stepanov. [1980s]
  - Alex Stepanov & Paul McJones. "Elements Of Programming". [2009]
  - Concept: groups types in terms of shared structures and properties
  - Programmer's awareness of mathematical properties
    - ==> Better programming discipline
    - ==> More code reusability and safety.
- Austern: Generic Programming and the STL [1998]
  - Documentation is Concepts-Oriented.
- J. Siek & A. Lumsdaine.
  - Boost Concepts Checking Library. [2000]
- Peter Gottschling
  - Property-Aware Programming
    - Facilitating the “exploitation” of the idea.
- In Practice: STL, BGL, MTL4, G Language (J. Siek's thesis), Adobe Open Systems, etc...

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: Not a New Idea

- Tecton: D. Kapur, D. Musser & A. Stepanov. [1980s]
  - Alex Stepanov & Paul McJones. "Elements Of Programming". [2009]
  - Concept: groups types in terms of shared structures and properties
  - Programmer's awareness of mathematical properties
    - ==> Better programming discipline
    - ==> More code reusability and safety.
- Austern: Generic Programming and the STL [1998]
  - Documentation is Concepts-Oriented.
- J. Siek & A. Lumsdaine.
  - Boost Concepts Checking Library. [2000]
- Peter Gottschling
  - Property-Aware Programming
    - Facilitating the "exploitation" of the idea.
- In Practice: STL, BGL, MTL4, G Language (J. Siek's thesis), Adobe Open Systems, etc...

A Comparative Study of Support for Concepts in PLs

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil
Multi-type concepts	-	●	○	●*	○	○	○	◒
Multiple constraints	-	◒	◒	●	○†	●	●	●
Associated type access	●	●	◒	●⋆	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●
Retroactive modeling	-	●	●	●	○	○	◒	●
Type aliases	●	●	●	●	○	○	○	○
Separate compilation	○	●	◒	●	●	●	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒

*Using the multi-parameter type class extension to Haskell (Peyton Jones *et al.*, 1997).
⋆Using the functional dependencies extension to Haskell (Jones, 2000).
†Planned language additions.

Table 1: *The level of support for important properties for generic programming in the evaluated languages. A black circle indicates full support, a white circle indicates poor support, and a half-filled circle indicates partial support. The rating of "-" in the C++ column indicates that C++ does not explicitly support the feature, but one can still program as if the feature were supported due to the permissiveness of C++ templates.*

- "An extended Comparative Study of Language Support for Generic Programming". [2007]. Garcia et. al

A Comparative Study of Support for Concepts in PLs

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil
Multi-type concepts	-	●	○	●*	○	○	○	◒
Multiple constraints	-	◒	◒	●	○†	●	●	●
Associated type access	●	●	◒	●*	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●
Retroactive modeling	-	●	●	●	○	○	◒	●
Type aliases	●	●	●	●	○	○	○	○
Separate compilation	○	●	◒	●	●	●	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒

*Using the multi-parameter type class extension to Haskell (Peyton Jones *et al.*, 1997).

- C++: (almost) full support, but indirectly.

- "An extended Comparative Study of Language Support for Generic Programming".

A Comparative Study of Support for Concepts in PLs

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil
Multi-type concepts	-	●	○	●*	○	○	○	◒
Multiple constraints	-	◒	◒	●	○†	●	●	●
Associated type access	●	●	◒	●*	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●
Retroactive modeling	-	●	●	●	○	○	◒	●
Type aliases	●	●	●	●	○	○	○	○
Separate compilation	○	●	◒	●	●	●	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒

*Using the multi-parameter type class extension to Haskell (Peyton Jones *et al.*, 1997).

- Concepts == Generic Programming ?

- “An extended Comparative Study of Language Support for Generic Programming”.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming: Differs by Perspective

In a few words...

- Safe Code Reusability
- Multiplicative functionality for additive work

For Concepts:

Genericity by ...

- Value – function abstraction
- Type – (parametric or adhoc) polymorphism
- Function – functions as values
- **Structure** – requirements and operations on types
- **Property** – properties on type
- Stage – metaprogramming
- Shape – datatype-generic

– "Datatype Generic Programming". Gibbons [3]

Generic Programming: Differs by Perspective

In a few words...

- Safe Code Reusability
- Multiplicative functionality for additive work

For Concepts:

Genericity by ...

- Value – function abstraction
- Type – (parametric or adhoc) polymorphism
- Function – functions as values
- **Structure** – requirements and operations on types
- **Property** – properties on type
- Stage – metaprogramming
- Shape – datatype-generic

– "Datatype Generic Programming". Gibbons [3]

Programming w/ Concepts

- Definition:
  - Capture the common interface
  - Capture the common semantics
  - Ignore irrelevant details
- Advantages
  - Better safety, expressiveness, usability
    - Separate type checking: generic algorithm + arguments
    - better error messages
    - low barrier to entry

Concept: The Ingredients

- Requirements:
  - associated types
  - associated requirements
  - associated functions
- Modeling implementations (types)
- Generic algorithms (templates)
- Applications (template instantiations)

Programming w/ Concepts

- Definition:
  - Capture the common interface
  - Capture the common semantics
  - Ignore irrelevant details
- Advantages
  - Better safety, expressiveness, usability
    - Separate type checking: generic algorithm + arguments
    - better error messages
    - low barrier to entry

Concept: The Ingredients

- Requirements:
  - associated types
  - associated requirements
  - associated functions
- Modeling implementations (types)
- Generic algorithms (templates)
- Applications (template instantiations)

Programming w/ Concepts

- Definition:
  - Capture the common interface
  - Capture the common semantics
  - Ignore irrelevant details
- Advantages
  - Better safety, expressiveness, usability
    - Separate type checking: generic algorithm + arguments
    - better error messages
    - low barrier to entry

Concept: The Ingredients

- Requirements:
  - associated types
  - associated requirements
  - associated functions
- Modeling implementations (types)
- Generic algorithms (templates)
- Applications (template instantiations)

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming in C++: Templates

Generic Algorithm

Definition

```
template<typename InputIterator,
         typename T,
         typename BinaryOperation>
T accumulate(InputIterator first,
             InputIterator last, T init,
             BinaryOperation binary_op) {
  for (; first != last; ++first)
      init = binary_op(init, *first);
  return init;
}
```

Use

```
vector<int> v;
int i = accumulate(v.begin(),
                   v.end(), 0,
                   plus<int>());
```

accumulate: traverse a range and accumulate its elements

- an iterator for traversal
- a binary operation to accumulate

Generic Programming in C++: Templates

Generic Algorithm

Definition

```
template<typename InputIterator,
         typename T,
         typename BinaryOperation>
T accumulate(InputIterator first,
             InputIterator last, T init,
             BinaryOperation binary_op) {
  for (; first != last; ++first)
      init = binary_op(init, *first);
  return init;
}
```

Use

```
vector<int> v;
int i = accumulate(v.begin(),
                   v.end(), 0,
                   plus<int>());
```

accumulate: traverse a range and accumulate its elements

- an iterator for traversal
- a binary operation to accumulate

Generic Programming in C++: Templates

Generic Algorithm

Definition

```
template<typename InputIterator,
         typename T,
         typename BinaryOperation>
T accumulate(InputIterator first,
             InputIterator last, T init,
             BinaryOperation binary_op) {
  for (; first != last; ++first)
      init = binary_op(init, *first);
  return init;
}
```

Use

```
vector<int> v;
int i = accumulate(v.begin(),
                   v.end(), 0,
                   plus<int>());
```

Concrete Algorithm

Instantiation

- == Generate concrete code
  - at compile time,
  - if it type-checks.
- At time of first use

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming in C++: Templates

Problem: Error Capture and Diagnosis...

```
std::vector<void*> v;

std::accumulate(v.begin(), v.end(),

                         0, std::plus<int>());
```

```
/usr/include/c++/4.3/bits/stl_numeric.h: In function ‘_Tp std::accumulate(_InputIterator,
_InputIterator, _Tp, _BinaryOperation) [with _InputIterator = __gnu_cxx::__normal_iterator<void*>

std::vector<void*, std::allocator<void*> > >, _Tp = int, _BinaryOperation = std::plus<int>]’:
test.cpp:7:   instantiated from here

/usr/include/c++/4.3/bits/stl_numeric.h:117: error: invalid conversion from ‘void*’ to ‘int’

/usr/include/c++/4.3/bits/stl_numeric.h:117: error:   initializing argument 2 of
‘_Tp std::plus<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = int]’
```

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming in C++: Templates

Problem: Error Capture and Diagnosis...

```
std::vector<void*> v;
```

Type checking: not separate

- generic algorithm and arguments, both at instantiation time.
- compile error messages: hard to understand
- library code leaking to user space...

```
std::vector<void*, std::allocator<void*> > >, _Tp = int, _BinaryOperation = std::plus<int>]':
test.cpp:7:   instantiated from here

/usr/include/c++/4.3/bits/stl_numeric.h:117: error: invalid conversion from 'void*' to 'int'

/usr/include/c++/4.3/bits/stl_numeric.h:117: error:   initializing argument 2 of
'_Tp std::plus<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = int]'
```

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming in C++: Templates

Problem: Error Capture and Diagnosis...

```
std::vector<void*> v;

std::accumulate(v.begin(), v.end(),

                         0, std::plus<int>());
```

```
/usr/include/c++/4.3/bits/stl_numeric.h: In function ‘_Tp std::accumulate(_InputIterator,
_InputIterator, _Tp, _BinaryOperation) [with _InputIterator = __gnu_cxx::__normal_iterator<void*
```

```
std::vector<int> vi;
std::sort(vi.begin(), vi.end(),
         std::not_equal_to<int>());
```

Error Not Detected!

```
‘_Tp std::plus<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = int]’
```

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming in C++: Templates

Problem: Error Capture and Diagnosis...

```
std::vector<void*> v;
```

Type checking: not separate

- generic algorithm and arguments, both at instantiation time.
- compile error messages: hard to understand
- library code leaking to user space...

WORSE:

- Silent compilation!
- Uncaught semantical errors.

```
'_Tp std::plus<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = int]'
```

Generic Programming in C++: Templates

Problem: Error Capture and Diagnosis...

```
std::vector<void*> v;
```

Type checking: not separate

Further...

- w/ the indirect "support" for concepts
- library code leaking to user space...

WORSE:

- Silent compilation!
- Uncaught semantical errors.

```
'_Tp std::plus<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = int]'
```

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Generic Programming in C++: Templates

Problem: w/ the Indirect Support for Concepts

The Indirect Support

- Naming and Documentation
- Language "tricks":
  - type traits, archetypes, tag dispatching, etc...
  - cf. Boost Concept Checking Library [6]

Problems

- Language "tricks": too complex, error-prone, and limited
  - awckward design
  - poor maintainability
  - unnecessary runtime checks
  - painfully verbose code

Generic Programming in C++: Templates

Problem: w/ the Indirect Support for Concepts

The Indirect Support

- Naming and Documentation
- Language "tricks":
  - type traits, archetypes, tag dispatching, etc...
  - cf. Boost Concept Checking Library [6]

Problems

- Language "tricks": too complex, error-prone, and limited
  - awckward design
  - poor maintainability
  - unnecessary runtime checks
  - painfully verbose code

Generic Programming in C++: Templates

Problems Recap

Error Diagnosis ...

- Type checking: not separate
  - generic algorithm and arguments, both at instantiation time.
- compile error messages: hard to understand
- library code leaking to user space...

Error Capture ...

- Silent compilation!
- Uncaught semantical errors.

Indirect Support for concept ...

- Language "tricks": too complex, error-prone, and limited
  - awckward design
  - poor maintainability
  - unnecessary runtime checks
  - painfully verbose code

Generic Programming in C++: Templates

Problems Recap

Error Diagnosis ...

- Type checking: not separate
  - generic algorithm and arguments, both at instantiation time.
- compile error messages: hard to understand

Solution:

- **Add (Full) Support for Concepts!**

Indirect Support for concept ...

- Language "tricks": too complex, error-prone, and limited
  - awckward design
  - poor maintainability
  - unnecessary runtime checks
  - painfully verbose code

C++ Templates w/ Concepts

Error Capture and Diagnosis

Ideal Error Message

```
The given types do not match the concept
     BinaryOperation<std::plus<int>, void*>
```

Currently

```
std::vector<void*> v;
std::accumulate(v.begin(), v.end(),
                0, std::plus<int>());
```

```
/usr/include/c++/4.3/bits/stl_numeric.h: In func
_InputIterator, _Tp, _BinaryOperation) [with _In
std::vector<void*, std::allocator<void*> > >, _T
test.cpp:7:   instantiated from here
/usr/include/c++/4.3/bits/stl_numeric.h:117: err
/usr/include/c++/4.3/bits/stl_numeric.h:117: err
‘_Tp std::plus<_Tp>::operator()(const _Tp&, cons
```

C++ Templates w/ Concepts

Error Capture and Diagnosis

Ideal Error Message

```
The given types do not match the concept
    BinaryOperation<std::plus<int>, void*>
```

Currently

```
std::vector<void*> v;
std::accumulate(v.begin(), v.end(),
                0, std::plus<int>());
```

```
/usr/include/c++/4.3/bits/stl_numeric.h: In func
_InputIterator, _Tp, _BinaryOperation) [with _In
std::vector<void*, std::allocator<void*> > >, _T
```

Ideal Error Message

```
The given types do not match the concept
StrictWeakOrdering<std::not_equal_to<int>,
int>
```

Currently

```
std::vector<int> vi;
std::sort(vi.begin(), vi.end(),
          std::not_equal_to<int>());
```

Error Not Detected!

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

C++ Templates w/ Concepts

Error Capture and Diagnosis

Ideal Error Message

```
The given types do not match the concept
    BinaryOperation<std::plus<int>, void*>
```

Currently

```
std::vector<void*> v;
std::accumulate(v.begin(), v.end()
```

The Generic Algorithm

```
template<typename II,
         typename T,
         typename BO>
     requires InputIterator<II, T> &&
         BinaryOperation<BO, T> &&
         StrictWeakOrdering<BO, T>
T accumulate(II first, II last, T init, BO binary_op) {
   for (; first != last; ++first)
     init = binary_op(init, *first);
   return init;
}
```

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: The Terminology

Definition

```
concept C< typename T > {
    // axiom t = ...
    typename t;
    requires R<T,t>;
    void f(T x, t a);
    ...
}
```

Model: Concept map

```
concept_map C<int> {
    typedef int t;
    void f(int x, int a) {... }
    ...
}
```

Constrained Template

```
template< typename T >
        requires (C<T>)
   void foo(T x, t a) {
     f(x, a);
}
```

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

Concepts: The Terminology

Definition

```
concept C< typename T > {
    // axiom t = ...
    typename t;
    requires R<T,t>;
    void f(T x, t a);
    ...
}
```

Model: Concept map

```
concept_map C<int> {
    typedef int t;
    void f(int x, int a) {... }
    ...
}
```

Constrained Template

```
template< typename T >
        requires (C<T>)
   void foo(T x, t a) {
    f(x, a);
}
```

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: The Terminology

Definition

```
concept C< typename T > {
     // axiom t = ...
     typename t;
     requires R<T,t>;
     void f(T x, t a);
     ...
}
```

Model: Concept map

```
concept_map R<int,int> {
     ...
}
```

```
concept_map C<int> {
     typedef int t;
     void f(int x, int a) {... }
     ...
}
```

Constrained Template

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: The Terminology

Definition

```
concept C< typename T > {
     // axiom t = ...
     typename t;
     requires R<T,t>;
     void f(T x, t a);
     ...
}
```

Model: Concept map Template

- Automatic Dispatching

```
template< typename T >
        requires (R<T,int>)
concept_map C<T> {
     typedef int t;
     void f(T x, int a) {... }
     ...
}
```

Constrained Template

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: The Terminology

Refinement

```
concept C< typename T > : PC<T> {
     // axiom t = ...
     typename t;
     requires R<T,t>;
     void f(T x, t a);
     ... }
```

Model: Concept map

```
concept_map C<int> {
     typedef int t;
     void f(int x, int a) {... }
     ...
}
```

Constrained Template

```
template< typename T >
         requires (C<T>)
   void foo(T x, t a) {
     f(x, a);
}
```

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

Concepts: The Terminology

Definition

```
concept C< typename T > : PC<T> {
    // axiom t = ...
    typename t;
    requires R<T,t>;
    void f(T x, t a);
    ... }
```

Model: Concept map

```
concept_map C<int> {
    typedef int t;
    void f(int x, int a) {... }
    ...
}
```

Constrained Template

```
template< typename T >
        requires (C<T>)
   void foo(T x, t a) {
    f(x, a);
}
```

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Concepts: The Terminology

Definition

- associated types
- associated requirements
- associated functions
- Refinement
  - Concept extends requirements of another

Model: Concept map

- How a given type meets a concept's requirements
- (Automatic) Concept Dispatching

Constrained Template

- Expressing the constraints on type parameters.

Checkpoints

1. Concept Definition
   - Non-dependent check
2. Concept Map Specification
   - Requirements met?
3. Generic Algorithm Definition
   - Valid concepts?
   - Concept Coverage:
     - Check body against constraint.
4. Generic Algorithm Use.
   - Constraints Check:
     - Type matches concept?
   - Pull-in implementation

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Review

- Concept: Definition and Terminology
  - "Constraints" on types
  - A type of genericity.
- in C++: Please Support Concepts, Directly!
- Advantages:
  - Better safety, expressiveness, usability
    - Separate type checking: generic algorithm + arguments
    - better error messages
    - low barrier to entry
  - in C++: W/o hurting existing features...

Review

- Concept: Definition and Terminology
  - "Constraints" on types
  - A type of genericity.
- in C++: Please Support Concepts, Directly!
- Advantages:
  - Better safety, expressiveness, usability
    - Separate type checking: generic algorithm + arguments
    - better error messages
    - low barrier to entry
  - in C++: W/o hurting existing features...

But... How exactly?

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Several Implementation Design Philosophies

... And Why Concepts are not in C++0x.

- 2005: The "Indiana" Proposal: **"Explicit" Concepts**
  - "*Concept for C++*" [2, 4]
  - Doug Gregor, Jeremy Siek, Andrew Lumsdaine, Ronald Garcia, Jeremiah Willcock, Jaakko Jarvi, etc...
  - ConceptGCC: (Author: Doug Gregor)
    - First (and only) prototype compiler, proof-of-concept
- 2005: The "Texas" Proposal: **"Implicit" Concepts**
  - "*A Concept Design*" [8, 1]
  - Bjarne Stroustrup, Gabriel Dos Reis, etc...
- 2006 + : The "Compromise" Proposal(s)
  - "*Concepts: linguistic support for generic programming in C++*" [5]
  - All
- \- 2009: Several Issues Raised...
  - "*Simplifying the Use of Concepts*", Bjarne Stroustrup [7]
  - Philosophies: **still** diverging
  - Implementation experience (w/ ConceptGCC)
  - Final Proposal: **"Implicit" Concepts & "Explicit" Derivation**
- Jul-2009: C++ Committee Meeting: Frankfurt, Germany
  - **Voted OUT!**
  - "*Not ready, untried, too risky*" – paraphrasing Dr. Bjarne Stroustrup.

(Ref: `http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/`)

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Several Implementation Design Philosophies

... And Why Concepts are not in C++0x.

- 2005: The "Indiana" Proposal: **"Explicit" Concepts**
  - "*Concept for C++*" [2, 4]
  - Doug Gregor, Jeremy Siek, Andrew Lumsdaine, Ronald Garcia, Jeremiah Willcock, Jaakko Jarvi, etc...
  - ConceptGCC: (Author: Doug Gregor)
    - First (and only) prototype compiler, proof-of-concept
- 2005: The "Texas" Proposal: **"Implicit" Concepts**
  - "*A Concept Design*" [8, 1]
  - Bjarne Stroustrup, Gabriel Dos Reis, etc...
- 2006 + : The "Compromise" Proposal(s)
  - "*Concepts: linguistic support for generic programming in C++*" [5]
  - All
- - 2009: Several Issues Raised...
  - "*Simplifying the Use of Concepts*", Bjarne Stroustrup [7]
  - Philosophies: **still** diverging
  - Implementation experience (w/ ConceptGCC)
  - Final Proposal: **"Implicit" Concepts & "Explicit" Derivation**
- Jul-2009: C++ Committee Meeting: Frankfurt, Germany
  - **Voted OUT!**
  - "*Not ready, untried, too risky*" – paraphrasing Dr. Bjarne Stroustrup.

(Ref: `http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/`)

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Several Implementation Design Philosophies

... And Why Concepts are not in C++0x.

- 2005: The "Indiana" Proposal: **"Explicit" Concepts**
  - "*Concept for C++*" [2, 4]
  - Doug Gregor, Jeremy Siek, Andrew Lumsdaine, Ronald Garcia, Jeremiah Willcock, Jaakko Jarvi, etc...
  - ConceptGCC: (Author: Doug Gregor)
    - First (and only) prototype compiler, proof-of-concept
- 2005: The "Texas" Proposal: **"Implicit" Concepts**
  - "*A Concept Design*" [8, 1]
  - Bjarne Stroustrup, Gabriel Dos Reis, etc...
- 2006 + : The "Compromise" Proposal(s)
  - "*Concepts: linguistic support for generic programming in C++*" [5]
  - All
- - 2009: Several Issues Raised...
  - "*Simplifying the Use of Concepts*", Bjarne Stroustrup [7]
  - Philosophies: **still** diverging
  - Implementation experience (w/ ConceptGCC)
  - Final Proposal: **"Implicit" Concepts & "Explicit" Derivation**
- Jul-2009: C++ Committee Meeting: Frankfurt, Germany
  - **Voted OUT!**
  - "*Not ready, untried, too risky*" – paraphrasing Dr. Bjarne Stroustrup.

(Ref:
`http://cpp-next.com/archive/2009/08/what-happened-in-frankfurt/`)

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Texas" Proposal (in a nutshell)

Implicit Match for Concepts

"Implicit" Concepts

Definition:

- **Use Patterns** – for associated functions
- Refinement
  - Ok.

Model: Concept Map

- Not needed – Matching Implicitly

Constrained Template Definition

- Ok.

Checkpoints

1. Concept Definition
   - Ok.
2. Concept Map Specification
   - Not needed
   - Similarly to explicit template instantiation – compiler optimizations
3. Generic Algorithm Definition
   - Ok.
4. Generic Algorithm Use.
   - **Match if valid expression found.**
   - **Structural conformance**
     - Accidental conformance

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Texas" Proposal (in a nutshell)

Implicit Match for Concepts

"Implicit" Concepts

Definition:

- **Use Patterns** – for associated functions
  - Example: *x++
  - Expressions of this form should be valid.
  - **For:** Less verbose, more efficient, more general, directly mappeable from current documentations.
  - **Against:** not so efficient (?), precision and compatibility (=> unintentional matches)
- Refinement
  - Ok.

Checkpoints

1. Concept Definition
   - Ok.
2. Concept Map Specification
   - Not needed
   - Similarly to explicit template instantiation – compiler optimizations
3. Generic Algorithm Definition
   - Ok.
4. Generic Algorithm Use.
   - **Match if valid expression found.**
   - **Structural conformance**
     - Accidental conformance

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The “Indiana” Proposal (in a nutshell)

Explicit Match for Concepts

“Explicit” concepts

Definition

- **Pseudo-signatures** – for associated functions
- Refinement
    - ok

Model: Concept Map

- MUST Specify – for each matching data type

Constrained Template Definition

- Ok.

Checkpoints

1. Concept Definition
    - Ok.
2. Concept Map Specification
    - Ok
3. Generic Algorithm Definition
    - Ok.
4. Generic Algorithm Use.
    - **Match if concept map found.**
    - **Named Conformance**
    - verbose, restrictive, difficult to teach and learn...
    - Accidental conformance not necessarily bad, if it does occur (?)...

The "Indiana" Proposal (in a nutshell)

Explicit Match for Concepts

"Explicit" concepts

Definition

- **Pseudo-signatures** – for associated functions
- Example: ***T operator++()***
  - Reusing existing features: C++ type checker...
- Refinement
  - ok

Model: Concept Map

- MUST Specify – for each matching data type

Constrained Template Definition

Checkpoints

1. Concept Definition
   - Ok.
2. Concept Map Specification
   - Ok
3. Generic Algorithm Definition
   - Ok.
4. Generic Algorithm Use.
   - **Match if concept map found.**
   - **Named Conformance**
   - verbose, restrictive, difficult to teach and learn...
   - Accidental conformance not necessarily bad, if it does occur (?)...

The “Compromise” Proposal(s) (in a nutshell)

Allow both options – “Explicit” by Default

The design: Pre-Frankfurt draft

Definition

- Both:
  - “Explicit” by default
  - **“auto” keyword** – for Implicit
- **Pseudo-signatures** – for associated functions
- Refinement
  - Ok

Model: Concept Map

- Dependent on qualifier on concept definition.

Constrained Template Definition

Checkpoints

1. Concept Definition
   - Ok.
2. Concept Map Specification
   - Ok
3. Generic Algorithm Definition
   - Ok.
4. Generic Algorithm Use.
   - **Match based on qualifier on concept definition.**

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Language Philosophy

- **Flexibility and Performance**: (Abstractions over) Implementation details
- Should not be hurt by additions of features
- Easy navigation into new features
- Existing codes should take advantage
- Learning and teaching: Lower barriers to entry.

Conclusion: "Implicit" Concepts + "Explicit" Refinements.

- Save people from writing redundant concept maps,
- Teach people to directly address the semantic problems, and
- not to unnecessarily fear automatic/implicit concepts.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- Several issues raised...

  ▸ Go

  - Debugging: What if I need to debug in the middle of an implementation?
  - Subsets: What if I cannot change the implementation of a concept?
  - Automatic selection of refined implementation: not always favorable.
- Key ideas:
  - Easier to build "**explicit**" concept maps **from** "**implicit**" ones, than the other way around.
  - Default of "**explicit**" ==> A proliferation of concept maps – and a mindset that goes with them.
  - Default of "**implicit**" ==> to the need for (far fewer) "**explicit**" refinements.

Conclusion: "Implicit" Concepts + "Explicit" Refinements.

- Save people from writing redundant concept maps,
- Teach people to directly address the semantic problems, and
- not to unnecessarily fear automatic/implicit concepts.

NIVERSITY
OLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- Several issues raised...

  Go

  - Debugging: What if I need to debug in the middle of an implementation?
  - Subsets: What if I cannot change the implementation of a concept?
  - Automatic selection of refined implementation: not always favorable.

```
auto concept ContiguousIterator<typename Iter> : RandomAccessIterator<Iter> {
   requires (LvalueReference<reference> && LvalueReference<subscript_reference>)
}
template<ContiguousIterator InIter, ContiguousIterator OutIter>
         requires (SameType<InIter::value_type, OutIter::value_type> &&
POD<InIter::value_type>)
OutIter copy(InIter first, InIter last, OutIter out) {
   if (first != last)
     memmove(&*out, *&first, (last - first) * sizeof(InIter::value_type));
   return out + (last - first);
}
```

- Syntactically similar, Semantically different concepts: ContiguousIterator and RandomAccessIterator
- Call to copy() ==> Implementation for ContiguousIterator.

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- Several issues raised...

  Go

  - Debugging: What if I need to debug in the middle of an implementation?
  - Subsets: What if I cannot change the implementation of a concept?
  - Automatic selection of refined implementation: not always favorable.
    - Solution: **"Explicit" Refinement**

```
concept CB<typename T> : explicit CA<T> {
   ...
}
```

    - "If type matches CA, do not select 'up' to CB's implementation".
    - A derivation is not (also) a specialization.

The “Nail to the Coffin” (in a nutshell)

Not Both. Only “Implicit”, w/ “Explicit” Refinement ?

Analysis

- Several issues raised...

  Go

  - Debugging: What if I need to debug in the middle of an implementation?
  - Subsets: What if I cannot change the implementation of a concept?
  - Automatic selection of refined implementation: not always favorable.
    - Solution: **“Explicit” Refinement** – Example

```
concept ContiguousIterator<typename Iter> : explicit
RandomAccessIterator<Iter> {... }
concept ForwardIterator<class T> : explicit InputIterator<T> {... }
```

    - “If type matches CA, do not select ‘up’ to CB’s implementation”.
    - A derivation is not (also) a specialization.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- Several issues raised...

  Go

  - Debugging: What if I need to debug in the middle of an implementation?
  - Subsets: What if I cannot change the implementation of a concept?
  - Automatic selection of refined implementation: not always favorable.
    - Solution: **"Explicit" Refinement**

```
concept ContiguousIterator<typename Iter> : explicit
RandomAccessIterator<Iter> {... }
concept ForwardIterator<class T> : explicit InputIterator<T> {... }

//Loss of optimization?
// Consider a int* a ForwardIterator, even if it is a InputIterator ...
concept_map ForwardIterator<int*> {}
```

    - "If type matches CA, do not select 'up' to CB's implementation".
    - A derivation is not (also) a specialization.

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- Several issues raised...

  Go
- Key ideas:
  - Easier to build **"explicit"** concept maps **from "implicit"** ones, than the other way around.
  - Default of **"explicit"** ==> A proliferation of concept maps – and a mindset that goes with them.
  - Default of **"implicit"** ==> to the need for (far fewer) **"explicit"** refinements.

Conclusion: "Implicit" Concepts + "Explicit" Refinements.

- Save people from writing redundant concept maps,
- Teach people to directly address the semantic problems, and
- not to unnecessarily fear automatic/implicit concepts.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- Several issues raised...
  ▸ Go
- Key ideas:
  - Easier to build **"explicit"** concept maps **from "implicit"** ones, than the other way around.
  - Default of **"explicit"** ==> A proliferation of concept maps – and a mindset that goes with them.
  - Default of **"implicit"** ==> to the need for (far fewer) **"explicit"** refinements.

Conclusion: "Implicit" Concepts + "Explicit" Refinements.

- Save people from writing redundant concept maps,
- Teach people to directly address the semantic problems, and
- not to unnecessarily fear automatic/implicit concepts.

Coming Up w/ the Right Philosophy

The Fall of Concepts in C++0x

"Not ready, untried, too risky"

- No disagreement on **whether to add** the feature.
- Disagreements on **how to add** the feature.
- Incomplete understanding of implications from each proposal.
- Most of the analysis is abstract and unverified
- **Demand for a concrete analysis!**
  - Only working prototype: ConceptGCC – insufficient
    - Poor compile-time performance
    - Lack of some advanced features (e.g., scoped concept maps, associated templates)
  - Need production-quality implementation
    - to validate the full concepts-based standard library

Enters ME! ...

Coming Up w/ the Right Philosophy

The Fall of Concepts in C++0x

"*Not ready, untried, too risky*"

- No disagreement on **whether to add** the feature.
- Disagreements on **how to add** the feature.
- Incomplete understanding of implications from each proposal.
- Most of the analysis is abstract and unverified
- **Demand for a concrete analysis!**
  - Only working prototype: ConceptGCC – insufficient
    - Poor compile-time performance
    - Lack of some advanced features (e.g., scoped concept maps, associated templates)
  - Need production-quality implementation
    - to validate the full concepts-based standard library

Enters ME! ...

Coming Up w/ the Right Philosophy

The Fall of Concepts in C++0x

"*Not ready, untried, too risky*"

- No disagreement on **whether to add** the feature.
- Disagreements on **how to add** the feature.
- Incomplete understanding of implications from each proposal.
- Most of the analysis is abstract and unverified
- **Demand for a concrete analysis!**
  - Only working prototype: ConceptGCC – insufficient
    - Poor compile-time performance
    - Lack of some advanced features (e.g., scoped concept maps, associated templates)
  - Need production-quality implementation
    - to validate the full concepts-based standard library

Enters ME! ...

My Work: ConceptClang

The goals

1. Implement Concepts in Clang
    - ConceptGCC in a different platform
    - Support all Philosophies
    - Follow the pre-Frankfurt standard as closely as possible.
    - As first-class entities of the language.
        - Lots of previous work reuse existing features
        - Yet, still no Concept feature.
        - Why not try something different ?
2. Analyze issues raised – concretely
3. Determine a right proposal.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

My Work: ConceptClang

The goals

1. Implement Concepts in Clang
   - ConceptGCC in a different platform
   - Support all Philosophies
   - Follow the pre-Frankfurt standard as closely as possible.
   - As first-class entities of the language.
     - Lots of previous work reuse existing features
     - Yet, still no Concept feature.
     - Why not try something different ?
2. Analyze issues raised – concretely
3. Determine a right proposal.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

ConceptClang: Update

December, 2010

Trivial Concepts, Maps, and Generic Algorithms

- Empty bodies

March, 2011 – Now

1. Features Implemented and Tested
   - Concept definitions (explicit)
   - Concept maps: definitions and instantiation.
   - Associated functions
   - Concept coverage and lookup
   - Concept refinement
   - Associated requirements
   - late_check
   - Implicit concepts
   - Explicit refinement
   - Constrained templates: constraints-check
2. Features Implemented, but Probably Buggy
   - Scoped Concepts
   - Associated function template
   - Concept map templates
   - Associated types
3. In the Horizon:
   1. Most Pressing Features
      - **Concept map templates**
      - **Associated types**
      - **Concept-based overloading**
   2. Eventually
      - Use-Patterns

ConceptClang: Update

December, 2010

Trivial Concepts, Maps, and Generic Algorithms

- Empty bodies

March, 2011 – Now

1. Features Implemented and Tested
    - Concept definitions (explicit)
    - Concept maps: definitions and instantiation.
    - Associated functions
    - Concept coverage and lookup
    - Concept refinement
    - Associated requirements
    - late_check
    - Implicit concepts
    - Explicit refinement
    - Constrained templates: constraints-check
2. Features Implemented, but Probably Buggy
    - Scoped Concepts
    - Associated function template
    - Concept map templates
    - Associated types
3. In the Horizon:
    1. Most Pressing Features
        - **Concept map templates**
        - **Associated types**
        - **Concept-based overloading**
    2. Eventually
        - Use-Patterns

ConceptClang: Update

December, 2010

Trivial Concepts, Maps, and Generic Algorithms

- Empty bodies

March, 2011 – Now

1. Features Implemented and Tested
   - Concept definitions (explicit)
   - Concept maps: definitions and instantiation.
   - Associated functions
   - Concept coverage and lookup
   - Concept refinement
   - Associated requirements
   - late_check
   - Implicit concepts
   - Explicit refinement
   - Constrained templates: constraints-check
2. Features Implemented, but Probably Buggy
   - Scoped Concepts
   - Associated function template
   - Concept map templates
   - Associated types
3. In the Horizon:
   1. Most Pressing Features
      - **Concept map templates**
      - **Associated types**
      - **Concept-based overloading**
   2. Eventually
      - Use-Patterns

Use-Case Examples

1. Prototype Released: Alpha mode.
   - http://zalewski.indefero.net/p/clang/
   - Download
   - Run Tests
   - Play!
2. Foresight
   - Mini-BGL
   - stdlib

Thank You!

Gabriel Dos Reis and Bjarne Stroustrup.
Specifying c++ concepts.
SIGPLAN Not., 41:295–308, January 2006.

Jeremy Siek Douglas, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine.
Concepts for c++0x.
Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, january 2005.

Jeremy Gibbons.
Datatype-generic programming.
In *Spring School on Datatype-Generic Programming, volume 4719 of Lecture Notes in Computer Science*. Springer-Verlag.

Douglas Gregor, Jeremy Siek Douglas, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine.
Concepts for c++0x revision 1.
Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, august 2005.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine.
Concepts: linguistic support for generic programming in c++.
SIGPLAN Not., 41:291–310, October 2006.

Jeremy Siek and Andrew Lumsdaine.
Concept checking: Binding parametric polymorphism in c++.
In *IN FIRST WORKSHOP ON C++ TEMPLATE PROGRAMMING*, 2000.

Bjarne Stroustrup.
Simplifying the use of concepts.
Technical Report N2906=09-0096, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, august 2009.

Bjarne Stroustrup and Gabriel Dos Reis.
A concept design (rev. 1).
Technical Report N1782=05-0042, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, april 2005.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Language Philosophy

- **Flexibility and Performance**: (Abstractions over) Implementation details
- Should not be hurt by additions of features
- Easy navigation into new features
- Existing codes should take advantage
- Learning and teaching: Lower barriers to entry.

Back

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Debug Example

- What if I need to debug in the middle of an implementation?

```
template<typename T>
        requires (ST<T>)
void cf(T& t) {
   cerr«"Storing"«t; // ???
   store(t);
}
```

- Solution1: "Print only if you can"
    - Postpones the execution of the error message to runtime.
    - requires some cleverness
- Solution 2: Hack: *late_check*
    - No concept-check: on some area of implementation
    - Violates the spirit of interface based on checking
    - Interface change

Back

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Debug Example

- What if I need to debug in the middle of an implementation?
- Solution1: "Print only if you can"

```
struct debuglog {
   debuglog(ostream& os) : os(os) {}
   ostream& os;
   // Identity adds no constraints, but causes this to be a constrained template:
   template <typename T> requires Identity<T>
   debuglog operator«(T const&) const {os«"<unprintable>"; return *this; }

   template <typename T> requires Identity<T> && OutputStreamable<T>
   debuglog operator«(T const& x) const {os«x; return *this; }
};
```

  - Postpones the execution of the error message to runtime.
  - requires some cleverness
- Solution 2: Hack: *late_check*
  - No concept-check: on some area of implementation
  - Violates the spirit of interface based on checking
  - Interface change

Back

The “Nail to the Coffin” (in a nutshell)

Not Both. Only “Implicit”, w/ “Explicit” Refinement ?

Debug Example

- What if I need to debug in the middle of an implementation?
- Solution1: “Print only if you can”
  - Postpones the execution of the error message to runtime.
  - requires some cleverness
- Solution 2: Hack: *late_check*
  - No concept-check: on some area of implementation
  - Violates the spirit of interface based on checking
  - Interface change

Back

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Subsets

- What if I cannot change the implementation of a concept?

```
concept AB<typename T> {
   void a(T&);
   void b(T&);
};
concept A<typename T> {
   void a(T&);
};
//Obviously, every type that's an AB is also an A, so:
template<typename T>
         requires (A<T>) void f(T);
template<typename T>
         requires (AB<T>) void f(T t);
void h(X x) // X is a type for which a(x) is valid
{
   f(x); // ambiguous
}
```

- A Solution:
  - Inside h? Local concept map not allowed.
  - Outside h? Leaking implementation details + Impossible (?)

▸ Back

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Subsets

- What if I cannot change the implementation of a concept?

```
concept AB<typename T> {
   void a(T&);
   void b(T&);
};
concept A<typename T> {
   void a(T&);
};
//Obviously, every type that's an AB is also an A, so:
template<typename T>
        requires (A<T>) void f(T);
template<typename T>
        requires (AB<T>) void f(T t);
void h(X x) // X is a type for which a(x) is valid
{
   f(x); // ambiguous
}
```

- A Solution:

```
template<typename T> requires (AB<T>) concept_map A<T> {}
```

  - Inside h? Local concept map not allowed.
  - Outside h? Leaking implementation details + Impossible (?)

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Subsets

- What if I cannot change the implementation of a concept?

```
concept AB<typename T> {
   void a(T&);
   void b(T&);
};
concept A<typename T> {
   void a(T&);
};
//Obviously, every type that's an AB is also an A, so:
template<typename T>
         requires (A<T>) void f(T);
template<typename T>
         requires (AB<T>) void f(T t);
void h(X x) // X is a type for which a(x) is valid
{
   f(x); // ambiguous
}
```

- A Solution: – Impossible in current wording
  - Inside h? Local concept map not allowed.
  - Outside h? Leaking implementation details + Impossible (?)

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

When implicit concepts are insufficient

- Automatic selection of refined implementation is not always favorable.

```
auto concept ContiguousIterator<typename Iter> : RandomAccessIterator<Iter> {
   requires (LvalueReference<reference> && LvalueReference<subscript_reference>)
}
template<ContiguousIterator InIter, ContiguousIterator OutIter>
         requires (SameType<InIter::value_type, OutIter::value_type> &&
POD<InIter::value_type>)
OutIter copy(InIter first, InIter last, OutIter out) {
   if (first != last)
     memmove(&*out, *&first, (last - first) * sizeof(InIter::value_type));
   return out + (last - first);
}
```

  - Syntactically similar, Semantically different concepts: ContiguousIterator and RandomAccessIterator
  - Call to copy() ==> Implementation for ContiguousIterator.

- Generalization:
- Solution: **"Explicit" Refinement**
  - "If type matches CA, do not select 'up' to CB's implementation".
  - A derivation is not (also) a specialization.

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

Back

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

When implicit concepts are insufficient

- Automatic selection of refined implementation is not always favorable.
- Generalization:
  1. Programmer **A** defines concept **CA**.
  2. Programmer **B** defines concept **CB** derived from **CA**.
     - syntactically very similar, yet semantically different
  3. Programmer **U** manages to use a type **T** somehow meant to be **CA** as a **CB**.

  - **A** does not know about **B** or **U**.
  - **B** knows about **CB** and **CA**
    - may not be able to modify **CA**.
  - **U** may only know about **CA** and **CB**,
    - and would rather know as little as possible.

  1. What can **B** do to protect **U** ?
  2. What can language designers do to "remind **B** to protect **U**"
     - and to help **U** if **B** forgets?
- Solution: **"Explicit" Refinement**
  - "If type matches CA, do not select 'up' to CB's implementation".
  - A derivation is not (also) a specialization.

Back

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

When implicit concepts are insufficient

- Automatic selection of refined implementation is not always favorable.
- Generalization:
- Solution: **"Explicit" Refinement**

```
concept CB<typename T> : explicit CA<T> {
    ...
}
```

- "If type matches CA, do not select 'up' to CB's implementation".
- A derivation is not (also) a specialization.

Back

The “Nail to the Coffin” (in a nutshell)

Not Both. Only “Implicit”, w/ “Explicit” Refinement ?

When implicit concepts are insufficient

- Automatic selection of refined implementation is not always favorable.
- Generalization:
- Solution: **“Explicit” Refinement** – Example

```
concept ContiguousIterator<typename Iter> : explicit RandomAccessIterator<Iter> {... }
concept ForwardIterator<class T> : explicit InputIterator<T> {... }
```

  - “If type matches CA, do not select ‘up’ to CB’s implementation”.
  - A derivation is not (also) a specialization.

Back

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

When implicit concepts are insufficient

- Automatic selection of refined implementation is not always favorable.
- Generalization:
- Solution: **"Explicit" Refinement**

```
concept ContiguousIterator<typename Iter> : explicit RandomAccessIterator<Iter> {... }
concept ForwardIterator<class T> : explicit InputIterator<T> {... }

//Loss of optimization?
// Consider a int* a ForwardIterator, even if it is a InputIterator ...
concept_map ForwardIterator<int*> {}
```

  - "If type matches CA, do not select 'up' to CB's implementation".
  - A derivation is not (also) a specialization.

Back

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- There are several other issues...
- Key ideas:
  - Easier to build **"explicit"** concept maps **from "implicit"** ones, than the other way around.
  - Default of **"explicit"** ==> A proliferation of concept maps – and a mindset that goes with them.
  - Default of **"implicit"** ==> to the need for (far fewer) **"explicit"** refinements.

Conclusion: "Implicit" Concepts + "Explicit" Refinements.

- Save people from writing redundant concept maps,
- Teach people to directly address the semantic problems, and
- not to unnecessarily fear automatic/implicit concepts.

Back

The "Nail to the Coffin" (in a nutshell)

Not Both. Only "Implicit", w/ "Explicit" Refinement ?

Analysis

- There are several other issues...
- Key ideas:
  - Easier to build **"explicit"** concept maps **from "implicit"** ones, than the other way around.
  - Default of **"explicit"** ==> A proliferation of concept maps – and a mindset that goes with them.
  - Default of **"implicit"** ==> to the need for (far fewer) **"explicit"** refinements.

Conclusion: "Implicit" Concepts + "Explicit" Refinements.

- Save people from writing redundant concept maps,
- Teach people to directly address the semantic problems, and
- not to unnecessarily fear automatic/implicit concepts.

Back

INDIANA UNIVERSITY
PERVASIVE TECHNOLOGY INSTITUTE