Don’t Get Overloaded by Overload Sets

Roth Michaels

2025

Roth Michaels
Principal Software Engineer
Native Instruments

NATIVE INSTRUMENTS

iZOTOPE

Plugin Alliance

BRAINWORX

01:00:39:02.62
20 3 1 214
44,1
Varispeed
±0.00%
120,0000
Tempo beibehalten
4/4
/16
Kein In
Kein Out
Bearbeiten
Funktionen
Ansicht
Einrasten:
Intelligent
Verschieben:
X-Fade
Norweg...d_audio
Inst 1
Vögel
Wind_Audio
Dub chords
Inst 2
Inst 3
Inst 4
Dub ch...alftime
Molek Melodies
Bongos...Merged
Bongos plus 100
Low Ris...ls pings
drums
Big 808
808 Bass
mm wave weaver
norwegi...S JAZZ
norwegi...o spirit
pads
Norwegian Wood_audio.1
Norwegian Wood_audio.2
Dub chords.2
Dub chords_halftime.5
Dub chords_halftime.2
Molek Melodies.5
Bongos plus 100.2
Low Risk Pharmaceuticals pings.13
norwegian woods mm wave weaver.1
norwegian woods mm RHODES JAZZ
norwegian woods mm atmo spirit.1
norwegian woods mm atmo spirit.5
norwegian woods mm pads.1
norwegian woods mm pads.3
Inst 3
Manuell
Vergleichen
Widerrufen
Wiederholen
Ansicht:
Editor
7 SKIES, Magnificence - THE DRILL
FILTER
CUTOFF
RESO
ENV AMT
AMP
LAYER 1
LAYER 2
THE DRILL
FLT ENV
ATTACK
DECAY
SUSTAIN
RELEASE
AMP ENV
Inst 4
gr808
808 Center
808 Sides
Mode
[mono]
Panning
[attack: fast]
[releaseeeeeee]
Reverb: [Cloud Grain]
Size
Amount:
Rev Lo-Cut:
Rev Hi-Cut:
20 20K
Presets: 1 2 3
Clay and Kelsy
Bouncy FX: [off]
Speed ModW:
gr808
kick
Kontakt
Inst 2
MS20 Drums
Cutoff
Saturation
Reverb
Delay
Delay Time
Peter Flint:
MS20 Drums
Kontakt

You can do it!

#include <c++>

https://www.includecpp.org

Don’t get overloaded by overload sets!

C++ on Sea 2025

What is an overload set?

I don’t think this is C anymore Toto...

C Functions

```
double          log(double);

float           logf(float);
long double     logl(long double);
```

C Functions

```
double          log(double);

float           logf(float);
long double     logl(long double);
```

C++ Overloaded Functions

```
float       std::log(float);

double      std::log(double);

long double std::log(long double);

float       std::logf(float);

long double std::logl(long double);

template <class Integer>
double      std::log(Integer);
```

C++ Overloaded Functions

```
float       std::log(float);

double      std::log(double);

long double std::log(long double);

float       std::logf(float);

long double std::logl(long double);

template <class Integer>
double      std::log(Integer);
```

C++ Overloaded Functions

```
float       std::log(float);

double      std::log(double);

long double std::log(long double);

float       std::logf(float);

long double std::logl(long double);

template <class Integer>
double      std::log(Integer);
```

Overloaded arguments (not return types!)

```
std::string to_string(float);

std::string to_string(double);

std::string to_string(long double);




template <class Integer>
std::string to_string(Integer);
```

Overloaded arguments (not return types!)

```
float       from_string(std::string);

double      from_string(std::string);

long double from_string(std::string);




template <class Integer>
Integer     from_string(std::string);
```

Overloaded arguments (not return types!)

```
float       from_string(std::string);

double      from_string(std::string);

long double from_string(std::string);

error: functions that differ only in their return
type cannot be overloaded

template <class Integer>
Integer     from_string(std::string);
```

Overloaded arguments (not return types!)

```
float       from_string(std::string);

double      from_string(std::string);

long double from_string(std::string);

error: functions that differ only in their return
type cannot be overloaded

template <class Integer>
Integer     from_string(std::string);
```

Constructor overloading

```
// empty vector
std::vector<int> v1;

// vector with 5 elements initialized to 0
std::vector<int> v2(5);

// vector with specified elements
std::vector<int> v3{10, 20, 30, 40, 50};
```

Constructor overloading

```
// empty vector
std::vector<int> v1;

// vector with 5 elements initialized to 0
std::vector<int> v2(5);

// vector with specified elements
std::vector<int> v3{10, 20, 30, 40, 50};
```

Constructor overloading

```cpp
// empty vector
std::vector<int> v1{};

// vector with 5 elements initialized to 0
std::vector<int> v2(5);

// vector with specified elements
std::vector<int> v3{10, 20, 30, 40, 50};
```

```cpp
template <class T, class Allocator = std::allocator<T>> class vector {
public:
      // Default constructor
      constexpr vector() noexcept(noexcept(Allocator()));

      constexpr explicit vector(const Allocator&) noexcept;

      // Fill constructor
      constexpr explicit vector(std::size_t count,
                                const T& value = T(),
                                const Allocator& = Allocator());

      // Range constructor
      template <class InputIt>
      constexpr vector(InputIt first, InputIt last, const Allocator& = Allocator());

      // Copy constructors
      constexpr vector(const vector& other);
      constexpr vector(const vector& other, const Allocator&);

      // Move constructors
      constexpr vector(vector&& other) noexcept;
      constexpr vector(vector&& other, const Allocator&);

      // Initializer list constructor
      constexpr vector(std::initializer_list<T> init, const Allocator& = Allocator());
};
```

Constructor overloading

```cpp
// empty vector
std::vector<int> v1{};

// vector with 5 elements initialized to 0
std::vector<int> v2(5);

// vector with specified elements
std::vector<int> v3{10, 20, 30, 40, 50};
```

```cpp
template <class T, class Allocator = std::allocator<T>> class vector {
public:
      // Default constructor
      constexpr vector() noexcept(noexcept(Allocator()));

      constexpr explicit vector(const Allocator&) noexcept;

      // Fill constructor
      constexpr explicit vector(std::size_t count,
                                const T& value = T(),
                                const Allocator& = Allocator());

      // Range constructor
      template <class InputIt>
      constexpr vector(InputIt first, InputIt last, const Allocator& = Allocator());

      // Copy constructors
      constexpr vector(const vector& other);
      constexpr vector(const vector& other, const Allocator&);

      // Move constructors
      constexpr vector(vector&& other) noexcept;
      constexpr vector(vector&& other, const Allocator&);

      // Initializer list constructor
      constexpr vector(std::initializer_list<T> init, const Allocator& = Allocator());
};
```

```cpp
template <class T, class Allocator = std::allocator<T>> class vector {
public:
      // Default constructor
      constexpr vector() noexcept(noexcept(Allocator()));

      constexpr explicit vector(const Allocator&) noexcept;

      // Fill constructor
      constexpr explicit vector(std::size_t count,
                                const T& value = T(),
                                const Allocator& = Allocator());

      // Range constructor
      template <class InputIt>
      constexpr vector(InputIt first, InputIt last, const Allocator& = Allocator());

      // Copy constructors
      constexpr vector(const vector& other);
      constexpr vector(const vector& other, const Allocator&);

      // Move constructors
      constexpr vector(vector&& other) noexcept;
      constexpr vector(vector&& other, const Allocator&);

      // Initializer list constructor
      constexpr vector(std::initializer_list<T> init, const Allocator& = Allocator());
};
```

What is an overload set?

What do you mean “set”?

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);

    void bar(int x) { foo(x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);          // option 1?

  namespace B {
    void foo(char);       // option 2?
    void foo(long);       // option 3?

    void bar(int x) { foo(x); }
  }
}
```

```
error: call to 'foo' is ambiguous
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);          // option 1?

  namespace B {
    void foo(char);       // option 2?
    void foo(long);       // option 3?

    void bar(int x) { foo(x); }
  }
}

error: call to 'foo' is ambiguous
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);    // option 3

    void bar(int x) { foo((long)x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);    // option 2
    void foo(long);

    void bar(int x) { foo((char)x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);          // option 1?

  namespace B {
    void foo(char);       // option 2?
    void foo(long);       // option 3?
  }
  namespace C {
    void bar(int x) { foo(x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);          // option 1

  namespace B {
    void foo(char);
    void foo(long);
  }
  namespace C {
    void bar(int x) { foo(x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);
  }
  namespace C {
    void bar(int x) { foo(x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);
  }
  namespace C {
    void bar(int x) { A::B::foo(x); }
  }
}
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);
  }
}

const auto& foo_t = typeid(A::foo);
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);
  }
}

const auto& foo_t = typeid(A::B::foo);

error: reference to overloaded function could not
be resolved; did you mean to call it?
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);
  }
}

using foo_t = decltype(A::B::foo);

error: reference to overloaded function could not
be resolved; did you mean to call it?
```

A name representing a set of functions found through name lookup

```
namespace A {
  void foo(int);

  namespace B {
    void foo(char);
    void foo(long);
  }
}

using foo_t = decltype((void(*)(char))A::B::foo);
```

How do we find an overload set?

Qualified or unqualified name lookup...

Qualified name-lookup algorithm of overload sets

If namespace scope:

1. Search current scope for name
2. Search parent scopes until name is found
3. When a name is found all overloads in that scope are included in overload set

If class scope (C::foo(42) or **`c.foo(42)`**):

only search classes and basses

Unqualified name-lookup algorithm of overload sets

1. Search current scope for name
2. Search parent scopes until name is found
3. If not found Argument dependent lookup
    a. if arg class member, class namespace, bass class namespaces, and parent namespaces
    b. If namespace member, namespace and parent namespaces
    c. Built-in: no ADL

ADL

```
namespace other_namespace {
    struct S {};
    void foo(S);
}
namespace A {
   void foo(int);

   namespace B {
      void foo(char);
      void foo(long);
      void bar() { foo(other_namespace::S{}); }
   }
}
```

Why ADL? Operator overloading

```
namespace N {
    struct Foo { /* ... */ };
    Foo operator+(const Foo&, const Foo&);
}
auto f = N::Foo{40} + N::Foo{2};
```

Why ADL? Generic programing

```
template <class T>
auto get_first(const T& x) {
   return get<0>(x);
}

template <>
auto get_first(const std::pair<int, int>& x) {
   return get<0>(x);
}
template <>
auto get_first(const boost::pair<int, int>& x) {
   return get<0>(x);
}
```

How does overload resolution work?

Finding the best match....

Selecting the “best” overload

1. Exact match
   - No conversions needed
   - Includes minor differences like const/volatile qualifiers

2. Promotion
   - char, short, unsigned short, bool → int
   - float → double

3. Standard conversion
   - int → double, float, long
   - pointer conversions (derived* → base*)
   - etc.

4. User-defined conversion
   - Conversions defined by conversion operators or constructors

5. Ellipsis (...) match
   - Last resort, no type checking
   - Avoid if possible in modern C++

Overload resolution

```
void select_func(int x);
void select_func(short x);
void select_func(double x);
void select_func(long x);
void select_func(std::string_view);
void select_func(...);
```

Overload resolution: Exact match

```cpp
void select_func(int x);
void select_func(short x);
void select_func(double x);
void select_func(long x);
void select_func(std::string_view x);
void select_func(...);

int i = 42;
select_func(i);  // Calls select_func(int)

short s = 42;
select_func(s);  // Calls select_func(short)
```

Overload resolution: Promotion

```cpp
void select_func(int x);
void select_func(short x);
void select_func(double x);
void select_func(long x);
void select_func(std::string_view);
void select_func(...);

char c = 'A';     // char is promoted to int
select_func(c);   // Calls select_func(int)

bool b = true;    // bool is promoted to int
select_func(b);   // Calls select_func(int)
```

Overload resolution: Standard conversions

```
void select_func(int x);
void select_func(short x);
void select_func(double x);
void select_func(long x);
void select_func(std::string_view);
void select_func(...);

select_func(42.0);  // Calls select_func(double)

float f = 3.14f;
select_func(f);     // Calls select_func(double)
```

Overload resolution: User-defined conversion

```
void select_func(int x);
void select_func(short x);
void select_func(double x);
void select_func(long x);
void select_func(std::string_view);
void select_func(...);

// Calls select_func(std::string_view)
// const char* const -> std::string_view
select_fun(“hello, world”)
```

Overload resolution: Ellipsis

```
void select_func(int x);
void select_func(short x);
void select_func(double x);
void select_func(long x);
void select_func(std::string_view);
void select_func(...);

struct MysteryType {};

auto x = MysteryType{};
select_func(x) // Calls select_func(...)
```

Selecting the “best” overload

1. Exact match
2. Promotion
3. Standard conversion
4. User-defined conversion
5. Ellipsis (...) match

Selecting the “best” overload

-1. In classes, {} prioritizes default constructor

0. {} prioritizes std::initializer_list

1. Exact match

2. Promotion

3. Standard conversion

4. User-defined conversion

5. Ellipsis (...) match

Constructor overloading

```
// empty vector
std::vector<int> v1{};

// vector with 5 elements initialized to 0
std::vector<int> v2(5);

// vector with specified elements
std::vector<int> v3{10, 20, 30, 40, 50};
```

How does overload resolution work?

What about overloading templates?

Function template overloading

```
template <class T>
T sqrt(T);

template <class T>
std::complex<T> sqrt(std::complex<T>);

float sqrt(float);

void do_sqrts(std::complex<float> z) {
  sqrt(42);
  sqrt(3.14f);
  sqrt(z);
}
```

Function template overloading

1. Find all template specializations
2. Only consider “most specialized”
3. Perform overload resolution for template specializations and functions
4. function > specialization
5. otherwise error

Templates can be surprising!

```cpp
class MaybeInt {
public:
    MaybeInt() {};

    explicit MaybeInt(int x) : m_value{x}, m_valid{true} {}

    MaybeInt(const MaybeInt&) = default;
    MaybeInt(MaybeInt&&) = default;
    MaybeInt& operator=(const MaybeInt&) = default;
    MaybeInt& operator=(MaybeInt&&) = default;

    explicit operator bool() const { return m_valid; }

    operator int() const {  return m_value; }

    operator std::optional<int>() const {
        return m_valid ? m_value : std::optional<int>{};
    }

private:
    int m_value{};
    bool m_valid{false};
};
```

```
class MaybeInt {
public:
    MaybeInt() {};

    explicit operator bool() const { return m_valid; }

    operator int() const {  return m_value; }

    operator std::optional<int>() const {
        return m_valid ? m_value : std::optional<int>{};
    }

private:
    int m_value{};
    bool m_valid{false};
};

MaybeInt no_value{};
assert(!no_value);

std::optional<int> maybe_has_value{no_value};
assert(!maybe_has_value);
```

```cpp
class MaybeInt {
public:
    MaybeInt() {};

    explicit operator bool() const { return m_valid; }

    operator int() const {  return m_value; }

    operator std::optional<int>() const {
        return m_valid ? m_value : std::optional<int>{};
    }

private:
    int m_value{};
    bool m_valid{false};
};

MaybeInt no_value{};
assert(!no_value);

std::optional<int> maybe_has_value{no_value};
assert(!maybe_has_value);  // bool(maybe_has_value) == true
```

Don’t be surprised by templates in overload sets

```
template<class U>
optional(const optional<U>&);

template<class U = std::remove_cv_t<T>>
constexpr optional(U&& value);
```

Don’t be surprised by templates in overload sets

```cpp
template<class U>
optional(const optional<U>&);

template<class U = std::remove_cv_t<T>>
constexpr optional(U&& value);
```

```cpp
class MaybeInt {
public:
    MaybeInt() {};

    explicit operator bool() const { return m_valid; }

    operator int() const {  return m_value; }

    operator std::optional<int>() const {
        return m_valid ? m_value : std::optional<int>{};
    }

private:
    int m_value{};
    bool m_valid{false};
};

MaybeInt no_value{};
assert(!no_value);

std::optional<int> maybe_has_value{no_value};
assert(!maybe_has_value);  // bool(maybe_has_value) == true
```

Don’t be surprised by templates in overload sets

```
template<class U>
optional(const optional<U>&);

template<class U = std::remove_cv_t<T>>
constexpr optional(U&& value);
```

Don’t be surprised by templates in overload sets

```
template<class U>
optional(const optional<U>&);

template<class U = std::remove_cv_t<T>>
constexpr optional(U&& value);
```

```cpp
class MaybeInt {
public:
    MaybeInt() {};

    explicit operator bool() const { return m_valid; }

    operator int() const {  return m_value; }

    operator std::optional<int>() const {
        return m_valid ? m_value : std::optional<int>{};
    }

private:
    int m_value{};
    bool m_valid{false};
};

MaybeInt no_value{};
assert(!no_value);

std::optional<int> maybe_has_value{no_value};
assert(!maybe_has_value);  // bool(maybe_has_value) == true
```

Don’t be surprised by templates in overload sets

```
void foo(std::uint64_t); // option 1?


template <class T>
void foo(T);             // option 2?


const int x = 42;
foo(x);
```

Don't be surprised by templates in overload sets

```
void foo(std::uint64_t);


template <class T>
void foo(T);                    // option 2


const int x = 42;
foo(x);
```

What about member function overload sets?

...classes are namespaces

```cpp
struct B1 {
    void foo(int);    // option 1?
    void foo(double); // option 2?
};



B1{}.foo(42);
```

```cpp
struct B1 {
    void foo(int);     // option 1
    void foo(double);
};



B1{}.foo(42);
```

Watch out for hiding!

```
struct B1 {
    void foo(int);     // option 1?
};
struct D : B1 {
    void foo(double); // option 2?
};

D{}.foo(42);
```

Watch out for hiding!

```
struct B1 {
    void foo(int);
};
struct D : B1 {
    void foo(double); // option 2
};

D{}.foo(42);
```

Watch out for multiple inheritance

```
struct S1 {};  struct S2 {};

struct B1 {
    void foo(S1); // option 1?
};
struct B2 {
    void foo(S2); // option 2?
};
struct D : B1, B2 {};

D{}.foo(S1{});
error: member 'foo' found in multiple base
classes of different types
```

C.138: Create an overload set for a derived class and its bases with using

```
struct S1 {};  struct S2 {};

struct B1 {
    void foo(S1); // option 1
};
struct B2 {
    void foo(S2);
};
struct D : B1, B2 {
using B1::foo;
using B2::foo;
};
D{}.foo(S1{});
```

What about member function overload sets?

...qualified member functions

const qualified member functions

```
class DataHolder {
public:
    std::vector<std::byte> accessData() {
        return m_data;
    }
    const std::vector<std::byte>& accessData() const {
        return m_data;
    }
private:
    std::vector<std::byte> m_data;
};
```

const qualified member functions

```
class DataHolder {
public:
    std::vector<std::byte>& accessData() {
        return m_data;
    }
    const std::vector<std::byte>& accessData() const {
        return m_data;
    }
private:
    std::vector<std::byte> m_data;
};
```

ref-qualified member functions

```
class DataHolder {
public:
    std::vector<std::byte>& accessData() & {
        return m_data;
    }
    const std::vector<std::byte>& accessData() const& {
        return m_data;
    }
    std::vector<std::byte>&& accessData() && {
        return std::move(m_data);
    }
private:
    std::vector<std::byte> m_data;
};
```

What about **`volatile`**-qualified?

...forget about it...

Tips!

Constraining overload sets

```
struct S1 {};

struct S2 {};

struct MyComponent {

    static void foo(S1);

    static void foo(S2);

};
```

F.51: Where there is a choice, prefer default arguments over overloading

```
void print(const string& s, format f = {});
```

```
//! use default format
void print(const string& s);
void print(const string& s, format f);
```

Constraining overload sets

```
inline constexpr my_printer =
    [](std::string_view text) {
        std::print(“{}!!!\n” text”);
    };
```

C.163: Overload only for operations that are roughly equivalent

```
//! remove obstacle from garage exit lane
void open_gate(Gate& g);

//! open file
void fopen(const char* name, const char* mode);
```

C.163: Overload only for operations that are roughly equivalent

```
//! remove obstacle from garage exit lane
void open(Gate& g);

//! open file
void open(const char* name, const char* mode ="r");
```

Users will expect the same behavior regardless of type

Use tagged dispatch for related but different behavior

```
template<class InputIt, class T>
InputIt find(InputIt first, InputIt last,
             const T& value);

template<class ExecutionPolicy, class ForwardIt,
class T>
ForwardIt find(ExecutionPolicy&& policy,
               ForwardIt first, ForwardIt last,
               const T& value);
```

Use tagged dispatch for related but different behavior

```
inline constexpr
std::execution::sequenced_policy seq { /* unspecified */ };

inline constexpr
std::execution::parallel_policy par { /* unspecified */ };

inline constexpr
std::execution::parallel_unsequenced_policy par_unseq {
    /* unspecified */
};

inline constexpr
std::execution::unsequenced_policy unseq { /* unspecified */ };
```

Watch out for hiding virtual methods! — try to limit overloads to bass class

```
struct B1 {
    virtual void foo(int);      // option 1?
};
struct D : B1 {
    void foo(double);           // option 2?
};


D{}.foo(42);
```

Watch out for hiding virtual methods! — try to limit overloads to bass class

```
struct B1 {
    virtual void foo(int);
};
struct D : B1 {
    void foo(double);             // option 2
};


D{}.foo(42);
```

Watch out for hiding virtual methods! — try to limit overloads to bass class

```
struct B1 {
    virtual void foo(int);     // option 1?
};
struct D : B1 {
    void foo(double);          // option 2?
};

auto d = D{};
B1* b = &d;
b->foo(42);
```

Watch out for hiding virtual methods! — try to limit overloads to bass class

```
struct B1 {
    virtual void foo(int);     // option 1
};
struct D : B1 {
    void foo(double);
};


auto d = D{};
B1* b = &d;
b->foo(42);
```

Watch out for hiding virtual methods! — try to limit overloads to bass class

```
struct B1 {
    virtual void foo(int);    // option 1?
};
struct D : B1 {
    void foo(double);         // option 2?
    void foo(int) override;   // option 3?
};

auto d = D{};
B1* b = &d;
b->foo(42);
```

Watch out for hiding virtual methods! — try to limit overloads to bass class

```
struct B1 {
    virtual void foo(int);
};
struct D : B1 {
    void foo(double);
    void foo(int) override;    // option 3
};

auto d = D{};
B1* b = &d;
b->foo(42);
```

Overloading with value categories

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

In Search of the Perfect Return

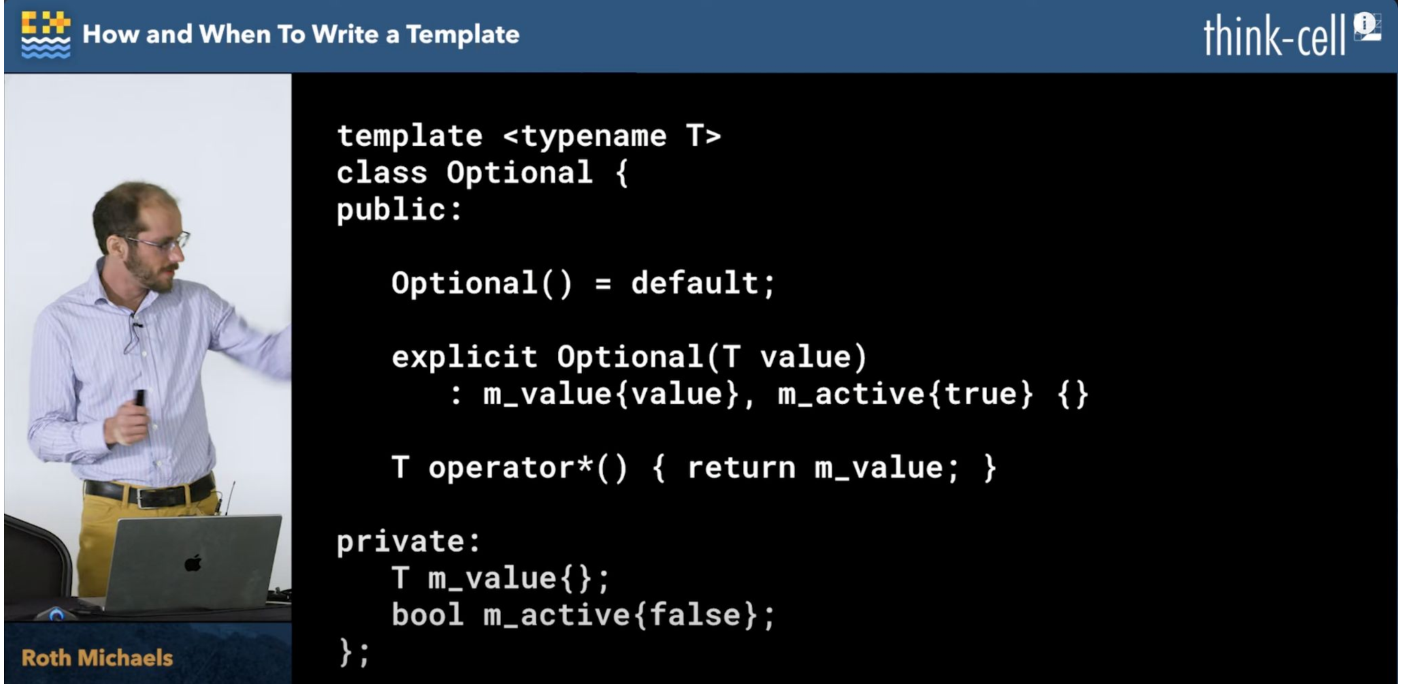
A C++ Programming Journey

Walter E Brown

2025

Function templates are recipes for function overload sets

...and are easier if you use concepts.



https://www.youtube.com/watch?v=wmIY5oNB4Dc

How and When To Write a C++ Template

Thank you!

Roth Michaels
Principal Software Engineer
roth.michaels@native-instruments.com
@thevibesman