Effective APIs

Thamara Andrade | Principal Software Engineer @ Cadence Design Systems

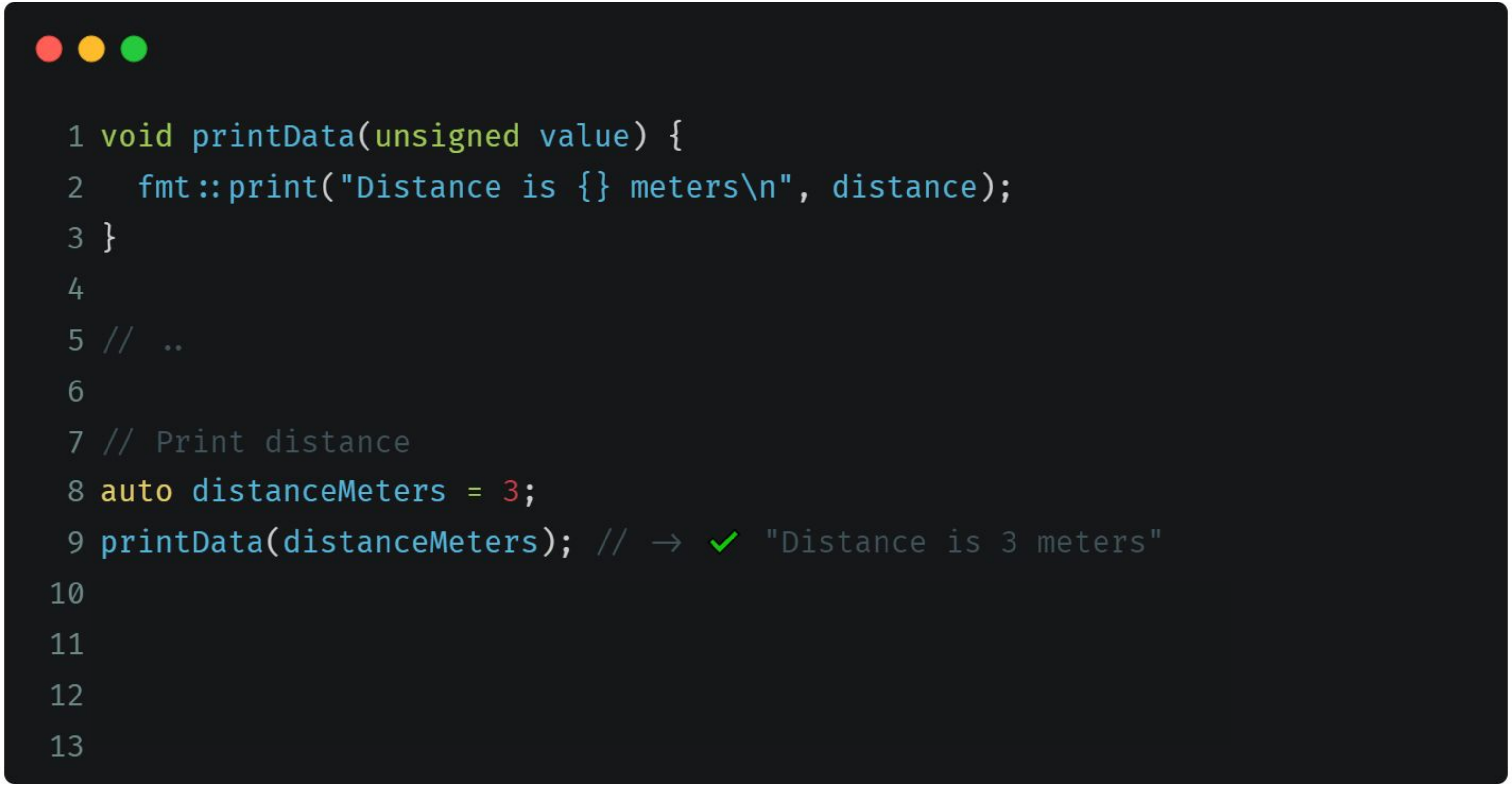
“Make interfaces easy to use correctly and hard to use incorrectly.”

Scott Meyers

I failed Scott Meyers.

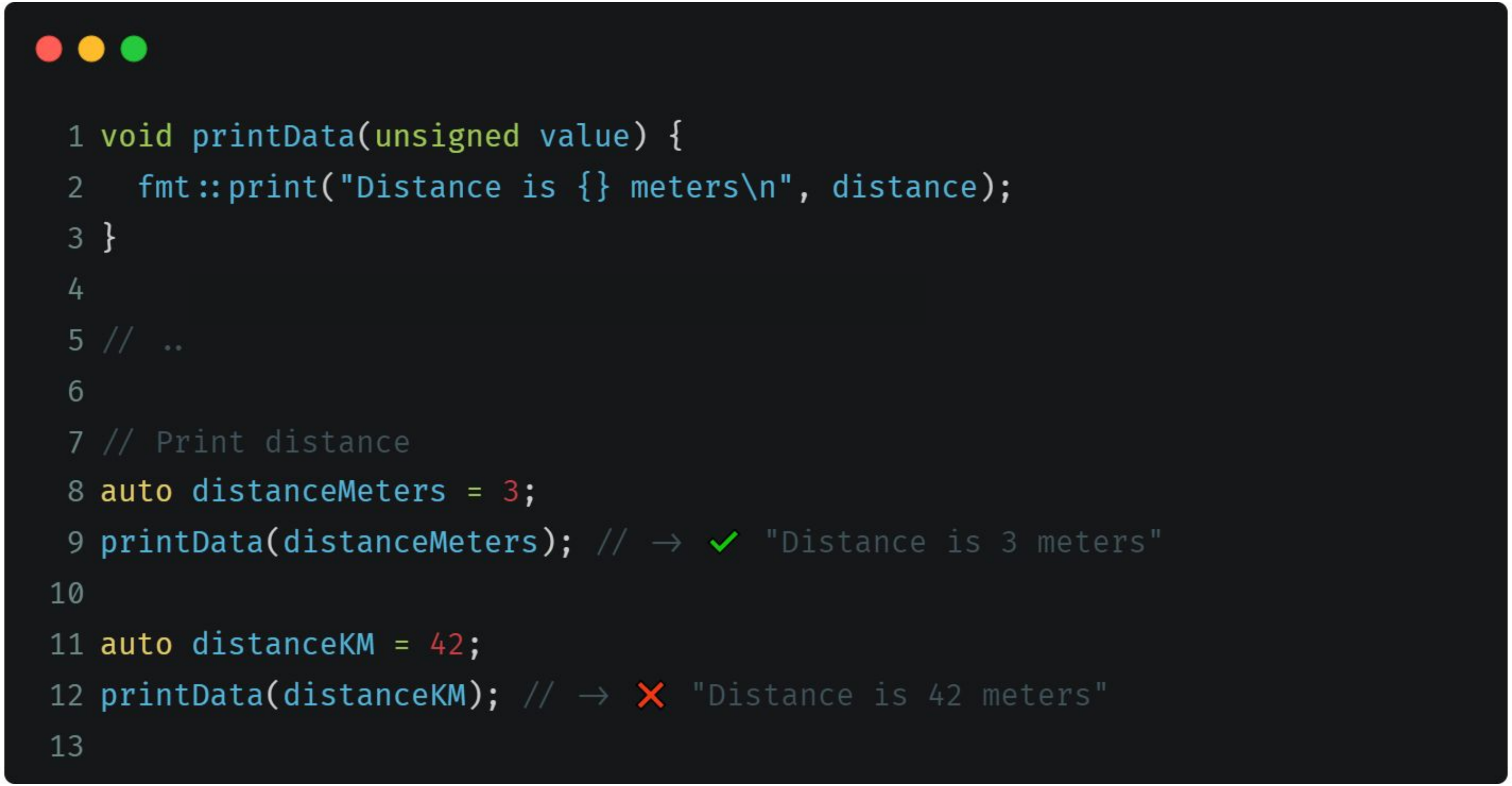
// 1. Better naming

// 1. Better naming



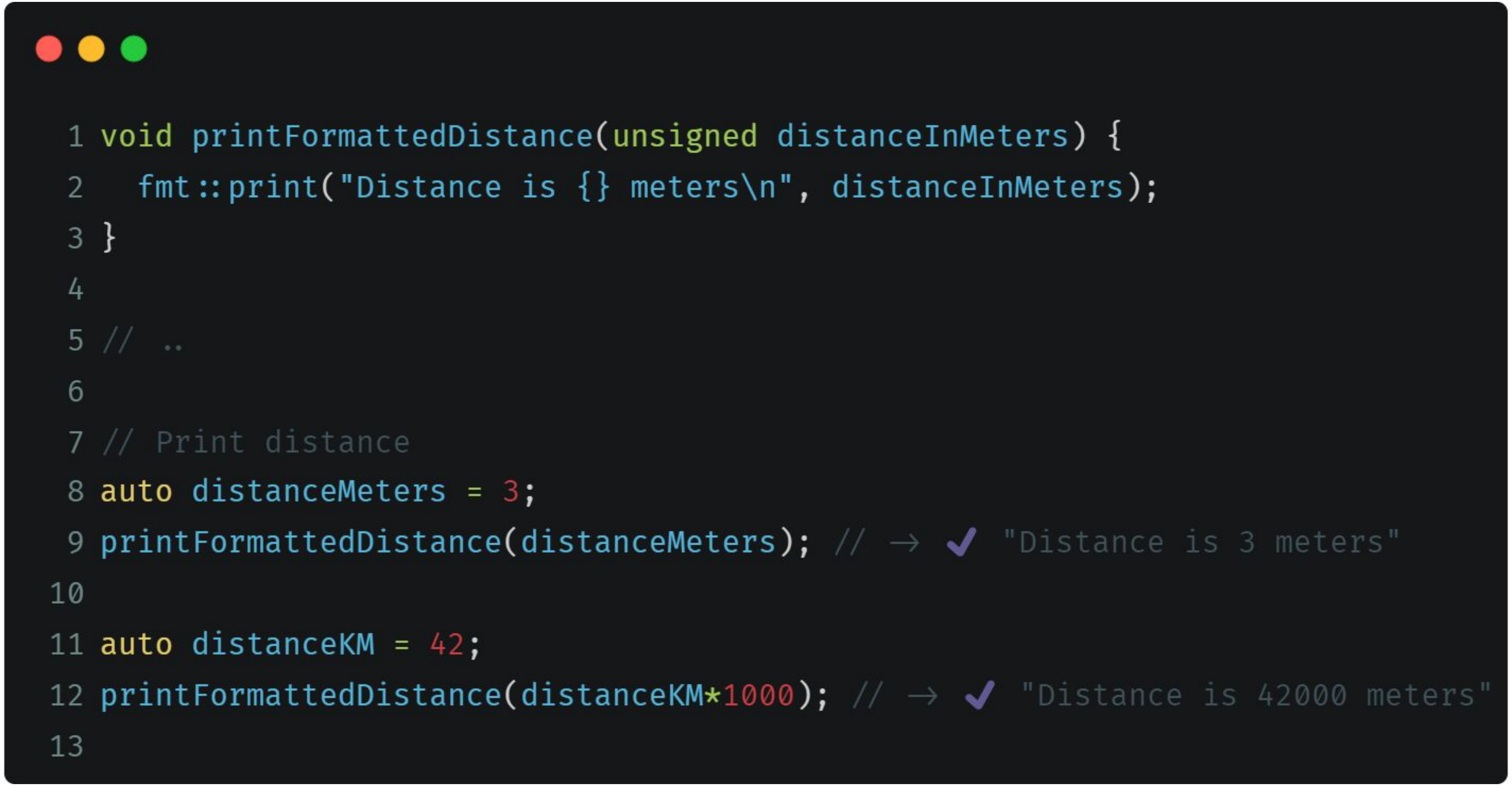
```cpp
void printData(unsigned value) {
  fmt::print("Distance is {} meters\n", distance);
}

// ..

// Print distance
auto distanceMeters = 3;
printData(distanceMeters); // → ✓ "Distance is 3 meters"




```

// 1. Better naming



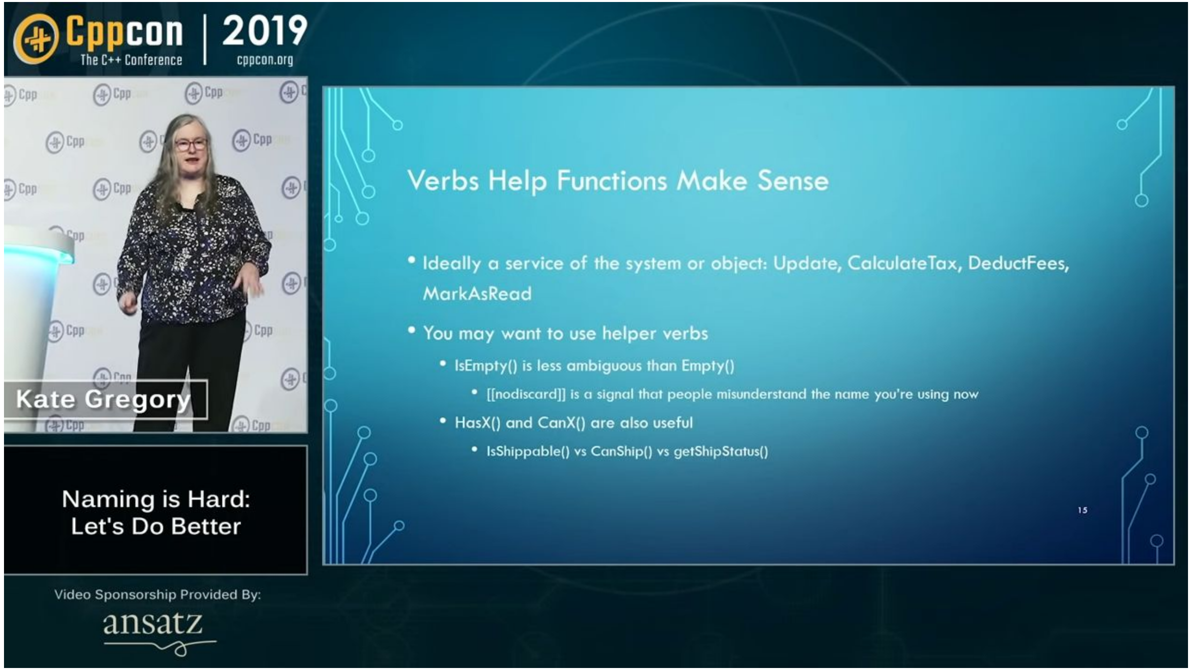
```cpp
void printData(unsigned value) {
  fmt::print("Distance is {} meters\n", distance);
}

// ..

// Print distance
auto distanceMeters = 3;
printData(distanceMeters); // → ✓ "Distance is 3 meters"

auto distanceKM = 42;
printData(distanceKM); // → ✗ "Distance is 42 meters"

```

// 1. Better naming



```
void printFormattedDistance(unsigned distanceInMeters) {
  fmt::print("Distance is {} meters\n", distanceInMeters);
}

// ..

// Print distance
auto distanceMeters = 3;
printFormattedDistance(distanceMeters); // → ✓ "Distance is 3 meters"

auto distanceKM = 42;
printFormattedDistance(distanceKM*1000); // → ✓ "Distance is 42000 meters"

```

// 1. Better naming

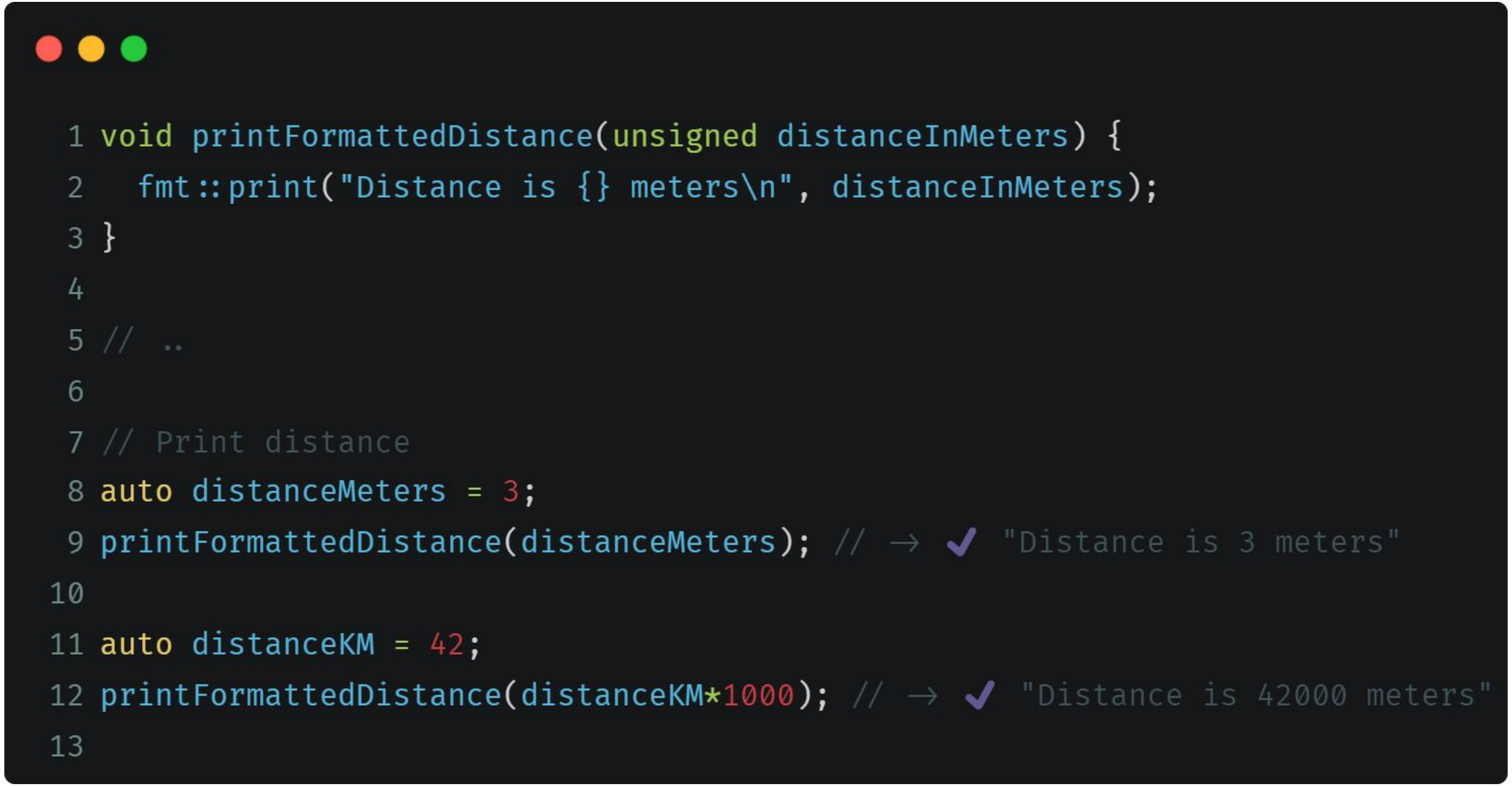

Cppcon The C++ Conference
2019 cppcon.org
Kate Gregory
Naming is Hard: Let's Do Better
Video Sponsorship Provided By:
ansatz
15

Verbs Help Functions Make Sense

- Ideally a service of the system or object: Update, CalculateTax, DeductFees, MarkAsRead
- You may want to use helper verbs
  - IsEmpty() is less ambiguous than Empty()
    - [[nodiscard]] is a signal that people misunderstand the name you're using now
  - HasX() and CanX() are also useful
    - IsShippable() vs CanShip() vs getShipStatus()

CppCon 2019: Kate Gregory “Naming is Hard: Let's Do Better”
https://www.youtube.com/watch?v=MBRoCdtZOYg

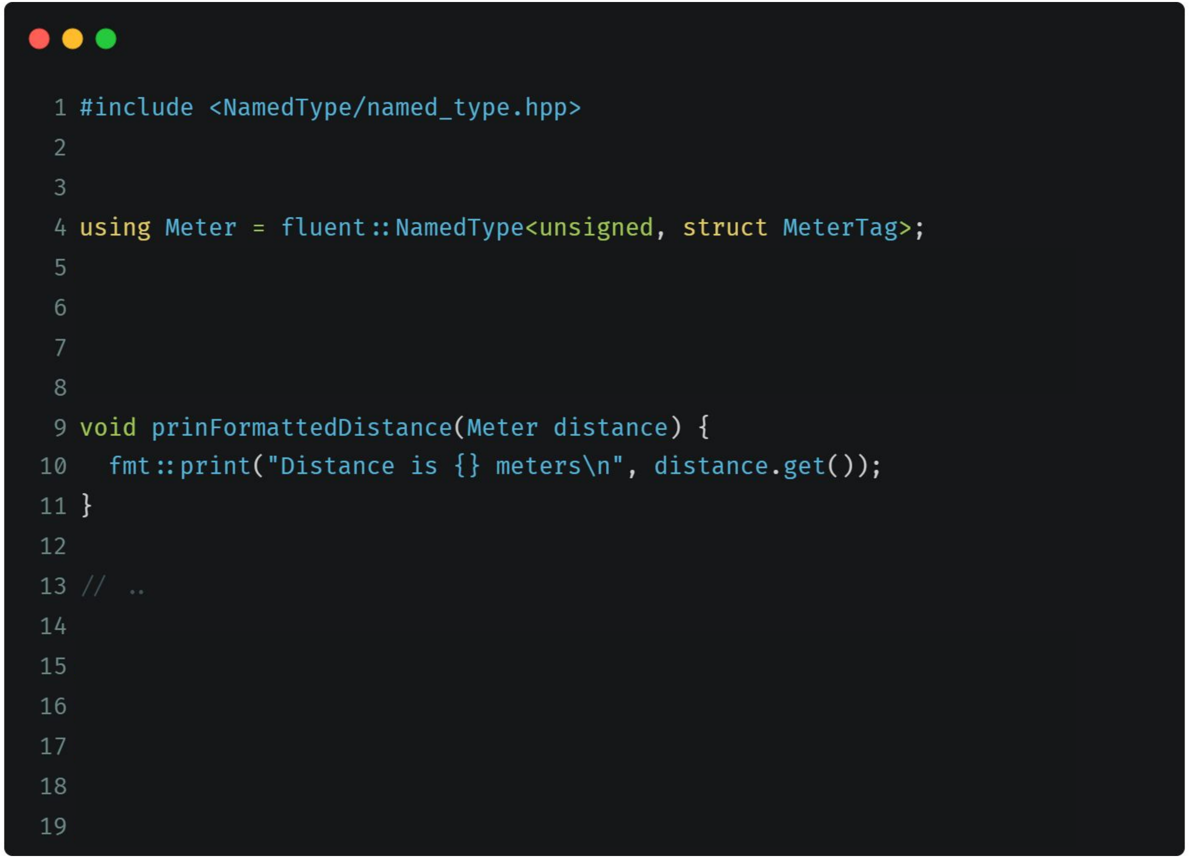
// 1. Better naming



```cpp
void printFormattedDistance(unsigned distanceInMeters) {
  fmt::print("Distance is {} meters\n", distanceInMeters);
}

// ..

// Print distance
auto distanceMeters = 3;
printFormattedDistance(distanceMeters); // → ✓ "Distance is 3 meters"

auto distanceKM = 42;
printFormattedDistance(distanceKM*1000); // → ✓ "Distance is 42000 meters"

```

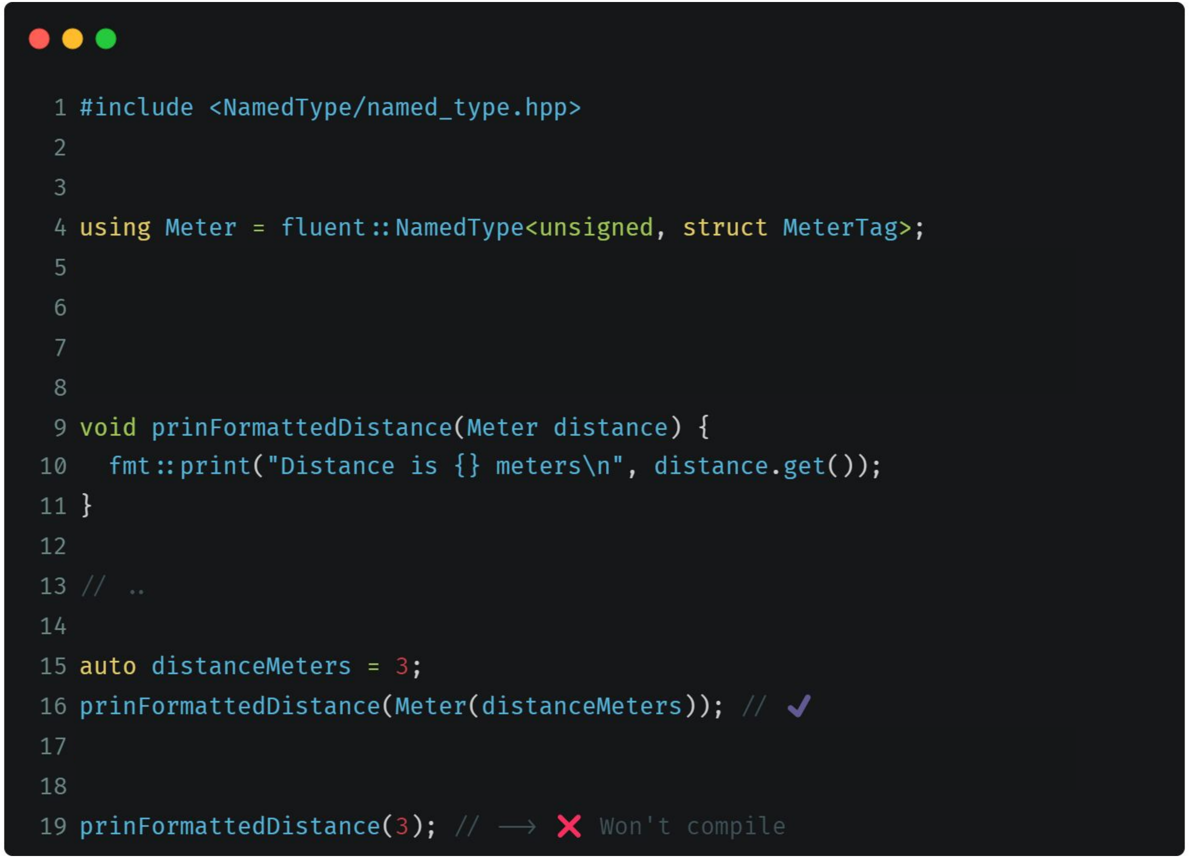
// 2. Use strong types

// 2. Use strong types



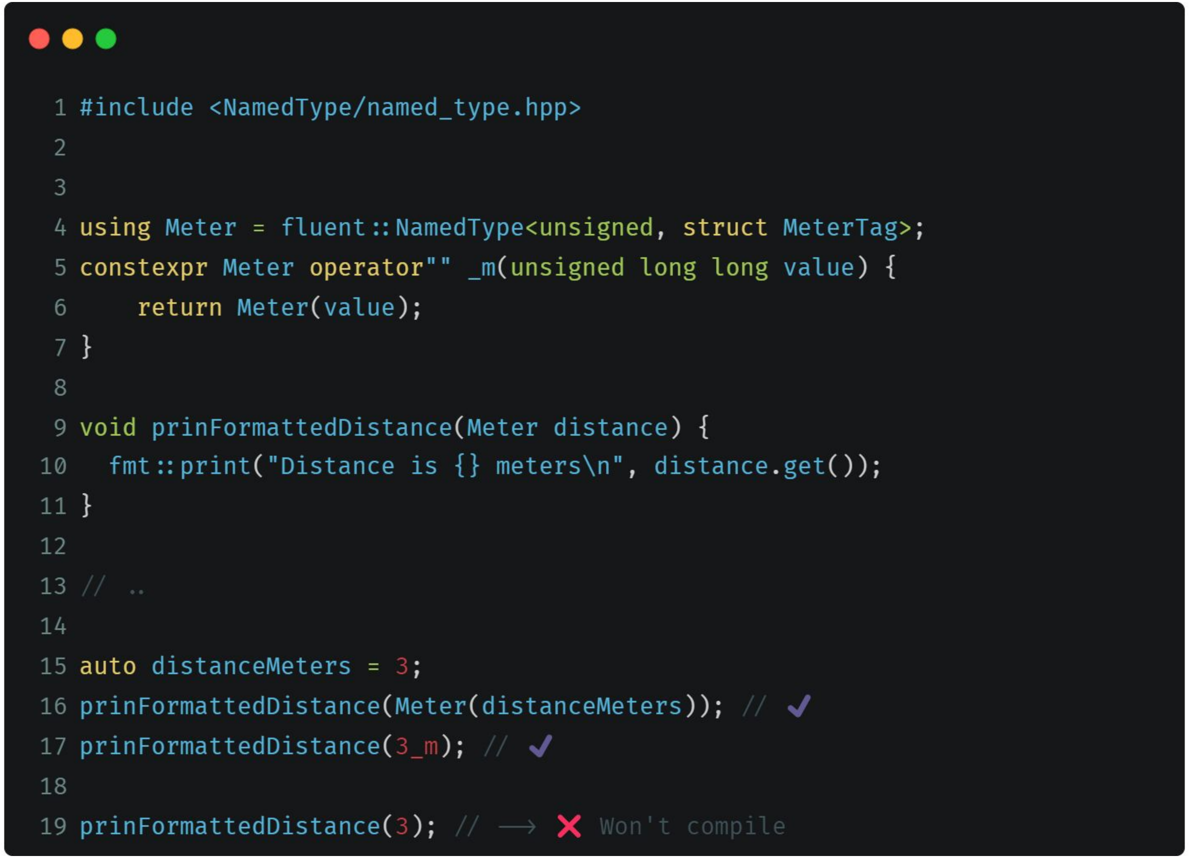
```
#include <NamedType/named_type.hpp>


using Meter = fluent::NamedType<unsigned, struct MeterTag>;




void prinFormattedDistance(Meter distance) {
  fmt::print("Distance is {} meters\n", distance.get());
}

// ..
```

// 2. Use strong types



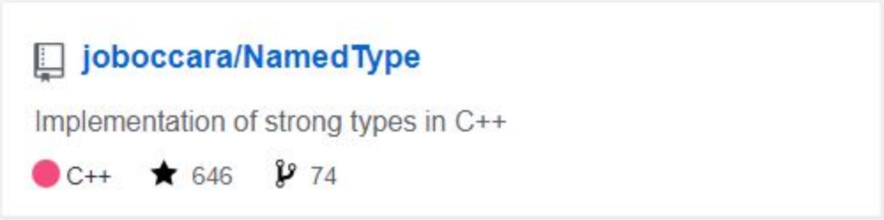
```cpp
#include <NamedType/named_type.hpp>


using Meter = fluent::NamedType<unsigned, struct MeterTag>;




void prinFormattedDistance(Meter distance) {
  fmt::print("Distance is {} meters\n", distance.get());
}

// ..

auto distanceMeters = 3;
prinFormattedDistance(Meter(distanceMeters)); // ✓


prinFormattedDistance(3); // ⟶ ❌ Won't compile
```

// 2. Use strong types



```cpp
#include <NamedType/named_type.hpp>


using Meter = fluent::NamedType<unsigned, struct MeterTag>;
constexpr Meter operator"" _m(unsigned long long value) {
    return Meter(value);
}

void prinFormattedDistance(Meter distance) {
  fmt::print("Distance is {} meters\n", distance.get());
}

// ..

auto distanceMeters = 3;
prinFormattedDistance(Meter(distanceMeters)); // ✓
prinFormattedDistance(3_m); // ✓

prinFormattedDistance(3); // ⟶ ❌ Won't compile
```

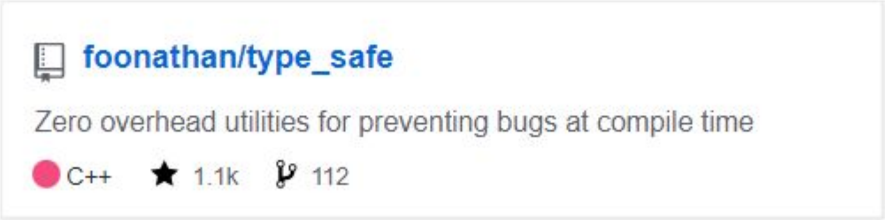
// 2. Use strong types



joboccara/NamedType

Implementation of strong types in C++

● C++ ★ 646 ⑂ 74



foonathan/type_safe

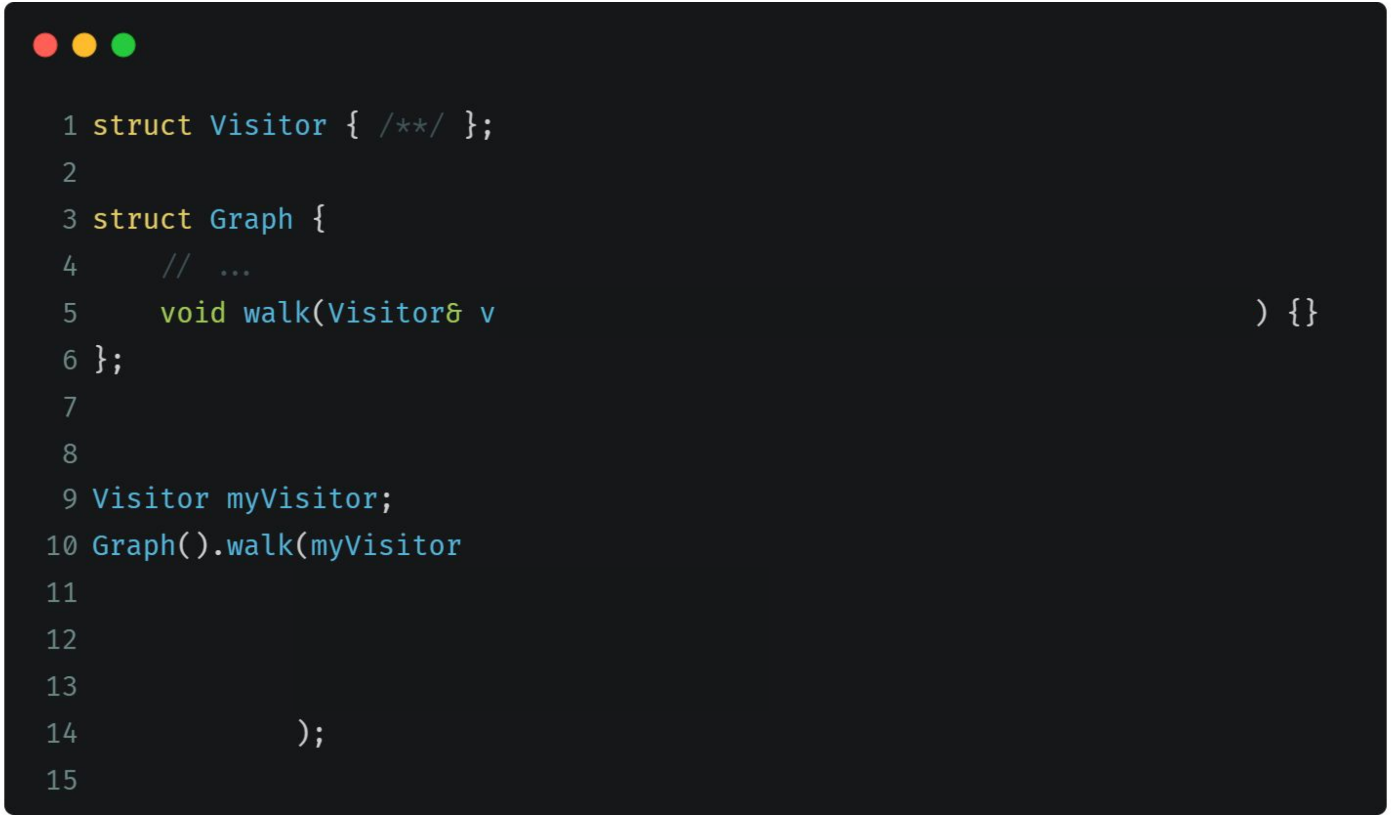
Zero overhead utilities for preventing bugs at compile time

● C++ ★ 1.1k ⑂ 112

https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/
https://www.foonathan.net/2016/10/strong-typedefs/

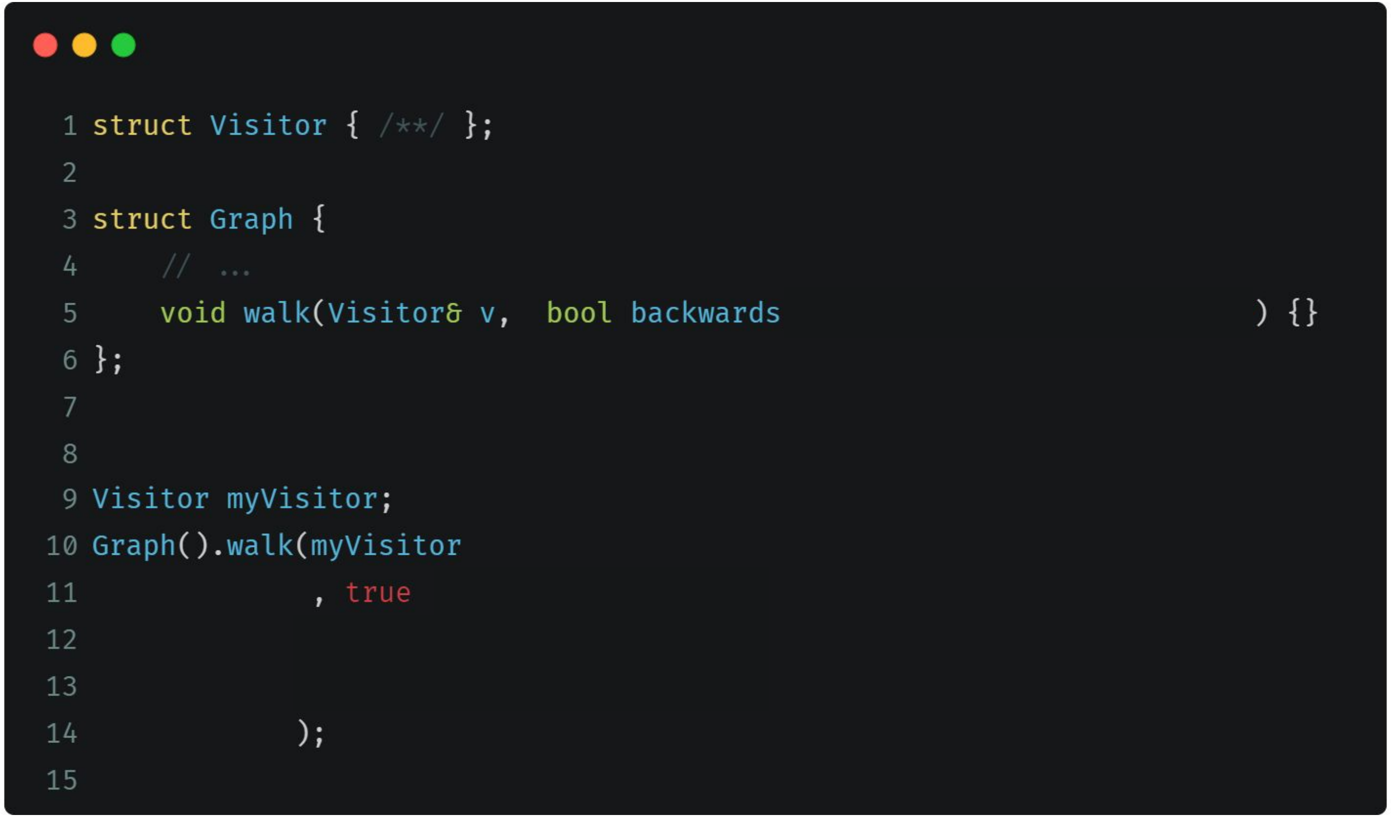
// 3. Avoid easily swappable parameters

// 3. Avoid easily swappable parameters



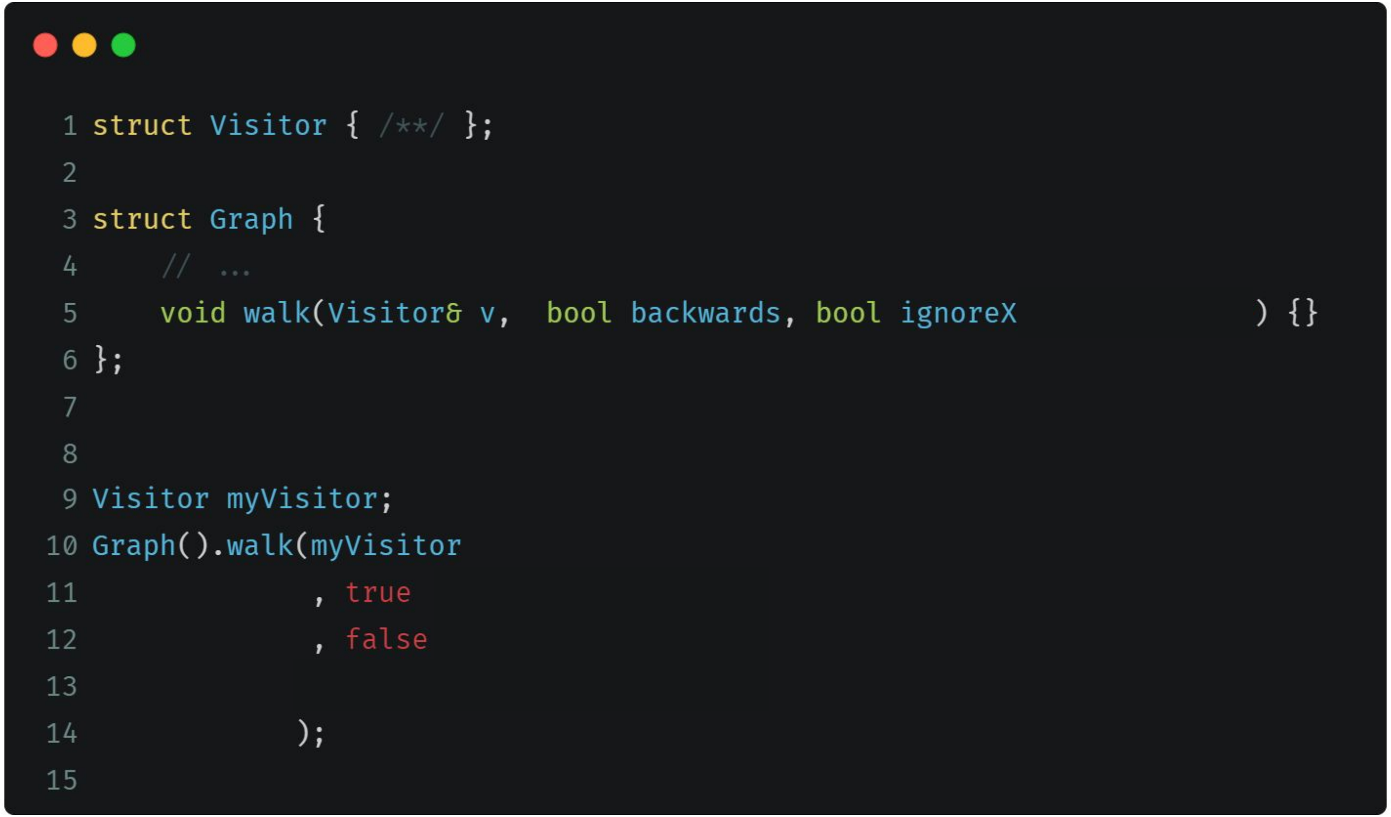
```
struct Visitor { /**/ };

struct Graph {
    // ...
    void walk(Visitor& v                                              ) {}
};


Visitor myVisitor;
Graph().walk(myVisitor



            );

```

// 3. Avoid easily swappable parameters



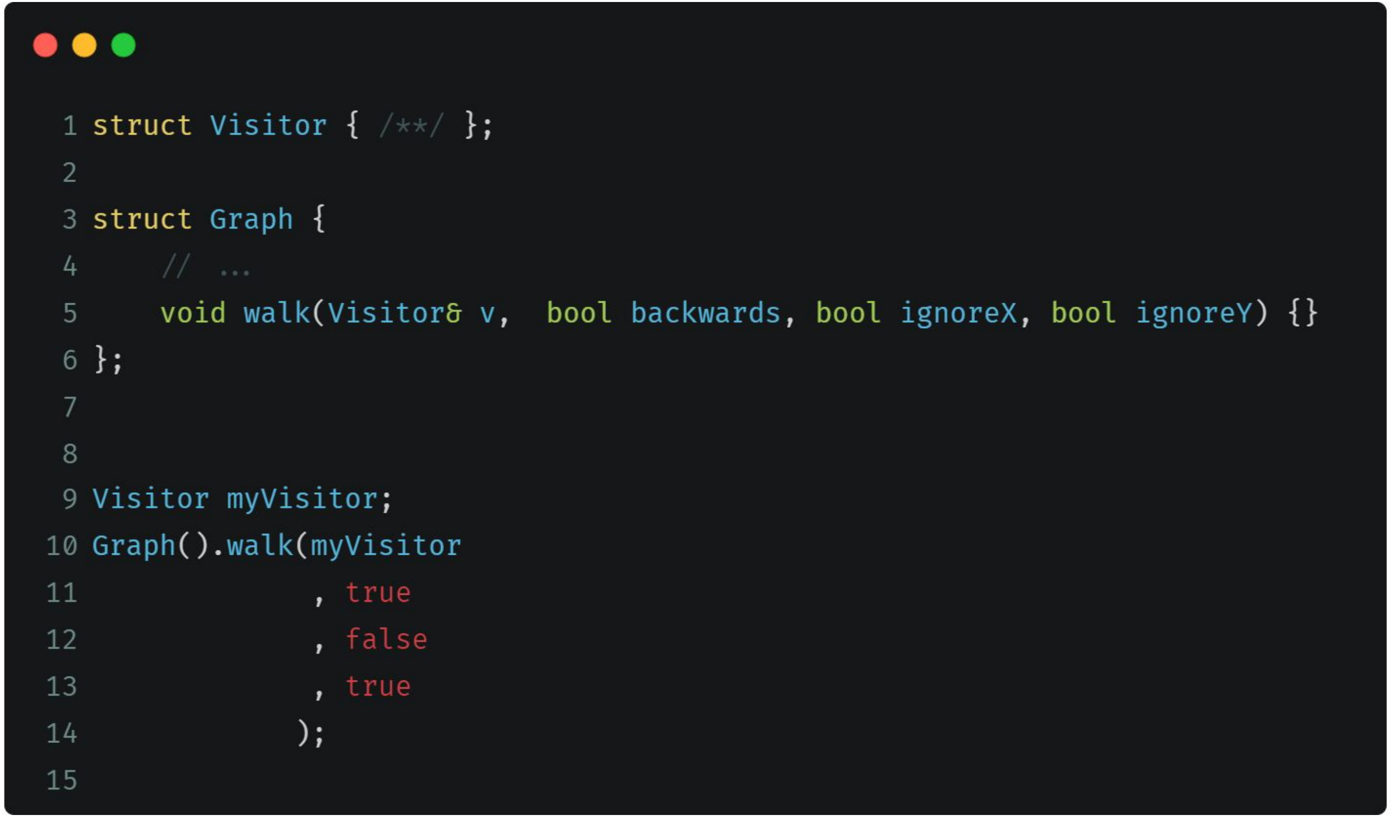
```
struct Visitor { /**/ };

struct Graph {
    // ...
    void walk(Visitor& v,  bool backwards                             ) {}
};


Visitor myVisitor;
Graph().walk(myVisitor
             , true


            );

```

// 3. Avoid easily swappable parameters



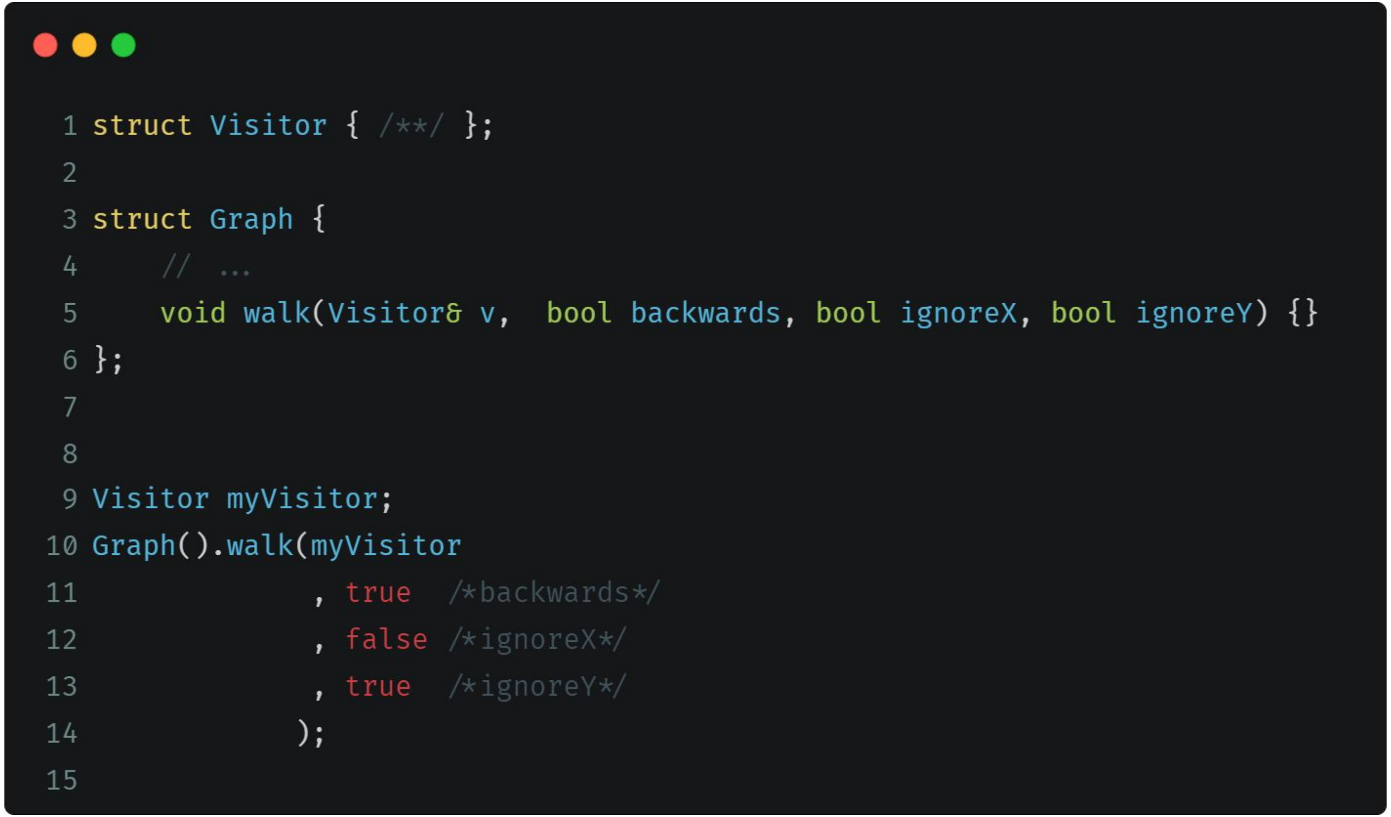
```cpp
struct Visitor { /**/ };

struct Graph {
    //  ...
    void walk(Visitor& v,  bool backwards, bool ignoreX               ) {}
};


Visitor myVisitor;
Graph().walk(myVisitor
             , true
             , false

            );

```

// 3. Avoid easily swappable parameters



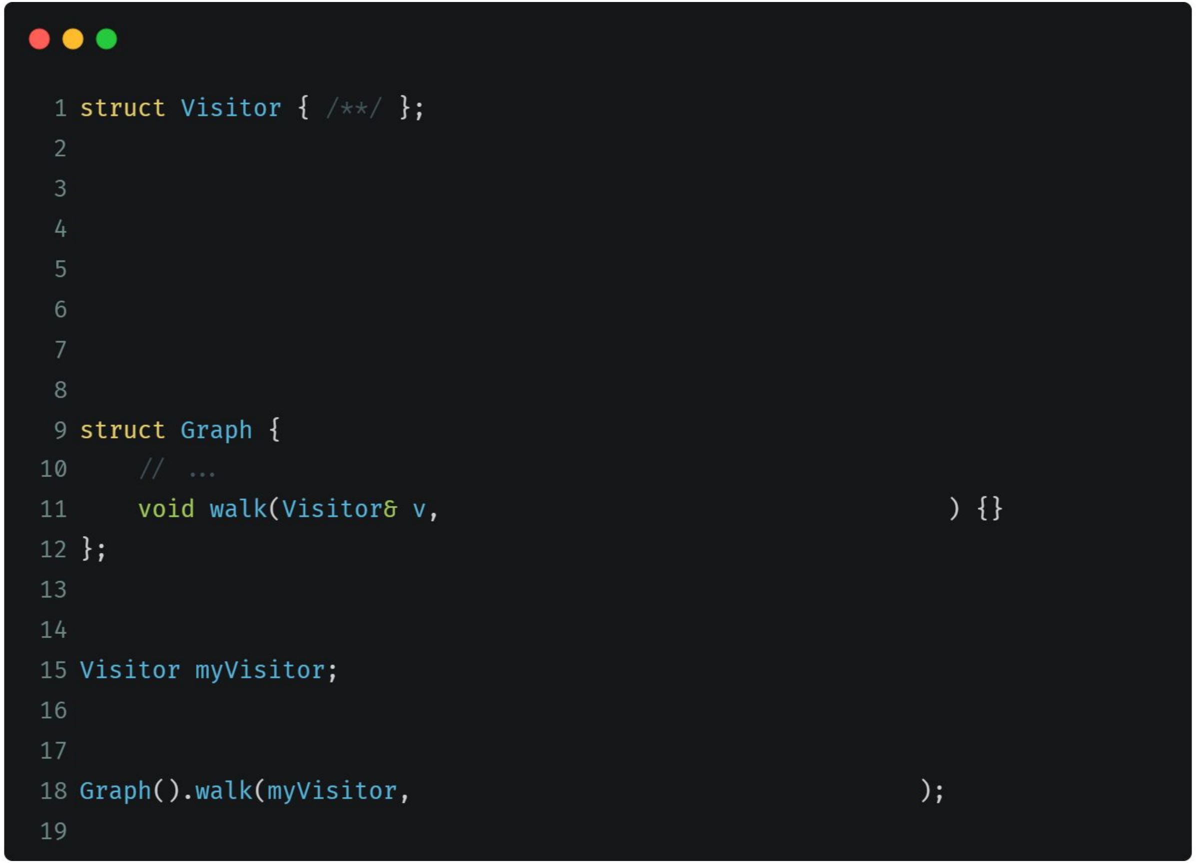
```cpp
struct Visitor { /**/ };

struct Graph {
    // ...
    void walk(Visitor& v,  bool backwards, bool ignoreX, bool ignoreY) {}
};


Visitor myVisitor;
Graph().walk(myVisitor
             , true
             , false
             , true
            );

```

// 3. Avoid easily swappable parameters



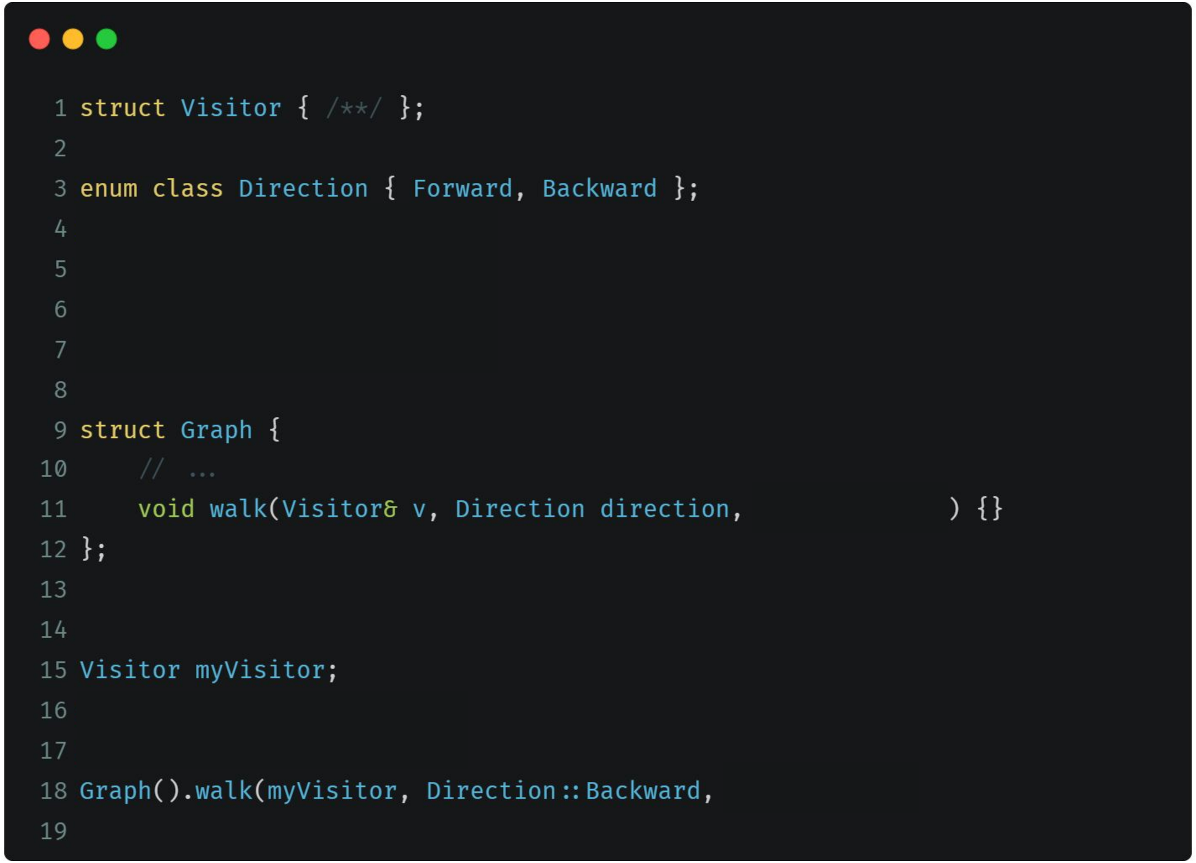
```cpp
struct Visitor { /**/ };

struct Graph {
    //  ...
    void walk(Visitor& v,  bool backwards, bool ignoreX, bool ignoreY) {}
};


Visitor myVisitor;
Graph().walk(myVisitor
             , true  /*backwards*/
             , false /*ignoreX*/
             , true  /*ignoreY*/
            );

```

// 3. Avoid easily swappable parameters



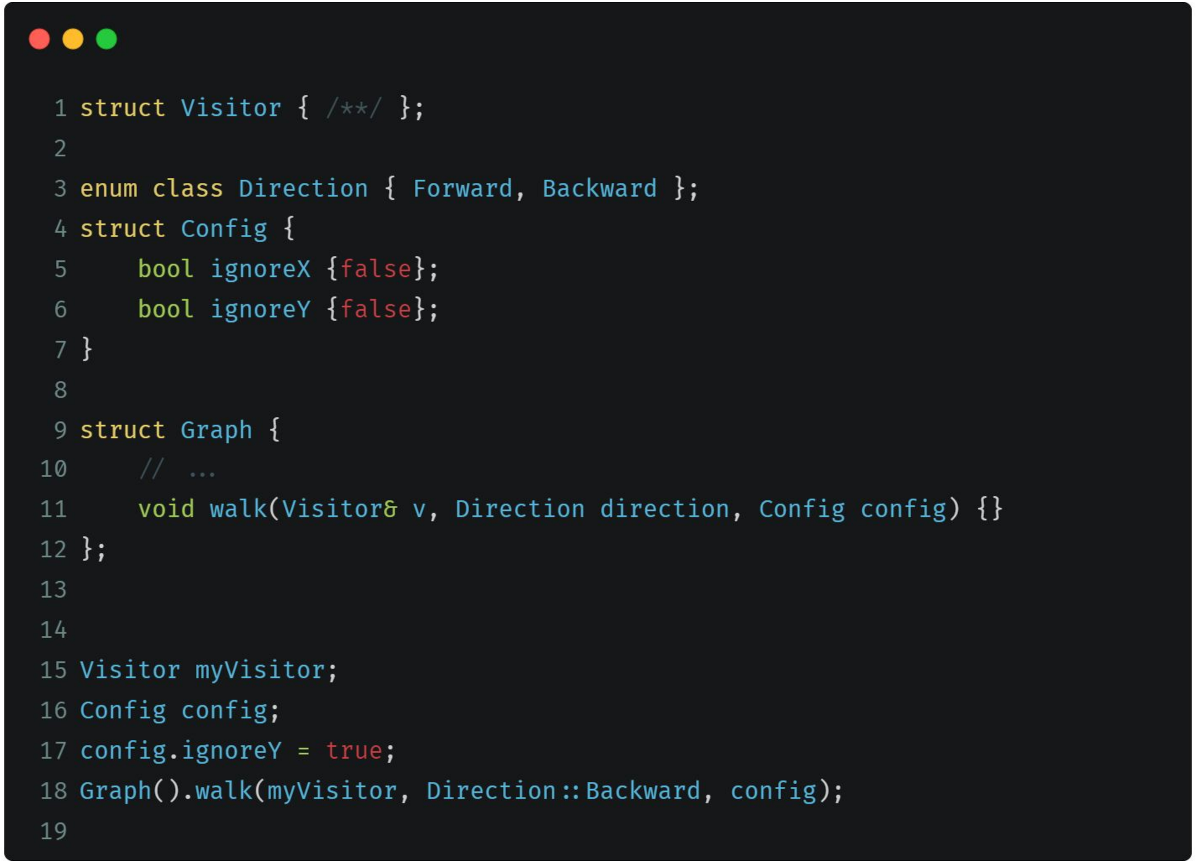
```cpp
struct Visitor { /**/ };







struct Graph {
    // ...
    void walk(Visitor& v,                                   ) {}
};


Visitor myVisitor;


Graph().walk(myVisitor,                                   );

```

// 3. Avoid easily swappable parameters



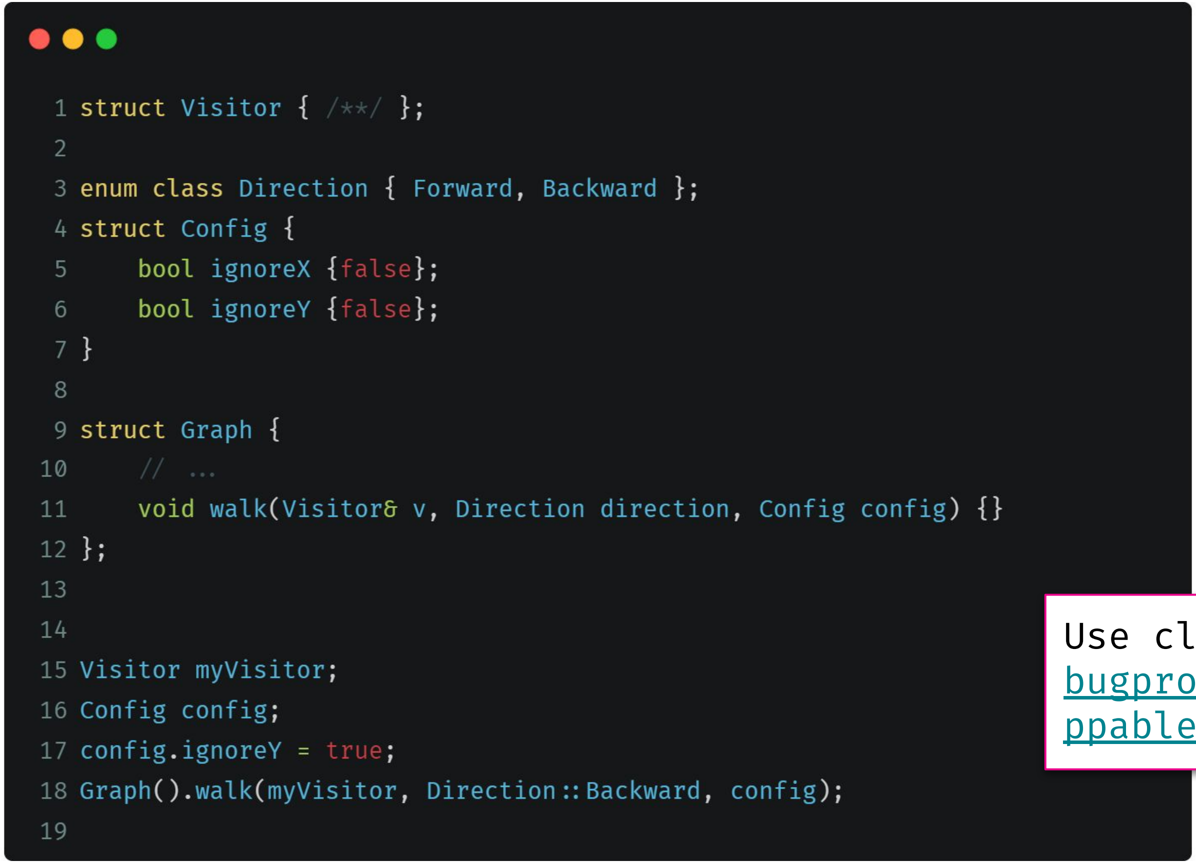
```cpp
struct Visitor { /**/ };

enum class Direction { Forward, Backward };





struct Graph {
    // ...
    void walk(Visitor& v, Direction direction,       ) {}
};


Visitor myVisitor;


Graph().walk(myVisitor, Direction::Backward,

```

// 3. Avoid easily swappable parameters



```cpp
struct Visitor { /**/ };

enum class Direction { Forward, Backward };
struct Config {
    bool ignoreX {false};
    bool ignoreY {false};
}

struct Graph {
    // ...
    void walk(Visitor& v, Direction direction, Config config) {}
};


Visitor myVisitor;
Config config;
config.ignoreY = true;
Graph().walk(myVisitor, Direction::Backward, config);

```

// 3. Avoid easily swappable parameters

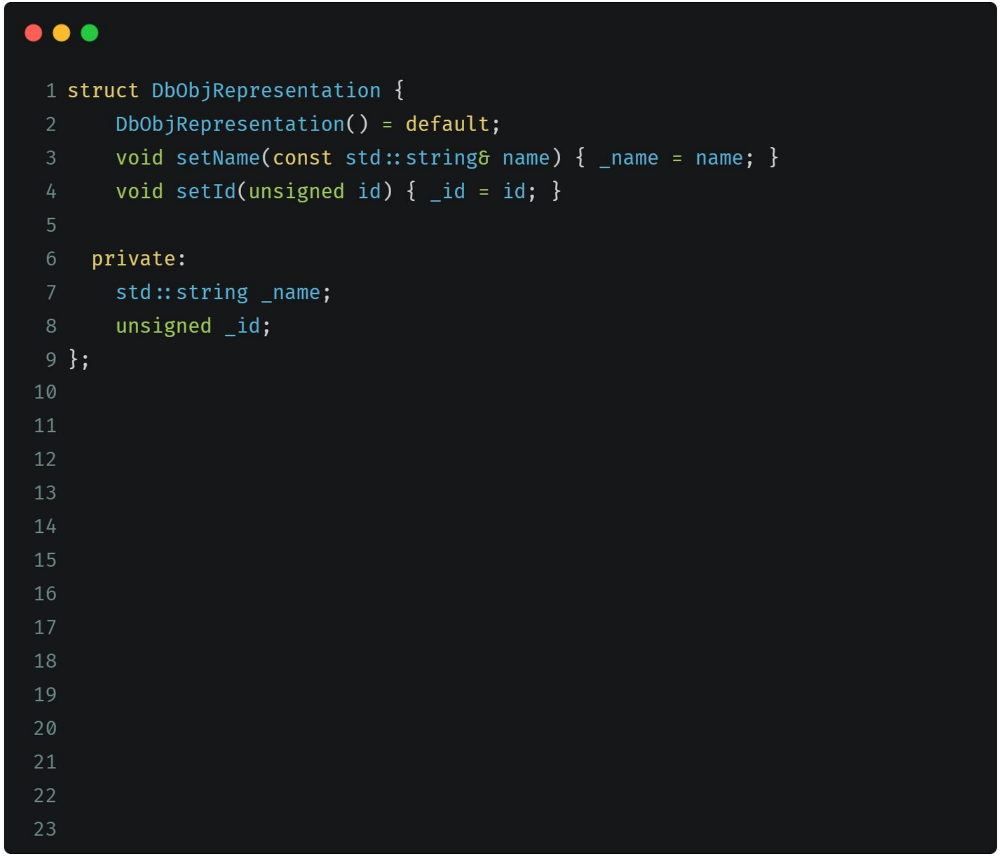


```cpp
struct Visitor { /**/ };

enum class Direction { Forward, Backward };
struct Config {
    bool ignoreX {false};
    bool ignoreY {false};
}

struct Graph {
    // ...
    void walk(Visitor& v, Direction direction, Config config) {}
};


Visitor myVisitor;
Config config;
config.ignoreY = true;
Graph().walk(myVisitor, Direction::Backward, config);
```

Use clang-tidy: bugprone-easily-swappable-parameters

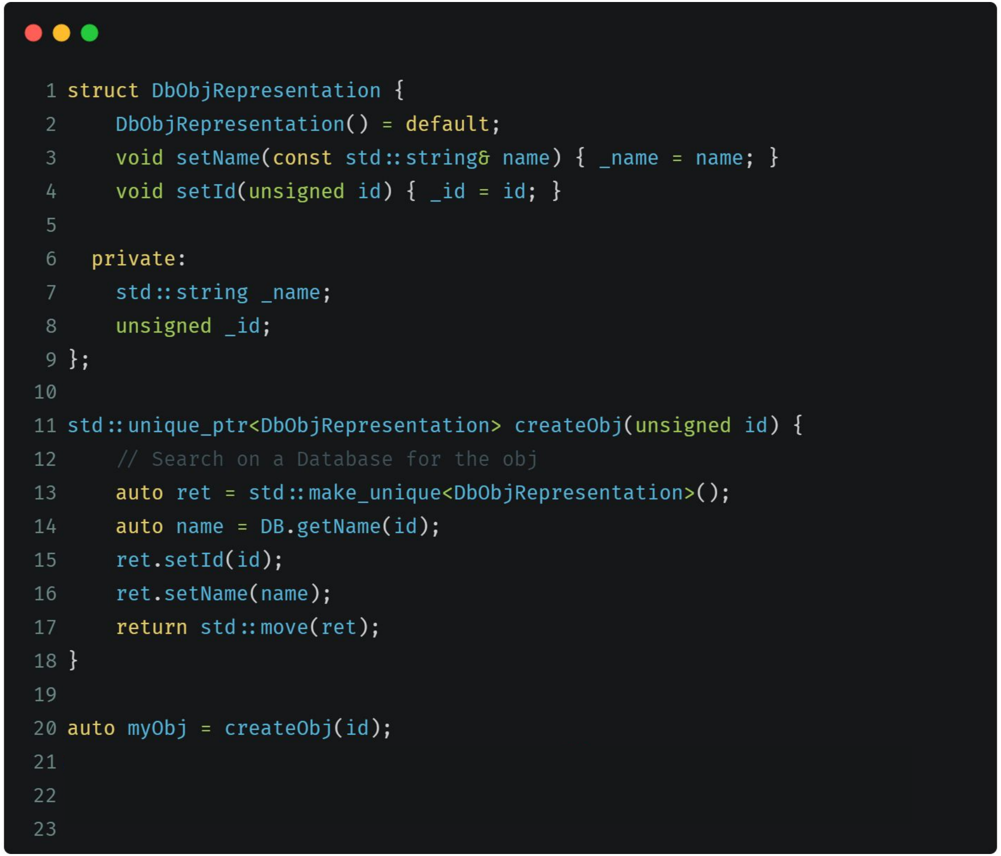
// 4. Carefully think about intent

// 4. Carefully think about intent



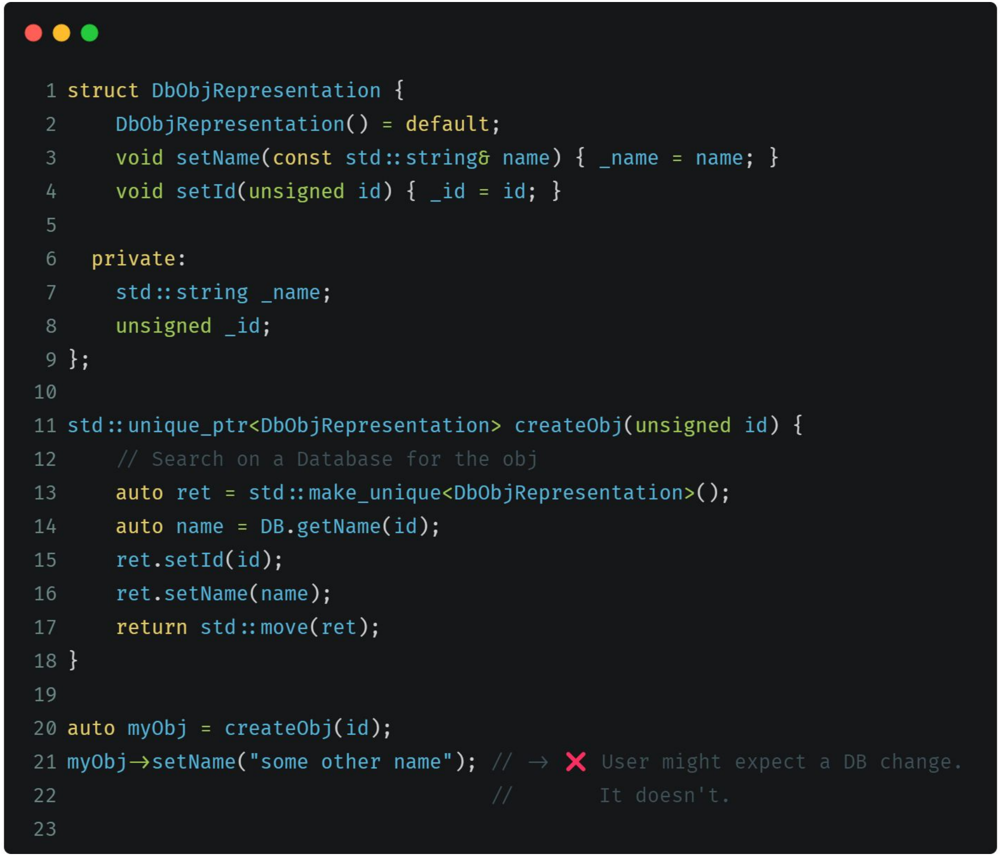
```
struct DbObjRepresentation {
    DbObjRepresentation() = default;
    void setName(const std::string& name) { _name = name; }
    void setId(unsigned id) { _id = id; }

  private:
    std::string _name;
    unsigned _id;
};
```

// 4. Carefully think about intent



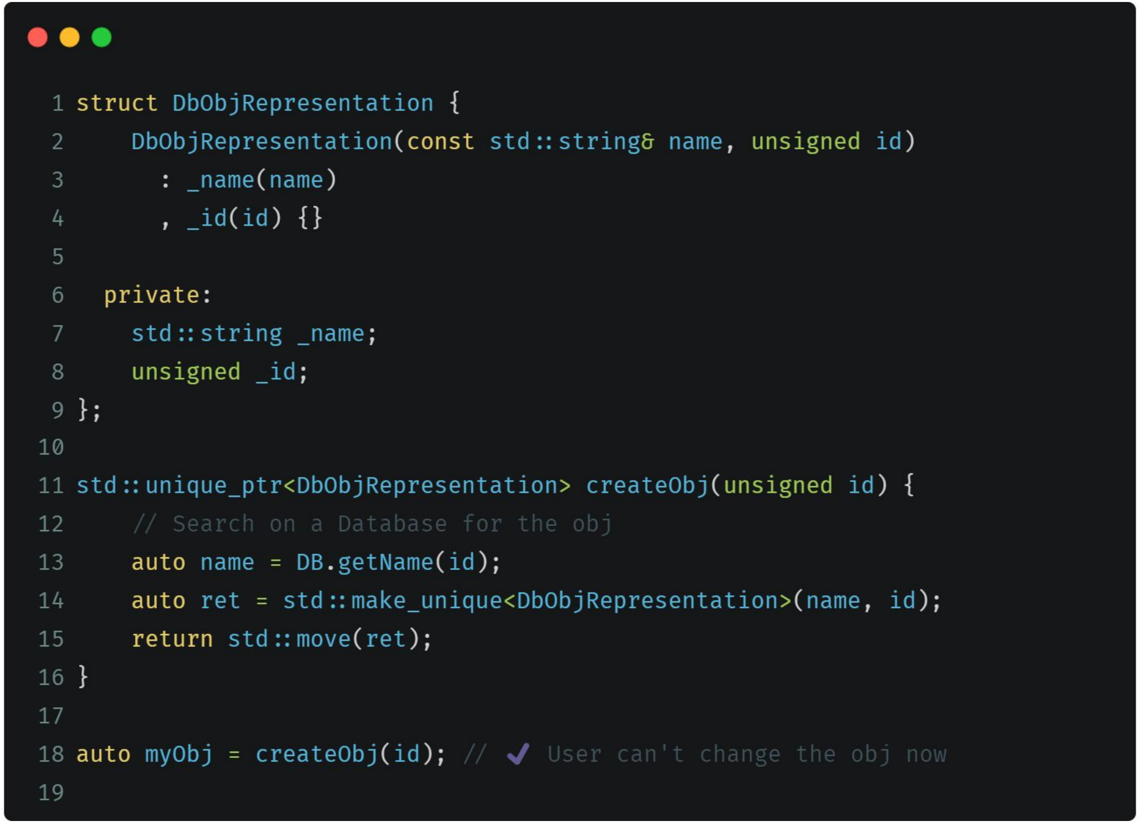
```
struct DbObjRepresentation {
    DbObjRepresentation() = default;
    void setName(const std::string& name) { _name = name; }
    void setId(unsigned id) { _id = id; }

  private:
    std::string _name;
    unsigned _id;
};

std::unique_ptr<DbObjRepresentation> createObj(unsigned id) {
    // Search on a Database for the obj
    auto ret = std::make_unique<DbObjRepresentation>();
    auto name = DB.getName(id);
    ret.setId(id);
    ret.setName(name);
    return std::move(ret);
}

auto myObj = createObj(id);



```

// 4. Carefully think about intent



```cpp
struct DbObjRepresentation {
    DbObjRepresentation() = default;
    void setName(const std::string& name) { _name = name; }
    void setId(unsigned id) { _id = id; }

  private:
    std::string _name;
    unsigned _id;
};

std::unique_ptr<DbObjRepresentation> createObj(unsigned id) {
    // Search on a Database for the obj
    auto ret = std::make_unique<DbObjRepresentation>();
    auto name = DB.getName(id);
    ret.setId(id);
    ret.setName(name);
    return std::move(ret);
}

auto myObj = createObj(id);
myObj->setName("some other name"); // -> ❌ User might expect a DB change.
                                   //       It doesn't.

```

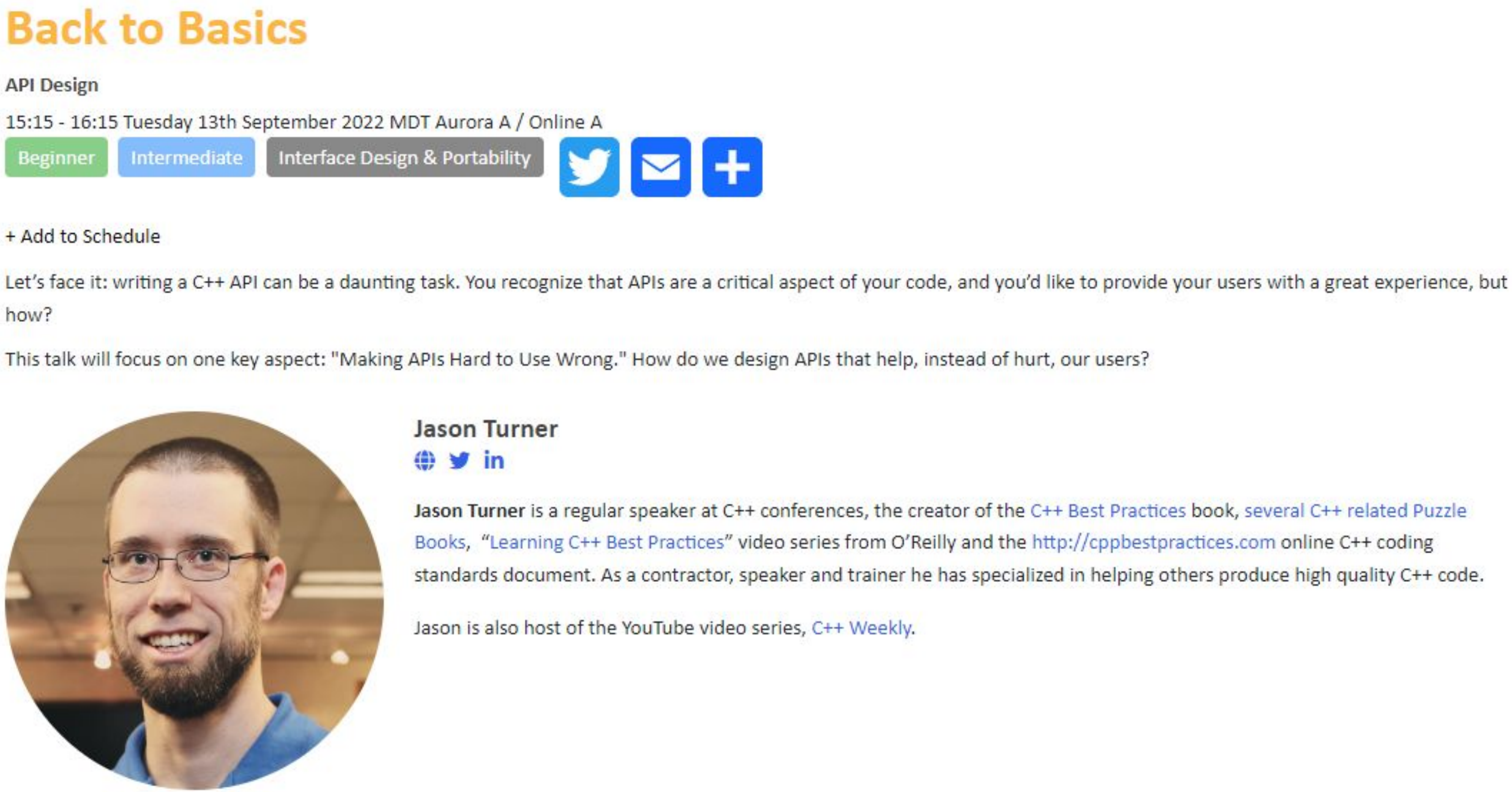
// 4. Carefully think about intent



```cpp
struct DbObjRepresentation {
    DbObjRepresentation(const std::string& name, unsigned id)
      : _name(name)
      , _id(id) {}

  private:
    std::string _name;
    unsigned _id;
};

std::unique_ptr<DbObjRepresentation> createObj(unsigned id) {
    // Search on a Database for the obj
    auto name = DB.getName(id);
    auto ret = std::make_unique<DbObjRepresentation>(name, id);
    return std::move(ret);
}

auto myObj = createObj(id); // ✓ User can't change the obj now
```

// 5. Check out more content

// 5. Check out more content



Back to Basics

API Design

15:15 - 16:15 Tuesday 13th September 2022 MDT Aurora A / Online A

Beginner Intermediate Interface Design & Portability

+ Add to Schedule

Let's face it: writing a C++ API can be a daunting task. You recognize that APIs are a critical aspect of your code, and you'd like to provide your users with a great experience, but how?

This talk will focus on one key aspect: "Making APIs Hard to Use Wrong." How do we design APIs that help, instead of hurt, our users?

Jason Turner

Jason Turner is a regular speaker at C++ conferences, the creator of the C++ Best Practices book, several C++ related Puzzle Books, "Learning C++ Best Practices" video series from O'Reilly and the http://cppbestpractices.com online C++ coding standards document. As a contractor, speaker and trainer he has specialized in helping others produce high quality C++ code.

Jason is also host of the YouTube video series, C++ Weekly.

Thank you!

Thamara Andrade

twitter: @thamyk

site: thamara.dev