Generic Programming & Templates

Nick Sumner
wsumner@sfu.ca
Generic Programming

- Recall: *Generic programming* is the idea that an algorithm should be written only once.
Generic Programming

- Recall: **Generic programming** is the idea that an algorithm should be written only once.
  - *Elements* of an algorithm that vary should be abstracted away.
Generic Programming

- Recall: **Generic programming** is the idea that an algorithm should be written only once.
  - *Elements* of an algorithm that vary should be abstracted away.
  - An algorithm can be instantiated by filling in these parameters later
Generic Programming

- Recall: **Generic programming** is the idea that an algorithm should be written only once.
  - *Elements* of an algorithm that vary should be abstracted away.
  - An algorithm can be instantiated by filling in these parameters later.
- This should immediately make you think: “Polymorphism”
  - We already called this **parametric polymorphism**
Generic Programming

- Recall: **Generic programming** is the idea that an algorithm should be written only once.
  - *Elements* of an algorithm that vary should be abstracted away.
  - An algorithm can be instantiated by filling in these parameters later.
- This should immediately make you think: “Polymorphism”
  - We already called this *parametric polymorphism*.
- In C++, this is done through templates.
Generic Programming

- Recall: *Generic programming* is the idea that an algorithm should be written only once.
  - *Elements* of an algorithm that vary should be abstracted away.
  - An algorithm can be instantiated by filling in these parameters later.
- This should immediately make you think: “Polymorphism”
  - We already called this *parametric polymorphism*.
- In C++, this is done through templates
  - Generics in Java & C#
Several different constructs can be templated...
Variable, Type, & Function Templates

```cpp
template<
  typename T
>
constexpr T PI = T(3.14159265358979323846)
```
template<typename T>
constexpr T PI = T(3.14159265358979323846)
Variable, Type, & Function Templates

template<typename T>
constexpr T PI = T(3.14159265358979323846)
template<typename T>
constexpr T PI = T(3.14159265358979323846)

float radius = ...
float area = PI<float> * radius * radius;
Variable, Type, & Function Templates

template<typename T>
struct pair {
    pair(const T& first, const T& second)
        : first{first},
          second{second}
    {}

    T first;
    T second;
};
template<typename T>
struct pair {
    pair(const T& first, const T& second) :
        first{first},
        second{second}
    {
    }

    T first;
    T second;
};

pair<Kitten> kittenPair = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
Variable, Type, & Function Templates

template<typename T>
const T&
min(const T& first, const T& second) {
    if (first < second) {
        return first;
    }
    return second;
};
Variable, Type, & Function Templates

template<typename T> 
const T& 
min(const T& first, const T& second) { 
  if (first < second) { 
    return first; 
  } 
  return second; 
};

int smaller = min<int>(1,2);
template<typename T>
const T&
min(const T& first, const T& second) {
    if (first < second) {
        return first;
    }
    return second;
};

But *something* about this should feel odd! (Apart from min already existing)

int smaller = min<int>(1,2);
Variable, Type, & Function Templates

- Several different constructs can be templated...
  - Variables
  - Classes
  - Functions
Variable, Type, & Function Templates

- Several different constructs can be templated...
  - Variables
  - Classes
  - Functions
  - Type aliases *(using)*
Several different constructs can be templated...

- Variables
- Classes
- Functions
- Type aliases (using)
- Member functions
Variable, Type, & Function Templates

- Several different constructs can be templated...
  - Variables
  - Classes
  - Functions
  - Type aliases (using)
  - Member functions
  - All of the above inside another template...
Template Argument Deduction

- In many places, template arguments can be deduced from context.
Template Argument Deduction

- In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
```
Template Argument Deduction

- In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};
```

Requires C++17
In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};
```

Uses the constructor as a guide for deduction.
In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};

int smaller = min<int>(1,2);
int smaller = min(1,2);
```
Template Argument Deduction

- In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};

typedef int smaller = min<int>(1, 2);
```

- Can only deduce based on function arguments
Template Argument Deduction

- In many places, template arguments can be deduced from context.

```
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};
```

- Can only deduce based on function arguments

```
int smaller = min<int>(1,2);
int smaller = \textcolor{red}{\textbf{min}}(1,2);
```

Requires C++17
Template Argument Deduction

- In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};

int smaller = min<int>(1,2);
int smaller = min(1,2);

vector from = {0, 1, 2, 3, 4, 5};
vector to = {0, 0, 0, 0, 0, 0};
copy(from.begin(), from.end(), to.begin());
```
In many places, template arguments can be deduced from context.

```cpp
pair<Kitten> kittens = {
    Kitten{"Pawsley"}, Kitten{"Steven"}
};
pair moreKittens = {Kitten{"Lionel"}, Kitten{"J"}};

int smaller = min<int>(1, 2);
int smaller = min(1, 2);

vector from = {0, 1, 2, 3, 4, 5};
vector to   = {0, 0, 0, 0, 0, 0, 0};
copy(from.begin(), from.end(), to.begin());
```

If types cannot be exactly deduced, they must be given
Parameters: Types, Literals, Templates

- Templates may parameterized on more than types!
Parameters: Types, Literals, Templates

- Templates may parameterized on more than types!
  - Literals: integers, (function) pointers, references, enums
Templates may parameterized on more than types!
- Literals: integers, (function) pointers, references, enums

```cpp
tuple<Kitten, Age, Lethality> kittenRecord = {
   Kitten{"Bitey McBiterson"}, 10, Lethality::TOTAL
};
```
Templates may parameterized on more than types!
  - Literals: integers, (function) pointers, references, enums

```
tuple<Kitten, Age, Lethality> kittenRecord = {
  Kitten{"Bitey McBiterson"}, 10, Lethality::TOTAL
};
auto lethality = std::get<2>(kittenRecord);
```
Parameters: Types, Literals, Templates

- Templates may parameterized on more than types!
  - Literals: integers, (function) pointers, references, enums

```cpp
tuple<Kitten, Age, Lethality> kittenRecord = {
    Kitten{"Bitey McBiterson"}, 10, Lethality::TOTAL
};
auto lethality = std::get<2>(kittenRecord);

array<Kitten, 10> kittens;
```
Templates may parameterized on more than types!
- Literals: integers, (function) pointers, references, enums

```cpp
tuple<Kitten, Age, Lethality> kittenRecord = {
    Kitten{"Bitey McBiterson"}, 10, Lethality::TOTAL
};
auto lethality = std::get<2>(kittenRecord);
```

```cpp
array<Kitten, 10> kittens;
```

What do you think the declaration of std::array looks like?
Templates may parameterized on more than types!

- Literals: integers, (function) pointers, references, enums

```cpp
tuple<Kitten, Age, Lethality> kittenRecord = {
    Kitten{"Bitey McBiterson"}, 10, Lethality::TOTAL
};
auto lethality = std::get<2>(kittenRecord);
```

```cpp
array<Kitten, 10> kittens;
```

```cpp
template<class T, std::size_t N>
struct array {
    T data[N];
};
```
Parameters: Types, Literals, Templates

- Templates may parameterized on more than types!
  - Literals: integers, (function) pointers, references, enums
  - Templates (less common in practice)

```cpp
template<template <class> class CreationPolicy>
struct WidgetLab {
  ...
};
```
Parameters: Types, Literals, Templates

- Templates may parameterized on more than types!
  - Literals: integers, (function) pointers, references, enums
  - Templates (less common in practice)

```cpp
template<template <class> class CreationPolicy>
struct WidgetLab {
...
};
```

Suppose WidgetLab uses & creates Widgets. Why is the CreationPolicy a template?
Parameters: Types, Literals, Templates

- Templates may parameterized on more than types!
  - Literals: integers, (function) pointers, references, enums
  - Templates (less common in practice)

- Thought experiment:
  How do I write a function that takes a lambda?
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important
- Templates can have default arguments
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important
- Templates can have default arguments

```cpp
template<class T=std::string,
        class C=std::vector<T>,
        auto size=10>
class SmallRoster { ... };
```
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important
- Templates can have default arguments

```cpp
template<class T=std::string,
         class C=std::vector<T>,
         auto size=10>
class SmallRoster { ... };

SmallRoster<Kitten> teamKittens;
SmallRoster<> teamStrings;
```
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important
- Templates can have default arguments
- Methods (& constructors) can be templated
  - You saw this on the first day!
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.

- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important

- Templates can have default arguments

- Methods (& constructors) can be templated
  - You saw this on the first day!
  - You may need to specify explicit templates

```cpp
template<typename T>
void foo() {
    Object<T> foo;
    foo.template someMethod<int>();
}
```
Pragmatic Usage Issues

- The complete definition of a template must be available before a template is instantiated.
- Templates are not type checked until instantiated.
  - Having uses of your templates to test them is important
- Templates can have default arguments
- Methods (& constructors) can be templated
  - You saw this on the first day!
  - You may need to specify explicit templates
- Some ambiguous nested types must be specified w/ typename

```cpp
T::iterator * p;
typename T::iterator * p;
```
Specialization

- Sometimes you want a type to behave differently for different parameters
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
  - Optimization (e.g. operation X on a Matrix can be ...)
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
  - Optimization (e.g. operation X on a Matrix can be ...)
  - Correctness constraints
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
  - Optimization (e.g. operation $X$ on a Matrix can be ...)
  - Correctness constraints
  - Strongly decoupled interfaces
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
  - Optimization (e.g. operation X on a Matrix can be ...)
  - Correctness constraints
  - Strongly decoupled interfaces

- This is achieved through template specialization
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
  - Optimization (e.g. operation X on a Matrix can be ...)
  - Correctness constraints
  - Strongly decoupled interfaces

- This is achieved through template specialization
  - Declaring a special variant of a template for known parameters
Specialization

- Sometimes you want a type to behave differently for different parameters
  - Generic implementation with guides where necessary
  - Optimization (e.g. operation X on a Matrix can be ...)
  - Correctness constraints
  - Strongly decoupled interfaces

- This is achieved through template specialization
  - Declaring a special variant of a template for known parameters

Consider having `std::hash` do the right thing custom types.
namespace std {
  template< class Key >
  struct hash;
}
This doesn’t implement hashing for custom types. What if I want to add a `Cat` to an `unordered_set`?
This doesn't implement hashing for custom types. What if I want to add a `Cat` to an `unordered_set`?
### Specialization

**<functional>**

```cpp
namespace std {
    template< class Key >
    struct hash;
}
```

**<Cats.h>**

```cpp
namespace std {
    template<>
    struct hash<Cat> {
        std::size_t
        operator()(Cat const& s) const noexcept {
            return ...;
        }
    };
}
```
Standard Library

### Functional

#### namespace std {
```cpp
  template< class Key >
  struct hash;
```
}

#### <Cats.h>
```cpp
namespace std {
  template<>
  struct hash<Cat> {
    std::size_t
    operator()(Cat const& s) const noexcept {
      return ...;
    }
  };
}
std::unordered_set<Cat> bigBagOfCats;
```
Things start to get strange.
Things start to get strange.

template <unsigned N>
struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
Specialization

- Things start to get strange.

```cpp
template <unsigned N> struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};

cout << Fib<7>::value << "\n";
```
Things start to get strange.

```cpp
template <unsigned N>
struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
```

cout << Fib<7>::value << "\n";

This prints 13.
The value is computed at compile time!
Things start to get strange.

```cpp
template <unsigned N>
struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
```

This prints 13.
The value is computed at compile time!
Specialization

- Things start to get strange.

```cpp
template <unsigned N>
struct Fib {
    static constexpr unsigned value = 
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
```

This prints 13.
The value is computed at compile time!
Things start to get strange.

This prints 13.
The value is computed at compile time!
Specialization

- Things start to get strange.

```
template <unsigned N>
struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
```

This prints 13. The value is computed at compile time!

```
struct Fib<2> {
    value = ...
};

struct Fib<3> {
    value = ...
};

struct Fib<4> {
    value = ...
};

struct Fib<5> {
    value = ...
};

struct Fib<6> {
    value = ...
};

struct Fib<7> {
    value = ...
};
```

This prints 13.
Specialization

- Things start to get strange.

```cpp
template <unsigned N>
struct Fib {
  static constexpr unsigned value =
    Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
  static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
  static constexpr unsigned value = 0;
};

struct Fib<2> { 
  value = ...
};

struct Fib<3> { 
  value = ...
};

struct Fib<4> { 
  value = ...
};

struct Fib<5> { 
  value = ...
};

struct Fib<6> { 
  value = ...
};

struct Fib<7> { 
  value = ...
};
```

This prints 13. Value is computed at compile time!
Specialization

- Things start to get strange.

```
template <unsigned N>
struct Fib {
    static constexpr unsigned value =
    Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
```

This prints 13.

Value is computed at compile time!
template <unsigned N>
struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
Specialization

- Things start to get strange.

```cpp
template <unsigned N>
struct Fib {
    static constexpr unsigned value =
        Fib<N-1>::value + Fib<N-2>::value;
};

template <>
struct Fib<1> {
    static constexpr unsigned value = 1;
};

template <>
struct Fib<0> {
    static constexpr unsigned value = 0;
};
```

This prints 13.
The value is computed at compile time!

```cpp
cout << Fib<7>::value << "\n";
```

`constexpr` functions make this less common.
Things start to get strange.

```cpp
constexpr unsigned fibonacci(unsigned target) {
    if (target < 2) {
        return target;
    }
    unsigned fib_back_2 = 0;
    unsigned fib_back_1 = 1;
    for (unsigned pos = 2; pos <= target; ++pos) {
        unsigned latest = fib_back_2 + fib_back_1;
        fib_back_2 = fib_back_1;
        fib_back_1 = latest;
    }
    return fib_back_1;
}
```

This prints 13.
The value is computed at compile time!

```cpp
constexpr auto result = fibonacci(40);
```

constexpr auto result = fibonacci(40);

constexpr functions make this less common.

```cpp
constexpr unsigned fibonacci(unsigned target) {
    if (target < 2) {
        return target;
    }
    unsigned fib_back_2 = 0;
    unsigned fib_back_1 = 1;
    for (unsigned pos = 2; pos <= target; ++pos) {
        unsigned latest = fib_back_2 + fib_back_1;
        fib_back_2 = fib_back_1;
        fib_back_1 = latest;
    }
    return fib_back_1;
}
```
Specialization

- Specialization can help build efficient, decoupled interfaces through type traits.
Specialization can help build efficient, decoupled interfaces through type traits.

```cpp
template<typename GraphKind>
struct GraphTraits {
    static_assert(false, "Not specialized");
};
```
Specialization can help build efficient, decoupled interfaces through type traits.

```cpp
template<
typename GraphKind>
struct GraphTraits {
    static_assert(false, "Not specialized");
};

template<>
struct GraphTraits<SocialGraph> {
    using NodeRef = ...
    using ChildIterator = ...
    NodeRef getEntryNode(SocialGraph&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};
```
Specialization can help build efficient, decoupled interfaces through *type traits*. Regardless of the actual graph data structure, or even its API, traits allow generic algorithms to work!
Specialization can help build efficient, decoupled interfaces through *type traits*.

Type traits in C++ are deeply related to type classes in Haskell.
- Concepts in the next version of C++ make that clearer & cleaner
Partial Specialization

- Maybe you do not want to *fully specialize* the type
  - A set of types behave similarly but not all
Partial Specialization

- Maybe you do not want to *fully specialize* the type
  - A set of types behave similarly but not all
  - We already saw this with default arguments!
Partial Specialization

- Maybe you do not want to fully specialize the type
  - A set of types behave similarly but not all
  - We already saw this with default arguments!

```cpp
template<class T=std::string, class C=std::vector<T>, auto size=10>
class SmallRoster { ... };

SmallRoster<Kitten> teamKittens;
SmallRoster<> teamStrings;
```
Sometimes information needs to flow from a derived class to a base class.
Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};
```
Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```
Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```
CRTP

• Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```
Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
    public:
        void print() { getDerived().printImpl(); }
    private:
        T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
    public:
        void printImpl() { printf("Yo\n"); }
};
```

What other approaches could we have used? What are the trade offs?
Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```

What other approaches could we have used? What are the trade offs?

Flexibility vs Efficiency
Sometimes information needs to flow from a derived class to a base class.

```
template<class T>
class Base {
    public:
        void print() { getDerived().printImpl(); }
    private:
        T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
    public:
        void printImpl() { printf("Yo\n"); }
};
```

We have already seen a pattern this could make safer. How?
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
  - Invert control so that the user of the algorithm can pass in new policies.
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
  - Invert control so that the user of the algorithm can pass in new policies.

This is essentially dependency injection at the template level!
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
  - Invert control so that the user of the algorithm can pass in new policies.

- Originally, policy based design
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
  - Invert control so that the user of the algorithm can pass in new policies.

- Originally, policy based design
  - focused on ad hoc, implicit interfaces amongst policies
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
  - Invert control so that the user of the algorithm can pass in new policies.

- Originally, policy based design
  - focused on ad hoc, implicit interfaces amongst policies
  - Used multiple inheritance for mixins and flexible policy coordination.
Policy Based Design

- All of these tools we’ve seen led to policy based design in the 2000’s.
  - Identify all of the design decisions in an algorithm & turn them into template parameters.
  - Invert control so that the user of the algorithm can pass in new policies.

- Originally, policy based design
  - focused on ad hoc, implicit interfaces amongst policies
  - Used multiple inheritance for mixins and flexible policy coordination.

- Lately people have wanted more assurances; it can be easy to make an interface too flexible.
What is printed by `foo(42)`?

```cpp
void foo(unsigned i) {
    std::cout << "unsigned " << i << "\n";
}

template <typename T>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}
```
void foo(unsigned i) {
    std::cout << "unsigned " << i << "\n";
}

template <typename T>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}

What is printed by foo(42)?
"template 42"
Why?
void foo(unsigned i) {
    std::cout << "unsigned " << i << "\n";
}

template <typename T>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}

What is printed by foo(42)?
"template 42"

Why?

What we want is a way to bound where our templates apply...
SFINAE & Correctness

- SFINAE is one approach to bounded static polymorphism in C++
SFINAE & Correctness

- SFINAE is one approach to bounded static polymorphism in C++
- **Substitution Failure Is Not An Error**
  - When trying to substitute into the template or function signature, skip errors & keep looking.
SFINAE & Correctness

- SFINAE is one approach to bounded static polymorphism in C++
- **Substitution Failure Is Not An Error**
  - When trying to substitute into the template or function signature, skip errors & keep looking.

```cpp
template <typename T, typename U=T::value_type>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}
```
SFINAE & Correctness

- SFINAE is one approach to bounded static polymorphism in C++
- **Substitution Failure Is Not An Error**
  - When trying to substitute into the template or function signature, skip errors & keep looking.

```cpp
template <typename T, typename U=T::value_type>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}
```

What happens if we try to match an integer?
SFINAE & Correctness

- template<B> enable_if {...};
  - Using the same techniques we've seen, enable_if allows arbitrary condition checking.
SFINAE & Correctness

- template<B> enable_if{...};
  - Using the same techniques we've seen, enable_if allows arbitrary condition checking.

```cpp
template <typename T, typename=std::enable_if_t<std::is_class_v<T>>> void foo(const T& t) {
    std::cout << "template " << t << "\n";
}
```
SFINAE & Correctness

- template<B> enable_if{...};
  - Using the same techniques we've seen, enable_if allows arbitrary condition checking.

```cpp
template <typename T, typename=std::enable_if_t<std::is_class_v<T>>>
void foo(const T& t) {
    std::cout << "template " << t << "\n";
}
```

How would we implement that?
SFINAE & Correctness

- NOTE: Going forward in C++20(+), much of this will be simplified via “Concepts”

```cpp
void foo(Sequence auto& s) {
    ...
}
std::list<int> asLinkedList = ...;
foo(asLinkedList);
std::vector<int> asVector = ...;
foo(asVector);
```
Templates

- Enable efficient generic programming in C++
Templates

- Enable efficient generic programming in C++
- Can be (partially) specialized to refine behavior
Templates

- Enable efficient generic programming in C++
- Can be (partially) specialized to refine behavior
- Can be used in traits similar to Haskell type classes
Templates

- Enable efficient generic programming in C++
- Can be (partially) specialized to refine behavior
- Can be used in traits similar to Haskell type classes
- Can be made safer using SFINAE based bounds