C++ Extensions for Concepts
A Bottom-Up View

Simon Brand

Codeplay Software Ltd.

August 5, 2015

Introduction

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:
  - The Problem

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:
  - The Problem
  - An Introduction to SFINAE

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:
  - The Problem
  - An Introduction to SFINAE
  - Expression SFINAE

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:
  - The Problem
  - An Introduction to SFINAE
  - Expression SFINAE
  - Concepts

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:
  - The Problem
  - An Introduction to SFINAE
  - Expression SFINAE
  - Concepts
- Trigger warning: contains advanced template metaprogramming.

Introduction

- This talk aims to cover what the "C++ Extensions for Concepts" proposal is, why it is needed and how an implementation might be used.
- Split into four parts:
  - The Problem
  - An Introduction to SFINAE
  - Expression SFINAE
  - Concepts
- Trigger warning: contains advanced template metaprogramming.
- I assume at least a basic knowledge of C++ templates, but more than that is probably helpful in getting the most out of this talk.

Outline

The Problem

The Problem

- The two main categories of polymorphism in C++ are:

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates
    - Relies on implicit interfaces

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates
    - Relies on implicit interfaces
- The problem with implicit interfaces is in the name: they don't have a simple, concrete definition.

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates
    - Relies on implicit interfaces
- The problem with implicit interfaces is in the name: they don't have a simple, concrete definition.
- This limits compiler diagnostics.

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates
    - Relies on implicit interfaces
- The problem with implicit interfaces is in the name: they don't have a simple, concrete definition.
- This limits compiler diagnostics.
- Some interfaces might just be enforced by convention.

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates
    - Relies on implicit interfaces
- The problem with implicit interfaces is in the name: they don't have a simple, concrete definition.
- This limits compiler diagnostics.
- Some interfaces might just be enforced by convention.
- We want a way to make these implicit interfaces explicit.

The Problem

- The two main categories of polymorphism in C++ are:
  - Run-Time Polymorphism
    - Virtual functions
    - Relies on explicit interfaces
  - Compile-Time Polymorphism
    - Templates
    - Relies on implicit interfaces
- The problem with implicit interfaces is in the name: they don't have a simple, concrete definition.
- This limits compiler diagnostics.
- Some interfaces might just be enforced by convention.
- We want a way to make these implicit interfaces explicit.
- Iterators are a good example.

Outline

SFIN-what?

SFIN-what?

- SFINAE stands for Substitution Failure is not an Error.

SFIN-what?

- SFINAE stands for Substitution Failure is not an Error.
- In certain circumstances, template substitution failures will result in that instance being removed from the overload candidate set rather than causing a hard compiler error.

SFIN-what?

- SFINAE stands for Substitution Failure is not an Error.
- In certain circumstances, template substitution failures will result in that instance being removed from the overload candidate set rather than causing a hard compiler error.
- SFINAE occurs in:

SFIN-what?

- SFINAE stands for Substitution Failure is not an Error.
- In certain circumstances, template substitution failures will result in that instance being removed from the overload candidate set rather than causing a hard compiler error.
- SFINAE occurs in:
  - All types and expressions used in the function signature.

SFIN-what?

- SFINAE stands for Substitution Failure is not an Error.
- In certain circumstances, template substitution failures will result in that instance being removed from the overload candidate set rather than causing a hard compiler error.
- SFINAE occurs in:
  - All types and expressions used in the function signature.
  - All types and expressions used in the template parameter declaration.

```
1  #include <iostream>
   #include <vector>
3
   template <typename T>
5  void foo(T) { std::cout << "foo 1\n"; }

7  template <typename T>
   void foo(typename T::value_type) { std::cout << "foo 2\n"; }
9
   int main() {
11     foo<int>(1);
       foo<std::vector<int>>(1);
13 }
```

```
#include <iostream>
#include <stack>
#include <map>

template <typename T, typename T::mapped_type* = nullptr>
void foo(T) { std::cout << "foo 1\n"; }

template <typename T, typename T::container_type* = nullptr>
void foo(T) { std::cout << "foo 2\n"; }

int main() {
    foo(std::map<int,int>{});
    foo(std::stack<int>{});
}
```

Great, can I abuse it horribly?

Great, can I abuse it horribly?
I'm glad you asked!

```
#include <iostream>
#include <type_traits>

template <typename T,
          typename std::enable_if<std::is_floating_point<T>::value>::type* =
          nullptr>
void foo(T t) { std::cout << "foo float\n"; }

template <typename T,
          typename std::enable_if<std::is_integral<T>::value>::type* = nullptr>
void foo(T t) { std::cout << "foo int\n"; }

int main() {
    foo(1);
    foo(1.0);
}
```

```
1 template <bool Cond, typename T = void>
  struct enable_if {};
3
  template <typename T>
5 struct enable_if<true,T>
  { using type = T; };
```

What's the worst we can do?

What's the worst we can do?

Well, expression SFINAE is pretty awful.

Outline

What is Expression SFINAE?

What is Expression SFINAE?

- Bog-standard, plebian SFINAE allows you to control the elimination of overloads from candidate sets by checking qualities of the types used.

What is Expression SFINAE?

- Bog-standard, plebian SFINAE allows you to control the elimination of overloads from candidate sets by checking qualities of the types used.
- Expression SFINAE allows you to do the same by checking the validity of any C++ expression on those types.

What is Expression SFINAE?

- Bog-standard, plebian SFINAE allows you to control the elimination of overloads from candidate sets by checking qualities of the types used.
- Expression SFINAE allows you to do the same by checking the validity of any C++ expression on those types.
- Note, this is not supported in Visual Studio, because Visual Studio.

```
#include <iostream>

struct Chicken{};

void process (Chicken){}
void process (int){}

template <typename T>
void tracedProcess (T t){
    process(t);
    std::cout << "Processed value" << t << std::endl;
}

int main() {
    int i = 10;
    Chicken c;
    std::string s = "Chicken template library";

    tracedProcess(i);
    tracedProcess(c);
    tracedProcess(s);
}
```

```
#include <iostream>

struct Chicken{};

void process (Chicken){}
void process (int){}

template <typename T>
void tracedProcess (T t, ...)
{ std::cout << "Cannot process given type\n"; }

template <typename T>
auto tracedProcess (T t, int) ->
    decltype(process(t), std::cout << t, void()) {
    process(t);
    std::cout << "Processed value " << t << std::endl;
}

int main() {
    int i = 10;
    Chicken c;
    std::string s = "Chicken template library";

    tracedProcess(i, 0);
    tracedProcess(c, 0);
    tracedProcess(s, 0);
}
```

```
1  #include <iostream>
2
3  struct Chicken{}; void process (Chicken){}
   void process (int){}
5
   template <typename T> void tracedProcess (T t, ...)
7  { std::cout << "Cannot process given type\n"; }
8
9  template <typename T> auto tracedProcess (T t, char) ->
       decltype(process(t), void()) {
11     process(t);
       std::cout << "Processed value, but can't output it\n";
13 }
14
15 template <typename T> auto tracedProcess (T t, int) ->
       decltype(process(t), std::cout << t, void()) {
17     process(t);
       std::cout << "Processed value " << t << std::endl;
19 }
20
21
   int main() {
23     int i = 10;
       Chicken c;
25     std::string s = "Chicken template library";
       tracedProcess(i, 0);
27     tracedProcess(c, 0);
       tracedProcess(s, 0);
29 }
```

```
1  #include <iostream>
2
3  struct A
   { virtual void foo() = 0; };
5
   template <typename T>
7  struct B : A {
       //this if instantiation is valid
9      void foo() override {
           std::cout << T{};
11     }
12
13     //otherwise this
       //virtual void foo() = 0;
15 };
16
17 class C : public B<void>
   { void foo() override {} };
19
   int main() {
21     A *a = new B<std::string>{}; //all good
       A *b = new B<void>{}; //should fail to compile
23     A *c = new C{}; //fails to compile, we want it to succeed
   }
```

```
template <typename...> struct voider { using type = void; };
template <typename... Args> using void_t = typename voider<Args...>::type;

template <template <typename...> class T, typename, typename... Args>
struct is_detected_impl { using type = std::false_type; };

template <template <typename...> class T, typename... Args>
struct is_detected_impl<T, void_t<T<Args...>>, Args...>
{ using type = std::true_type; };

template <template <typename...> class T, typename... Args>
using is_detected = typename is_detected_impl<T, void, Args...>::type;
```

```cpp
#include <iostream>
#include "is_detected.hpp"

template <typename T>
using default_constructor_foo_t = decltype(T{}, std::cout << T{});

template <typename T>
using has_default_constructor_foo = is_detected<default_constructor_foo_t, T>;

struct A
{ virtual void foo() = 0; };

template <typename T, typename = has_default_constructor_foo<T> >
struct B : A {};

template <typename T>
struct B <T, std::true_type> : A
{ void foo() override { std::cout << T{}; } };

class C : public B<void>
{ void foo() override {} };

int main() {
    A *a = new B<std::string>{}; //all good
    A *b = new B<void>{}; //should fail to compile
    A *c = new C{}; //succeeds!
}
```

```
exprsfinae3.cpp: In function 'int main()':
exprsfinae3.cpp:28:23: error: no matching function for call to 'tracedProcess(
    std::__cxx11::string&, int)'
     tracedProcess(s, 0);
                       ^

exprsfinae3.cpp:15:28: note: candidate: template<class T> decltype (((process(t)
    , (std::cout << t)), void())) tracedProcess(T, int)
 template <typename T> auto tracedProcess (T t, int) ->
                            ^
exprsfinae3.cpp:15:28: note:   template argument deduction/substitution failed:
exprsfinae3.cpp: In substitution of 'template<class T> decltype (((process(t), (
    std::cout << t)), void())) tracedProcess(T, int)  [with T = std::__cxx11::
    basic_string<char>]':
exprsfinae3.cpp:28:23:   required from here
exprsfinae3.cpp:16:21: error: no matching function for call to 'process(std::
    __cxx11::basic_string<char>&)'
     decltype(process(t), std::cout << t, void()) {
                     ^

exprsfinae3.cpp:3:24: note: candidate: void process(Chicken)
 struct Chicken{}; void process (Chicken){}
                        ^

exprsfinae3.cpp:3:24: note:   no known conversion for argument 1 from 'std::
    __cxx11::basic_string<char>' to 'Chicken'
exprsfinae3.cpp:4:6: note: candidate: void process(int)
 void process (int){}
      ^

exprsfinae3.cpp:4:6: note:   no known conversion for argument 1 from 'std::
    __cxx11::basic_string<char>' to 'int'
```

Outline

A Trip to the Past

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.
    - E.g. does a function exist, does this expression result in that type, does a class have this member.

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.
    - E.g. does a function exist, does this expression result in that type, does a class have this member.
    - Checked statically.

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.
    - E.g. does a function exist, does this expression result in that type, does a class have this member.
    - Checked statically.
  - Axioms

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.
    - E.g. does a function exist, does this expression result in that type, does a class have this member.
    - Checked statically.
  - Axioms
    - Puts restrictions on semantic qualities of types.

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.
    - E.g. does a function exist, does this expression result in that type, does a class have this member.
    - Checked statically.
  - Axioms
    - Puts restrictions on semantic qualities of types.
    - E.g. is *operator* $==$ associative, is *operator*$+$ commutative, is *operator* $>$ the opposite of *operator* $<$

A Trip to the Past

- *Concepts* were proposed as an extension for C++11, but were removed before standardisation as they were deemed not ready.
- Broadly split into *constraints* and *axioms*.
  - Constraints
    - Puts restrictions on syntactic qualities of types.
    - E.g. does a function exist, does this expression result in that type, does a class have this member.
    - Checked statically.
  - Axioms
    - Puts restrictions on semantic qualities of types.
    - E.g. is *operator* == associative, is *operator*+ commutative, is *operator* > the opposite of *operator* <
    - Not checked by the compiler.

A Trip to the Future

A Trip to the Future

- There are currently no official plans to support full concepts in upcoming C++ releases.

A Trip to the Future

- There are currently no official plans to support full concepts in upcoming C++ releases.
- There is a proposal for C++ Extension for Concepts (previously known as Concepts Lite) which currently exists as a Technical Specification proposal.

A Trip to the Future

- There are currently no official plans to support full concepts in upcoming C++ releases.
- There is a proposal for C++ Extension for Concepts (previously known as Concepts Lite) which currently exists as a Technical Specification proposal.
- Essentially the constraints element of the original proposal.

A Trip to the Future

- There are currently no official plans to support full concepts in upcoming C++ releases.
- There is a proposal for C++ Extension for Concepts (previously known as Concepts Lite) which currently exists as a Technical Specification proposal.
- Essentially the constraints element of the original proposal.
- Full concepts *might* make it into C++17.

A Trip to the Future

- There are currently no official plans to support full concepts in upcoming C++ releases.
- There is a proposal for C++ Extension for Concepts (previously known as Concepts Lite) which currently exists as a Technical Specification proposal.
- Essentially the constraints element of the original proposal.
- Full concepts *might* make it into C++17.
- Visual Studio support expected in 2087.

```
1  #include <iostream>
2
3  struct Chicken{}; void process (Chicken){}
   void process (int){}
5
   template<typename T> concept bool Processable =
7      requires (T t) { { process(t) } -> void; };
   template<typename T> concept bool Coutable = requires (T t) { std::cout << t; };
9
   template <typename T> void tracedProcess (T t, ...)
11 { std::cout << "Cannot process given type\n"; }
12
13 template <Processable T> void tracedProcess (T t, char) {
       process(t);
15     std::cout << "Processed value, but can't output it\n";
   }
17
   template <typename T> requires Processable<T> && Coutable<T>
19 void tracedProcess (T t, int) {
       process(t);
21     std::cout << "Processed value " << t << std::endl;
   }
23
   int main() {
25     int i = 10; Chicken c; std::string s = "Chicken template library";
26
27     tracedProcess(i, 0);
       tracedProcess(c, 0);
29     tracedProcess(s, 0);
   }
```

```
exprsfinae3.cpp: In function 'int main()':
exprsfinae3.cpp:28:23: error: no matching function for call to 'tracedProcess(
    std::__cxx11::string&, int)'
     tracedProcess(s, 0);
                       ^
exprsfinae3.cpp:15:28: note: candidate: template<class T> decltype (((process(t)
    , (std::cout << t)), void())) tracedProcess(T, int)
 template <typename T> auto tracedProcess (T t, int) ->
                            ^
exprsfinae3.cpp:15:28: note:   template argument deduction/substitution failed:
exprsfinae3.cpp: In substitution of 'template<class T> decltype (((process(t), (
    std::cout << t)), void())) tracedProcess(T, int)  [with T = std::__cxx11::
    basic_string<char>]':
exprsfinae3.cpp:28:23:   required from here
exprsfinae3.cpp:16:21: error: no matching function for call to 'process(std::
    __cxx11::basic_string<char>&)'
     decltype(process(t), std::cout << t, void()) {
                     ^
exprsfinae3.cpp:3:24: note: candidate: void process(Chicken)
 struct Chicken{}; void process (Chicken){}
                        ^
exprsfinae3.cpp:3:24: note:   no known conversion for argument 1 from 'std::
    __cxx11::basic_string<char>' to 'Chicken'
exprsfinae3.cpp:4:6: note: candidate: void process(int)
 void process (int){}
      ^
exprsfinae3.cpp:4:6: note:   no known conversion for argument 1 from 'std::
    __cxx11::basic_string<char>' to 'int'
```

```
exprsfinaeconcept.cpp: In function 'int main()':
exprsfinaeconcept.cpp:34:23: error: cannot call function 'void tracedProcess(T,
     int) [with T = std::__cxx11::basic_string<char>]'
     tracedProcess(s, 0);
                       ^
exprsfinaeconcept.cpp:22:6: note:   constraints not satisfied
 void tracedProcess (T t, int) {
      ^
exprsfinaeconcept.cpp:22:6: note:   'Processable<T>' evaluated to false
exprsfinaeconcept.cpp:22:6: note:   'Processable<T>' evaluated to false
```

```
1  #include <iostream>
2
3  struct A
   { virtual void foo() = 0; };
5
   template <typename T>
7  struct B : A {};
8
9  template <typename T>
     requires requires (T t) { std::cout << T{}; }
11 struct B<T> : A
   { void foo() override { std::cout << T{}; } };
13
   class C : public B<void>
15 { void foo() override {} };
16
17 int main() {
       A *a = new B<std::string>{}; //all good
19     A *b = new B<void>{}; //should fail to compile
       A *c = new C{}; //succeeds!
21 }
```

The End!

The End!
Any questions?