Fundamentals of Type-Dependent Code Reuse in C++

Mark Isaacson

Roadmap

- Reusing an implementation
- Selecting between implementations
- Opt-in functions

A Beginning

```
assert(max(3, 5) == 5);
assert(max("abc"s, "def"s) == "def"s);
```

Sharing an Implementation

```
template<typename T>
T max(T x, T y) {
  return x < y ? y : x;
}
```

Sharing an Implementation

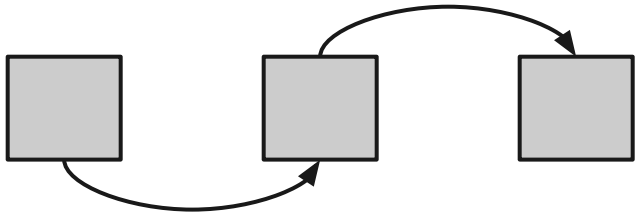
```
template<LessThanComparable T>
T max(T x, T y) {
  return x < y ? y : x;
}
```

getNthElement

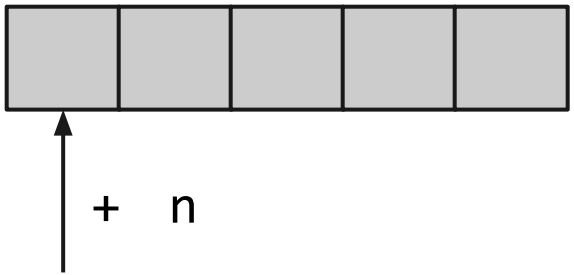
```
vector<int> x { 1, 2, 3, 4, 5 };
list<int> y { 1, 2, 3, 4, 5 };

assert(getNthElement(x, 3) == 4);
assert(getNthElement(y, 3) == 4);
```

Same Interface, Different Implementation

```
int getNthElement(const list<int>& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i)
    ++itr;
  return *itr;
}
```



Same Interface, Different Implementation




```
int getNthElement(const vector<int>& c, size_t n) {
  return c[n];
}
```

Same Interface, Different Implementation

```
int getNthElement(const list<int>& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i)
    ++itr;
  return *itr;
}
int getNthElement(const vector<int>& c, size_t n) {
  return c[n];
}
```

Limitations of Overloading

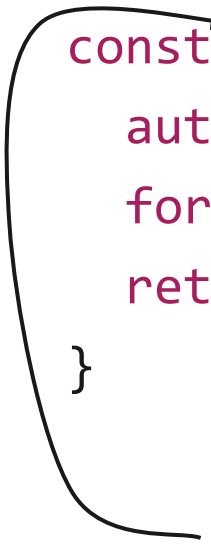
```
vector<int> x { 1, 2, 3, 4, 5 };
list<int> y { 1, 2, 3, 4, 5 };
deque<int> z { 1, 2, 3, 4, 5 };
vector<double> a { 1.3, 2.5 };

assert(getNthElement(x, 3) == 4);
assert(getNthElement(y, 3) == 4);
assert(getNthElement(z, 3) == 4); //Doesn't compile (yet...)
assert(getNthElement(a, 3) == 4); //Doesn't compile (yet...)
```

Accepting Any Container

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i) { ++itr; }
  return *itr;
}
```

Accepting Any Container

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i) { ++itr; }
  return *itr;
}
```



```
typename Container::value_type
```

Conditionally Sharing an Implementation

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i) { ++itr; }
  return *itr;
}
template<typename T>
const T& getNthElement(const vector<T>& c, size_t n) {
  return c[n];
}
```

Still Fishy

```
vector<int> x { 1, 2, 3, 4, 5 };
list<int> y { 1, 2, 3, 4, 5 };
deque<int> z { 1, 2, 3, 4, 5 };
vector<double> a { 1.3, 2.5 };

assert(getNthElement(x, 3) == 4);
assert(getNthElement(y, 3) == 4);
assert(getNthElement(z, 3) == 4);
assert(getNthElement(a, 3) == 4);
```

Quick Recap (before moving on)

- Need to share? Think templates!
- Need to distinguish? Think overloading!
- Need both? Use both.

Level 2

Goal: Make `getNthElement` work for *all* types, *and* make it run in O(1) for types that can do so.

What do we need for O(1)

- `vector`
- `list`
- `deque`
- `set`
- `map`
- `array`
- `...`

A Generalized O(1) Implementation

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto iteratorToNthElement = cbegin(c) + n;
  return *iteratorToNthElement;
}
```

O(N) Implementation Stays the Same

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i) { ++itr; }
  return *itr;
}
```

Overloading is Ambiguous

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i) { ++itr; }
  return *itr;
}
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  return *(cbegin(c) + n);
}
```

Selecting an Implementation

```
struct input_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag
  : public forward_iterator_tag{};
struct random_access_iterator_tag
  : public bidirectional_iterator_tag{};

void foo(forward_iterator_tag x);        //Implementation 1
void foo(random_access_iterator_tag x); //Implementation 2
```

Selecting an Implementation

```
void foo(forward_iterator_tag x);         //Implementation 1
void foo(random_access_iterator_tag x); //Implementation 2

//Which implementation gets called?
foo(forward_iterator_tag{});
foo(bidirectional_iterator_tag{});
foo(random_access_iterator_tag{});
```

Tagged Implementations

```
template<typename Container>
const auto& getNthElementImpl(const Container& c, size_t n,
                              random_access_iterator_tag) {
  auto iteratorToNthElement = cbegin(c) + n;
  return *iteratorToNthElement;
}
```

Tagged Implementations

```
template<typename Container>
const auto& getNthElementImpl(const Container& c, size_t n,
                              forward_iterator_tag) {
  auto itr = cbegin(c);
  for (auto i = 0u; i < n; ++i) {
    ++itr;
  }
  return *itr;
}
```

Selecting an Implementation (Tag Dispatch)

```
template<typename Container>
const auto& getNthElement(const Container& c, size_t n) {
  auto tag = typename iterator_traits<
    typename Container::iterator>::iterator_category{};
  return getNthElementImpl(c, n, tag);
}
```

std::copy_n

```
template<typename T> void copy_n(T from, size_t n, T to) {
  auto tag = typename is_trivially_copyable<T>::type{};
  copy_n_impl(from, n, to, tag);
}
```

std::copy_n

```
template<typename T>
void copy_n_impl(T from, size_t n, T to, true_type) {
  memmove(to, from, n);
}

template<typename T>
void copy_n_impl(T from, size_t n, T to, false_type) {
  for (auto i = 0u; i < n; ++i) *to++ = *from++;
}
```

Performance Implications

```
template<typename T>
void copy_n_impl(T from, size_t n, T to, true_type) {
  for (auto i = 0u; i < n; ++i) *to++ = *from++;
}

template<typename T>
void copy_n_impl(T from, size_t n, T to, false_type) {
  for (auto i = 0u; i < n; ++i) *to++ = *from++;
}
```

High Level Idea

We generalized overloading. We can select an implementation based on an *arbitrary* condition.

See the `<type_traits>` header for more ways to distinguish types (e.g. `is_polymorphic`, `is_floating_point`, `is_same`, etc).

Value?

- We abstracted an idea: "get the Nth element"
- It *works* with all possible types
- It uses the *fastest* possible algorithm for *all* types
- Users are totally *insulated* from design details

Value?

```
template<typename Container>
void printNthElement(const Container& c, size_t n) {
  cout << getNthElement(c, n) << endl;
}
```

Oops

```
template<typename Container>
auto getNthElement(const Container& c, size_t n) {
  auto iteratorToNthElement = advance(cbegin(c), n);
  return *iteratorToNthElement;
}
```

Oops

```
template<typename Container>
auto getNthElement(const Container& c, size_t n) {
  return *next(cbegin(c), n);
}
```

Opt-in Functions

Caller Opt-in

```
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
}
void foo() {
  Widget x { 1 };
  Widget y { 2 };
  assert(x != y); //Goal: Make this compile
}
```

Caller Opt-in

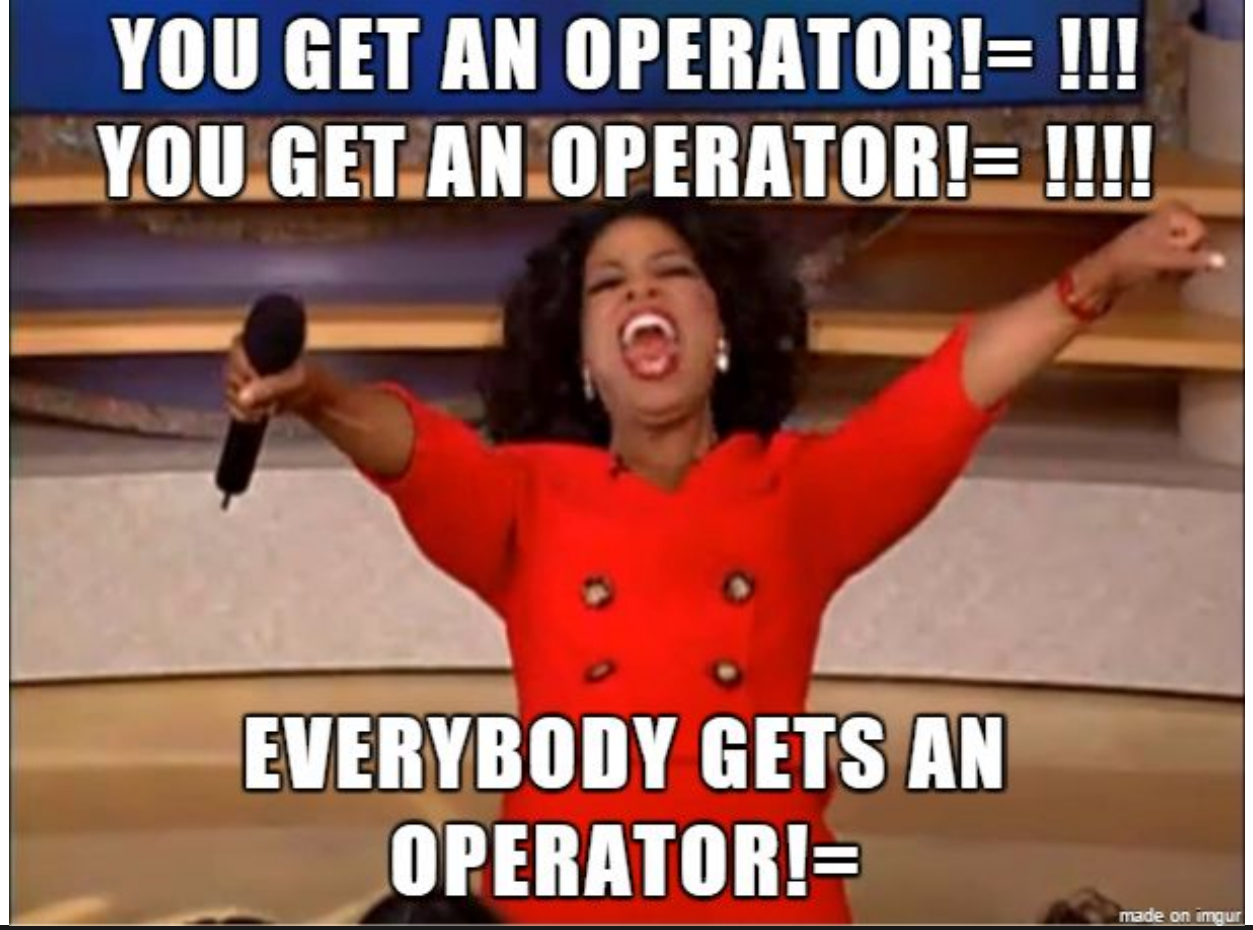
```
namespace notEqual {
  template<typename T>
  bool operator!=(const T& lhs, const T& rhs) {
    return !(lhs == rhs);
  }
}
```

Caller Opt-in

```
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
}
void foo() {
  using namespace notEqual;
  Widget x { 1 };
  Widget y { 2 };
  assert(x != y);
}
```

Caller Opt-in

```
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
}
void foo() {
  using namespace notEqual;
  Widget x { 1 };
  Widget y { 2 };
  assert(x != y);
}
```



(When the function is in scope)

Designer Opt-in

```
struct NotEqualMixin {};
struct Gadget : public NotEqualMixin {
  bool operator==(const Gadget& rhs) const {return x == rhs.x;}
  int x;
};
void foo() {
  Gadget x { 1 };
  Gadget y { 2 };
  assert(x != y); //Goal: Make this compile
}
```

Designer Opt-in

```
template<typename Derived> struct NotEqualMixin {};
struct Gadget : public NotEqualMixin<Gadget> {
  bool operator==(const Gadget& rhs) const {return x == rhs.x;}
  int x;
};
void foo() {
  Gadget x { 1 };
  Gadget y { 2 };
  assert(x != y); //Goal: Make this compile
}
```

Designer Opt-in

```
template<typename Derived>
bool operator!=(const NotEqualMixin<Derived>& lhs,
                const NotEqualMixin<Derived>& rhs) {
  static_assert(
    is_base_of<NotEqualMixin<Derived>, Derived>::value,
    "Detected misuse of NotEqualMixin");
  const auto& derivedLhs = static_cast<const Derived&>(lhs);
  const auto& derivedRhs = static_cast<const Derived&>(rhs);
  return !(derivedLhs == derivedRhs);
}
```

Instance by Instance Opt-in

```
void foo() {
  Gadget x { 1 };
  Gadget y { 2 };
  assert(x != y); //Goal: Does *not* compile
  GadgetWithNotEqual a { 1 };
  GadgetWithNotEqual b { 2 };
  assert(a != b); //Goal: *Does* compile
}
//Overall goal: No duplicated implementation code.
```

Instance by Instance Opt-in

```
template<template<typename> class... Mixins>
struct GadgetImpl
    : public Mixins<GadgetImpl<Mixins...>>... {
  bool operator==(const GadgetImpl& rhs) const;
  int x;
};

using Gadget = GadgetImpl<>;
using GadgetWithNotEqual = GadgetImpl<NotEqualMixin>;
```

Doing templates by hand

```
template<template<typename> class... Mixins>
struct GadgetImpl
    : public Mixins<GadgetImpl<Mixins...>>... {
  bool operator==(const GadgetImpl& rhs) const;
  int x;
};
template<> struct GadgetImpl {
  ...
```

Doing templates by hand

```
template<template<typename> class... Mixins>
struct GadgetImpl
    : public Mixins<GadgetImpl<Mixins...>>... {
  bool operator==(const GadgetImpl& rhs) const;
  int x;
};
template<NotEqualMixin> struct GadgetImpl
    : public NotEqualMixin<GadgetImpl<NotEqualMixin>> {
  ...
```

Instance by Instance Opt-in

```
template<template<typename> class... Mixins>
struct GadgetImpl
    : public Mixins<GadgetImpl<Mixins...>>... {
  bool operator==(const GadgetImpl& rhs) const;
  int x;
};

using Gadget = GadgetImpl<>;
using GadgetWithNotEqual = GadgetImpl<NotEqualMixin>;
```

Non-intrusive
Instance by Instance Opt-in (C++20)

```
void foo() {
  Widget x { 1 };
  Widget y { 2 };
  assert(x != y); //Does *not* compile
  Mixer<Widget, NotEqualMixin> a { 1 };
  Mixer<Widget, NotEqualMixin> b { 2 };
  assert(x != y); //*Does* compile
}
```

Non-intrusive Instance by Instance Opt-in (C++20)

```
template<typename T, template Mixin>
struct Mixer : public Mixin<Mixer<T, Mixin>> {
  T& operator.() { return value; }
private:
  T value;
};
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
};
```

Operator Dot

```
template<typename T>
struct Mixer {
  T& operator.() { return value; }
private:
  T value;
};
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
};
```

```
Mixer<Widget> m;
m.x = 5;
```

Non-intrusive Instance by Instance Opt-in (C++20)

```
template<typename T, template Mixin>
struct Mixer : public Mixin<Mixer<T, Mixin>> {
  T& operator.() { return value; }
private:
  T value;
};
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
};
```

Non-intrusive Instance by Instance Opt-in (C++20)

```
template<typename T, template<typename> class... Mixins>
struct Mixer : public Mixins<Mixer<T, Mixins...>>... {
  T& operator.() { return value; }
private:
  T value;
};
struct Widget {
  bool operator==(const Widget& rhs) const {return x == rhs.x;}
  int x;
};
```

Non-intrusive
Instance by Instance Opt-in (C++20)

```
void foo() {
  Widget x { 1 };
  Widget y { 2 };
  assert(x != y); //Does *not* compile
  Mixer<Widget, NotEqualMixin> a { 1 };
  Mixer<Widget, NotEqualMixin> b { 2 };
  assert(x != y); //*Does* compile
}
```

Credits

- cppreference.com
- Armen Tsirunyan on Stack Overflow

Summary

- Share implementations with templates
- Select implementations with overloading
- User opt-in with namespaces
- Designer opt-in with inheritance

Thanks :)

markisaa@fb.com

http://www.modernmaintainablecode.com