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#, TypeScript, Swift, Python, ...
  – Parameterized types in ML, Haskell, (Python again), ...
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)
template<typename T>
constexpr T PI = T(3.14159265358979323846)
Variable, Type, & Function Templates

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

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

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

    T first;
    T second;
};
```
Variable, Type, & Function Templates

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"}
};
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);
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);

But *something* about this should feel odd!
(Apart from min already existing)
Variable, Type, & Function Templates

- Several different constructs can be templated...
  - Variables
  - Classes
  - Functions
Several different constructs can be templated...  
- Variables  
- Classes  
- Functions  
- Type aliases (`using`)
Variable, Type, & Function Templates

- 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...
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"}
};
```
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
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"}};
```

• Uses the constructor as a guide for deduction.

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);
```

- Can only deduce based on function arguments
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);

- Can only deduce based on function arguments
```
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());
```

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, 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
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
};
```
• 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);
```
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;
```
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);
```

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

What do you think the declaration of `std::array` looks like?
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);
```

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

```
template<typename 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.
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.
Specialization

<functional>

```cpp
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`?
Specialization

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 ...;
        }
    };
}
```
Specialization

\textit{<functional>}

\begin{verbatim}
namespace std {
    template< class Key >
    struct hash;
}
\end{verbatim}

\textit{<Cats.h>}

\begin{verbatim}
namespace std {
    template<>
    struct hash<Cat> {
        std::size_t
        operator()(Cat const& s) const noexcept {
            return ...;
        }
    };
}
\end{verbatim}

\begin{verbatim}
std::unordered_set<Cat> bigBagOfCats;
\end{verbatim}
Specialization

- Things start to get strange.
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;
};
```
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";
```
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!

```
cout << Fib<7>::value << "\n";
```
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<6> { 
    value = ... 
};

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

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

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

Specialization
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;
};

Things start to get strange.

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

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

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

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

struct Fib<4> {
    value = ...};
Things start to get strange.

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

- Things start to get strange.

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. Value is computed at compile time!

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

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

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

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

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

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

Things start to get strange.

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

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!

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

constexpr functions make this less common.
**Specialization**

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

`constexpr` functions make this less common.

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

- 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
cout << Fib<7>::value << "\n";
constexpr auto result = fibonacci(40);
```

constexpr functions make this less common.

Where would you use it?
look up tables, efficient data structures, bare metal, ...
Specialization can help build efficient, decoupled interfaces through type traits.
Specialization

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

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

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

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

template<>
struct GraphTraits<SocialNetwork> {
    using NodeRef = ...
    using ChildIterator = ...
    NodeRef getEntryNode(SocialNetwork&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};
```
Specialization can help build efficient, decoupled interfaces through type traits.

We can define custom types & behavior related to the type parameter.
Specialization

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

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

template<>
struct GraphTraits<SocialNetwork> {
    using NodeRef = ...;
    using ChildIterator = ...;
    NodeRef getEntryNode(SocialNetwork& ...) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};

template<>
struct GraphTraits<RoadMap> {
    using NodeRef = ...;
    using ChildIterator = ...;
    NodeRef getEntryNode(RoadMap& ...) {...}
    ChildIterator child_begin(NodeRef& ...) {...}
    ChildIterator child_end(NodeRef& ...) {...}
};
```
Specialization

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

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

template<>
struct GraphTraits<SocialNetwork> {
    using NodeRef = ...;
    using ChildIterator = ...;
    NodeRef getEntryNode(SocialNetwork&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};

template<>
struct GraphTraits<RoadMap> {
    using NodeRef = ...;
    using ChildIterator = ...;
    NodeRef getEntryNode(RoadMap&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};

template<class Kind, class GT=GraphTraits<Kind>>
void printGraph(const& Kind graph) { ... }

RoadMap roadMap;
printGraph(roadMap);

SocialNetwork socialGraph;
printGraph(socialGraph);
```
Specialization

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

```cpp
template<typename GraphKind>
struct GraphTraits {
    static_assert(false, "Not specialized");
};
template<>
struct GraphTraits<SocialNetwork> {
    using NodeRef = ...;
    using ChildIterator = ...;
    NodeRef getEntryNode(SocialNetwork&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};
template<>
struct GraphTraits<RoadMap> {
    using NodeRef = ...;
    using ChildIterator = ...;
    NodeRef getEntryNode(RoadMap&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};
template<class Kind, class GT=GraphTraits<Kind>>
void printGraph(const& Kind graph) { ... }
```

We can use GT to provide a graph interface to an arbitrary Kind and write the function only once.

```cpp
RoadMap roadMap;
printGraph(roadMap);

SocialNetwork socialGraph;
printGraph(socialGraph);
```
Specialization can help build efficient, decoupled interfaces through *type traits*.

```cpp
template<typename GraphKind>
struct GraphTraits {
    static_assert(false, "Not specialized");
};
template<>
struct GraphTraits<SocialNetwork> {
    using NodeRef = ...
    using ChildIterator = ...
    NodeRef getEntryNode(SocialNetwork&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};
template<>
struct GraphTraits<RoadMap> {
    using NodeRef = ...
    using ChildIterator = ...
    NodeRef getEntryNode(RoadMap&) {...}
    ChildIterator child_begin(NodeRef&) {...}
    ChildIterator child_end(NodeRef&) {...}
};

template<class Kind, class GT=GraphTraits<Kind>>
void
printGraph(const& Kind graph) {
    ...
}
```

```cpp
RoadMap roadMap;
printGraph(roadMap);

SocialNetwork socialGraph;
printGraph(socialGraph);

printGraph<SocialNetwork,CustomView>(socialGraph);
```

And we *can* even customize how the interface is bound if so desired.
Specialization

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

```cpp
template<typename GraphKind>
struct GraphTraits {
    static_assert(false, "Not specialized");
};
template<>
struct GraphTraits<SocialNetwork> {
    using NodeRef = ...;
    using ChildIterator = ...
    NodeRef getEntryNode(SocialNetwork&) {...}
    ChildIterator child_begin(NodeRef&){...}
    ChildIterator child_end(NodeRef&){...}
};
template<>
struct GraphTraits<RoadMap> {
    using NodeRef = ...;
    using ChildIterator = ...
    NodeRef getEntryNode(RoadMap&) {...}
    ChildIterator child_begin(NodeRef&){...}
    ChildIterator child_end(NodeRef&){...}
};

Regardless of the actual graph data structure, or even its API, traits allow generic algorithms to work!
```

Let's see it in action...

```cpp
// Generic algorithm
template<class Kind, class GT=GraphTraits<Kind>>
void printGraph(const& Kind graph) { ... }

... 
RoadMap roadMap;
printGraph(roadMap);

SocialNetwork socialGraph;
printGraph(socialGraph);

printGraph<SocialNetwork,CustomView>(socialGraph);`
Specialization

- Specialization can help build efficient, decoupled interfaces through type traits.
- Type traits in C++ are deeply related to type classes in Haskell.
Specialization

- 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
Specialization

- 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.

How does this relate to *coupling*?
Specialization

- 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.
Specialization

- 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.

SocialNetwork
- printGraph()
- shortestPath()
- findClique()

RoadMap

How does this relate to *coupling*?
Specialization

- 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

SocialNetwork
RoadMap

GraphTraits
  printGraph()
  shortestPath()
  findClique()

How does this relate to *coupling*?
Specialization

- 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.

Information & behavior can be added to data types regardless of original APIs.
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;
```
CRTP

- 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.

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"); }
};
```
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"); }
};
```
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
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.

```cpp
template<class T, class Allocator = std::allocator<T>>
class vector;
```
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.

```cpp
template<class T, class Allocator = std::allocator<T>>
class vector;
```

This addresses the *combinatorial explosion* of hand written types. We shall see this again in design patterns.
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.

```cpp
namespace TF {
  class LeakyReluOp : public Op<LeakyReluOp,
    OpTrait::OneResult,
    OpTrait::HasNoSideEffect,
    OpTrait::SameOperandsAndResultType,
    OpTrait::OneOperand> {

public:
  static StringRef getOperationName() {
    return "tf.LeakyRelu";
  }
  Value* value() { ... }
  APFloat alpha() const { ... }
  static void build(...) { ... }
  bool verify() const {
    if (...) return emitOpError("requires 32-bit float attribute 'alpha'");
    return false;
  }

};
} // end namespace
```

Lattner, MLIR Primer
Compilers for Machine Learning Workshop, CGO 2019
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)`?
SFINAE & Correctness

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 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**
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.
SFNAE & Correctness

- SFNAE 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=typename 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.
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 \n";
}
```

How would we implement that?
This can also be attacked with `if constexpr`:

```cpp
template <typename T>
void foo(const T& t) {
  if constexpr (std::is_class_v<T>) {
    std::cout << "template \n";
  } else if constexpr (std::is_unsigned_v<T>) {
    std::cout << "unsigned " << t << "\n";
  }
}
```

But this may not be exactly the same!
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);
```
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);
```
SFINAE & Correctness

- NOTE: Going forward in C++20(+), much of this will be simplified via “Concepts”
- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages
SFINAE & Correctness

- NOTE: Going forward in C++20(+), much of this will be simplified via “Concepts”
- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{a} } -> std::convertible_to<std::size_t>;
};
```

[cppreference.com]
SFINAE & Correctness

- NOTE: Going forward in C++20(+), much of this will be simplified via “Concepts”
- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{a} } -> std::convertible_to<std::size_t>;
};
```
[cppreference.com]
SFINAE & Correctness

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

- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);
```
SFINE & Correctness

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

- Provide rich predicates and clear error messages, while templates & SFINE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);
```
SFINAE & Correctness

- NOTE: Going forward in C++20(+), much of this will be simplified via “Concepts”
- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{a} } -> std::convertible_to<std::size_t>
};

template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);
```

```cpp
foo("Oh bother."s);
bar("Oh bother."s);
foo(32);
bar(32);
Cat kitten;
bar(kitten);
Dog doggo;
bar(doggo);
```
NOTE: Going forward in C++20(+), much of this will be simplified via “Concepts”

Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{a} } -> std::convertible_to<std::size_t>;
};
```

```cpp
template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);
```

```cpp
foo("Oh bother."s);
bar("Oh bother."s);
foo(32);
bar(32);
foo(32);
bar(kitten);
foo("Oh bother."s);
bar(kitten);
```

```cpp
Cat kitten;
bar(kitten);
Dog doggo;
bar(doggo);
```
SFINAE & Correctness

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

- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);
```

```cpp
foo("Oh bother."s);
bar("Oh bother."s);
foo(32);
bar(32);
foo(kitten);
bar(kitten);
```

```cpp
Dog doggo;
bar(doggo);
```
SFINAE & Correctness

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

Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);
```

```cpp
foo("Oh bother."s);
bar("Oh bother."s);
foo(32);
bar(32);
Cat kitten;
bar(kitten);
Dog doggo;
bar(doggo);
```

<source>: In function 'int main()':
<source>:49:12: error: use of function 'void bar(const auto:11&)
    with unsatisfied constraints
    49 |   bar(doggo);
    | ^
<source>:10:9:   required for the satisfaction of 'Hashable<auto:11>'
<source>:10:20:   in requirements with 'const T& a' [with _Tp = Dog; T = Dog]
<source>:11:21: note: the required expression 'std::hash<_Tp>{}(a)' is invalid
    11 |   { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
```
SFINAE & Correctness

- **NOTE:** Going forward in C++20(+), much of this will be simplified via "Concepts"
- Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
#include <type_traits>

// Template with SFINAE
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

void foo(const Hashable& hashable);
void bar(const Hashable auto& hashable);

// Simple usage examples
foo("Oh bother.");
bar("Oh bother.");
foo(32);
bar(32);
Cat kitten;
bar(kitten);
Dog doggo;
bar(doggo);
```

Error message:
```
The required expression 'std::hash<_Tp>{}(a)' is invalid in requirements with 'const T& a' [with _Tp = Dog; T = Dog]
```
SFINAE & Correctness

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

Provide rich predicates and clear error messages, while templates & SFINAE alone create notorious error messages

```cpp
template<typename T>
concept Hashable = requires(T a) {
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>
};

template<Hashable T>
void foo(const T& hashable);

void bar(const Hashable auto& hashable);

foo("Oh bother."s);
bar("Oh bother."s);
foo(32);
bar(32);

Cat kitten;
bar(kitten);

Dog doggo;
bar(doggo);
```

<source>: In function 'int main()':
<source>:49:12: error: use of function 'void bar(const auto:11&)'
    with unsatisfied constraints
        49 |   bar(doggo);
           | ^
<source>:10:9:   required for the satisfaction of 'Hashable<auto:11>,'
<source>:10:20:   in requirements with 'const T& a' [with _Tp = Dog; T = Dog]
<source>:11:21:   note: the required expression 'std::hash<_Tp>{a)' is invalid
        11 |   { std::hash<T>{a) } -> std::convertible_to<std::size_t>;
```
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 for highly efficient decoupling
Templates

- Enable efficient generic programming in C++
- Can be (partially) specialized to refine behavior
- Can be used in traits for highly efficient decoupling
- Can be made safer using SFINAE and now Concepts based bounds